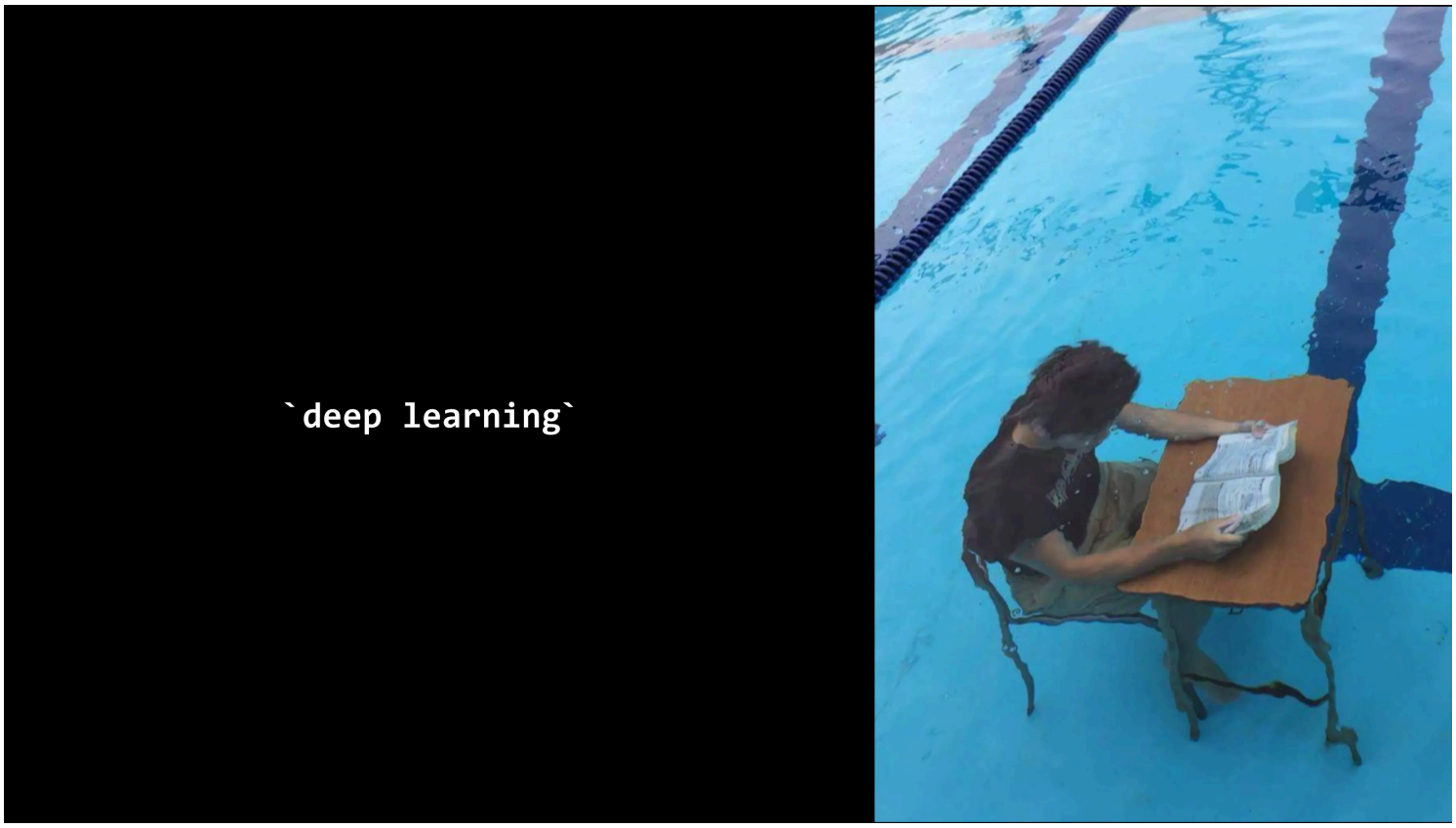
Build RAG Application Using a LLM Running on Local Computer with Ollama and Langchain

Privacy-preserving LLM without GPU

(λx.x)eranga · Follow

Published in Effectz.AI · 16 min read · Mar 17, 2024



Background

In my previous post, I explored how to develop a Retrieval-Augmented Generation (RAG) application by leveraging a locally-run Large Language Model (LLM) through GPT-4All and Langchain. This time, I will demonstrate building the same RAG application using a different tool, `Ollama`. All source codes related to this post have been published on GitLab. Please clone the repository to follow along with the post.

Ollama

`Ollama` is a lightweight and flexible framework designed for the local deployment of LLM on personal computers. It simplifies the development, execution, and management of LLMs with an intuitive API and provides a collection of pre-configured models ready for immediate use across a variety of applications. Central to its design is the bundling of model weights,

configurations, and data into a unified package, encapsulated within a `Modelfile`.

The framework features a curated assortment of pre-quantized, optimized models, such as `Llama 2`, `Mistral`, and `Gemma`, which are ready for deployment. These models are specifically engineered to operate on standard consumer hardware, spanning CPUs and GPUs, and are compatible with multiple operating systems, including macOS, Linux, and Windows. This approach negates the necessity for users to undertake intricate model optimization tasks themselves.

Given that LLMs typically demand robust GPUs for their operation due to their considerable size, the models supported by Ollama employ neural network quantization. This technique dramatically reduces the hardware requirements, allowing LLMs to function efficiently on common computing devices without an internet connection. Ollama thus makes it more accessible to LLM technologies, enabling both individuals and organizations to leverage these advanced models on consumer-grade hardware.

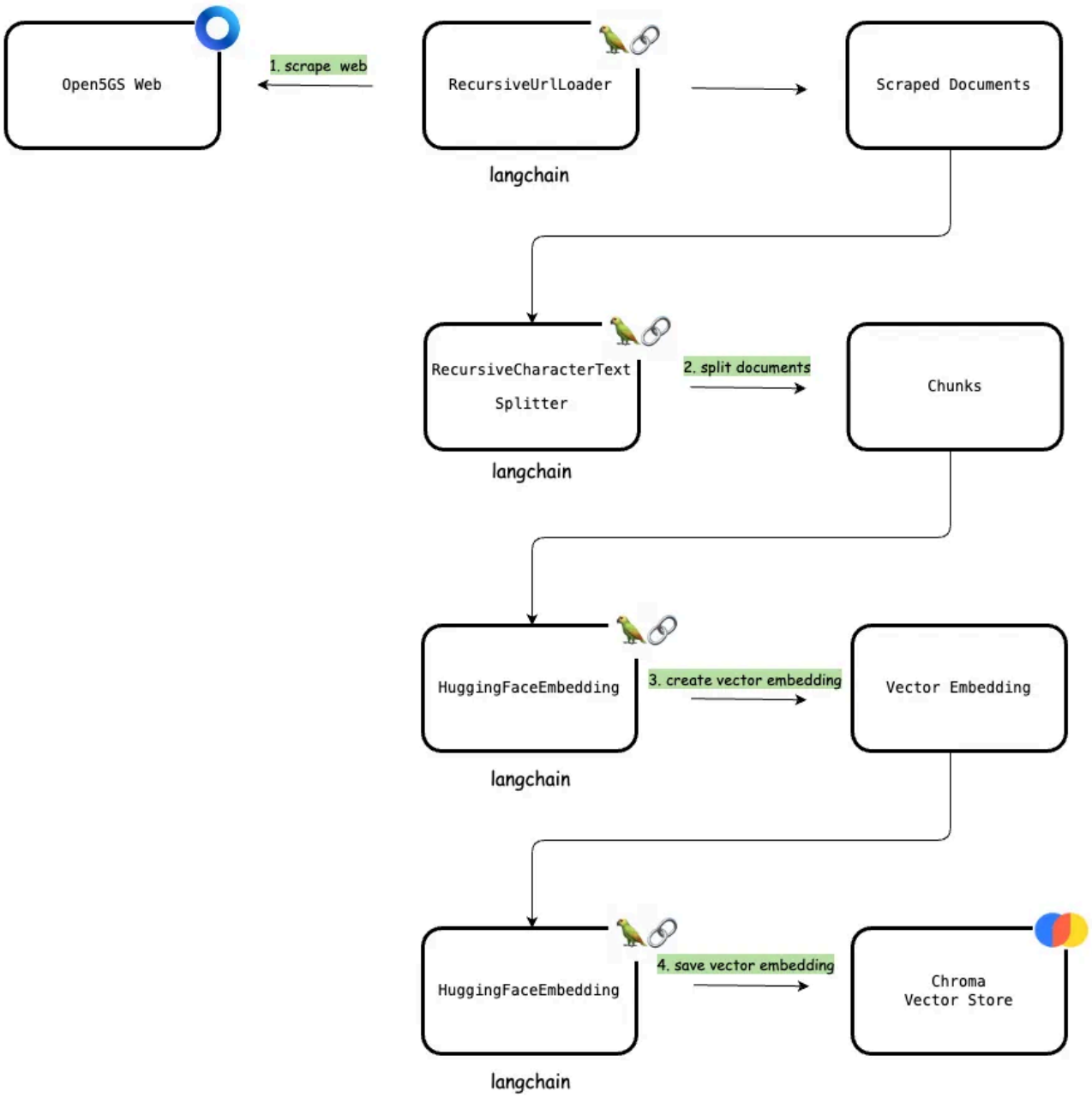
RAG Application

This RAG application incorporates a custom-made dataset, which is dynamically scraped from an online website. Users can interact with the website's data through the an API(e.g REST API). For demonstration purposes, I've selected the `Open5GS documentation` website (Open5GS is a C-language implementation of the 5G Core). The data from the Open5GS documentation is scrap, split, and then stored in the `Chroma` vector database as vector embeddings. Consequently, users can seamlessly interact with the content of the Open5GS documentation via the API.

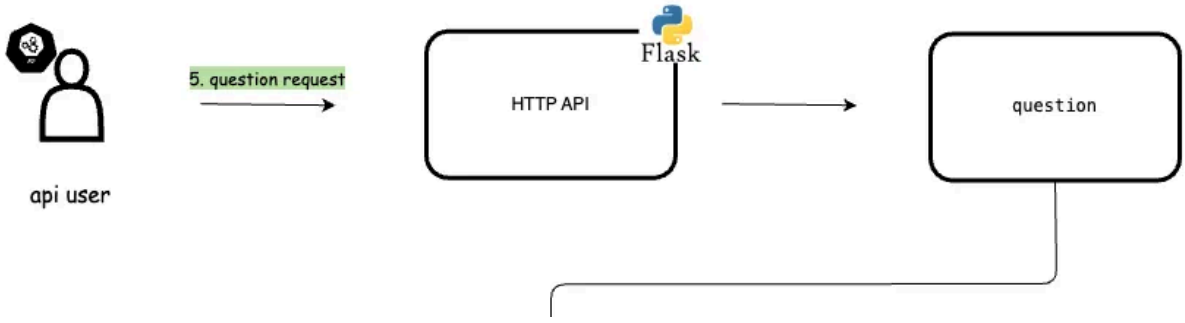
For the LLM component of this RAG application, I've opted for the `Llama2 7B` model, which is run through Ollama. Llama2 which runing on Ollama is Meta's Llama-2 based LLM, quantized for optimal performance on consumer-grade hardware, such as CPUs. In this RAG application, the Llama2 LLM which running with Ollama provides answers to user questions based on the content in the `Open5GS` documentation. The integration of the RAG application and LLM facilitated through `Langchain`.

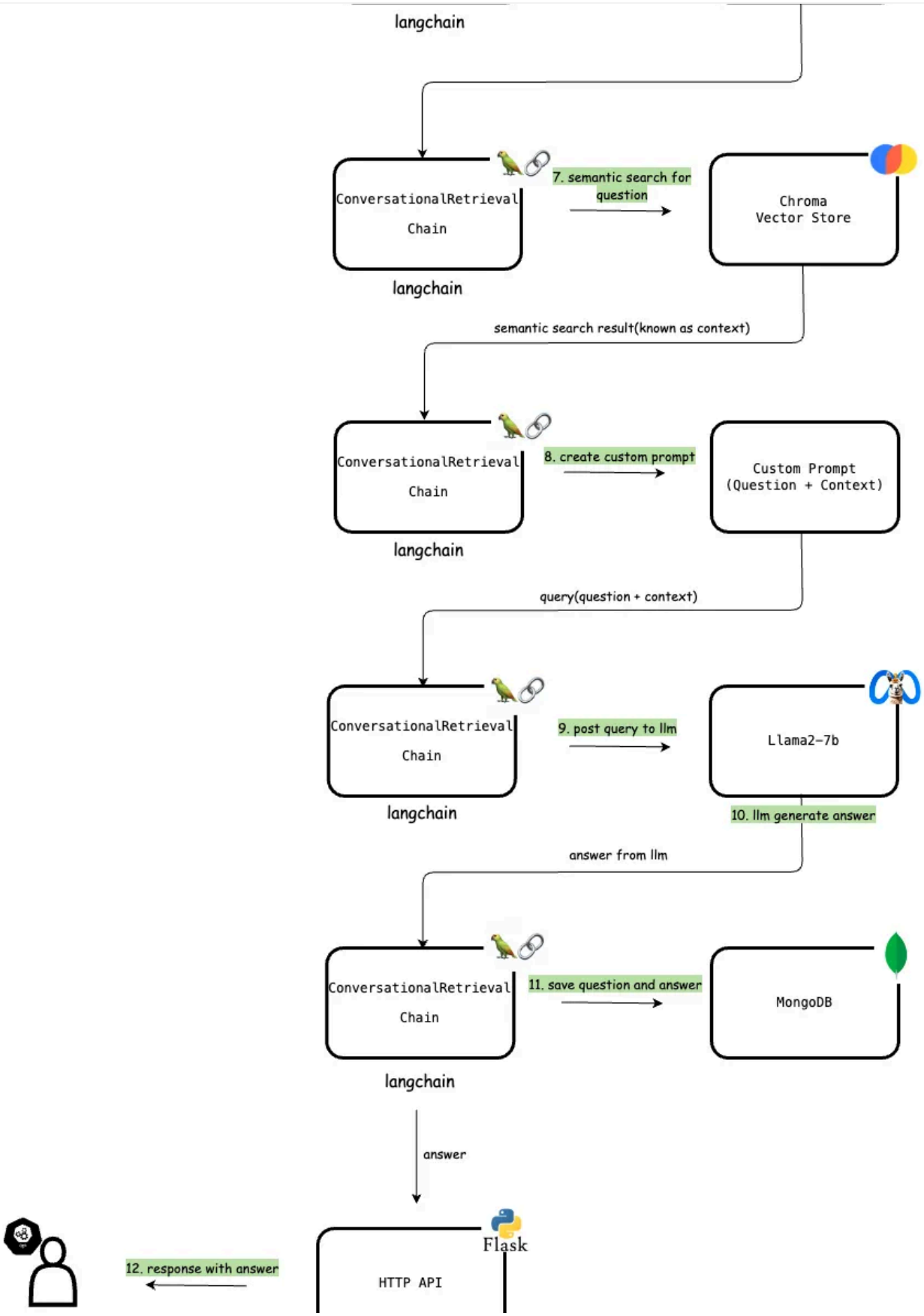
Following are the main functionalities of the RAG application. A comprehensive functional architecture, encompassing these various components, is detailed in the figure below.

data indexing



data querying





1. Scrape Web Data

Langchain provide different types of document loaders to load data from different source as `Document`'s. `RecursiveUrlLoader` is one such document loader that can be used to load the data in web url into documents. This step employs Langchain's `RecursiveUrlLoader` to scrape data from the web as documents. `RecursiveUrlLoader` scrapes the given url recursively in to given `max_depth` and read the data on the web. This data used to create vector embedding and answer questions of the user.

2. Split Documents

When handling lengthy pieces of text, it's essential to divide the text into smaller segments. Although this task seems straightforward, it can encompass considerable complexity. The goal is to ensure that semantically related segments of text remain together. The Langchain text splitter accomplishes this task effectively. Essentially, it divides the text into small, semantically meaningful units (often sentences). These smaller segments are then combined to form larger chunks until they reach a certain size, determined by a specific function. Upon reaching this size, the chunk is designated as an individual piece of text, and the process begins anew with some overlap. For this particular scenario, I have employed the `RecursiveCharacterTextSplitter` to split the scraped documents into manageable chunks.

3. Create Vector Embedding

Once the data is collected and split, the next step involves converting this textual information into vector embeddings. These embeddings are then created from the split data. Text embeddings are crucial to the functioning of LLM operations. While it's technically feasible to work with language models

using natural language, storing and retrieving such data is highly inefficient. To enhance efficiency, it's necessary to transform text data into vector form. There are dedicated machine learning models specifically designed for creating embeddings from text. In this case, I have utilized open-souce `HuggingFaceEmbedding` model `all-MiniLM-L6-v2` to generate vector embeddings. The text is thereby converted into multidimensional vectors, which are essentially high-dimensional numerical representations capturing semantic meanings and contextual nuances. Once embedded, these data can be grouped, sorted, searched, and more. We can calculate the distance between two sentences to determine their degree of relatedness. Importantly, these operations transcend traditional database searches that rely on keywords, capturing instead the semantic closeness between sentences.

4. Store Vector Embedding in Chroma

The generated vector embeddings are then stored in the `Chroma` vector database. Chroma(commonly referred to as ChromaDB) is an open-source embedding database that makes it easy to build LLM apps by storing and retrieving embeddings and their metadata, as well as documents and queries. Chroma efficiently handles these embeddings, allowing for quick retrieval and comparison of text-based data. Traditional databases work well for exact queries but fall short when it comes to understanding the nuances of human language. Enter Vector Databases, a game-changer in handling semantic search. Unlike traditional text matching, which relies on exact words or phrases, vector databases like Postgres with pgvector process information semantically. This database is a cornerstone of the system's ability to match user queries with the most relevant information from the scraped content, enabling fast and accurate responses.

5. User Ask Question

The system provides an API through which users can submit their questions. In this use case, users can ask any question related to the content of the Open5GS documentation. This API serves as the primary interface for interactions between the user and the chatbot. The API takes a parameter, `user_id`, which is used to identify different user sessions. This `user_id` is used for demonstration purposes. In real-world scenarios, it could be managed with an Authorization header (e.g., JWT Bearer token) in the HTTP request. The API is designed to be intuitive and accessible, enabling users to easily input their queries and receive responses.

6. Create Vector Embedding of Question

When a user submits a question through the API, the system converts this question into a vector embedding. The generation of the embedding is automatically handled by the `ConversationalRetrievalChain`. This facilitates the semantic search of documents related to the question within the vector database.

7. Semantic Search Vector Database

Once the vector embedding for the question is created, the system employs semantic search to scan through the vector database, identifying content most relevant to the user's query. By comparing the vector embedding of the question with those of the stored data, the system can accurately pinpoint information that is contextually similar or related to the query. In this scenario, I have utilized the `ConversationalRetrievalChain`, which automatically handles semantic searches based on the input query. The results of the semantic search are then identified as `context` for the LLM.

8. Generate Prompt

Next, the `ConversationalRetrievalChain` generates a `custom_prompt` with the user's question and the semantic search result (context). A prompt for a

language model is a set of instructions or input provided by the user to guide the model's response. This helps the model understand the context and generate relevant and coherent language-based outputs, such as answering questions, completing sentences, or engaging in a conversation.

9. Post Prompt to LLM

After generating the prompt, it is posted to the LLM (in our case, the `Llama2 7B`) through Langchain libraries `Ollama` (Langchain officially supports the `Ollama` with in `langchain_community.llms`). The LLM then finds the answer to the question based on the provided context. The `ConversationalRetrievalChain` handles this function of posting the query to the LLM (behind the scenes, it uses Ollama's REST APIs to submit the question).

10. LLM Generate Answer

The LLM, utilizing the advanced capabilities of Meta's Llama-2, processes the question within the context of the provided content. It then generates a response and sends it back.

11. Save Query and Response in MongoDB Chat History

Langchain provides a variety of components for managing conversational memory. In this chatbot, `MongoDB` has been employed for the management of conversational memory. At this stage, both the user's question and the chatbot's response are recorded in MongoDB storage as part of the chat history. This approach ensures that all user chat histories are persistently stored in MongoDB, thus enabling the retrieval of previous interactions. The data is stored in MongoDB on a per-user-session basis. To distinguish between user sessions, the API utilizes the `user_id` parameter, as previously mentioned. This historical data is pivotal in shaping future interactions. When the same user poses subsequent questions, the chat history, along

with the new semantic search results (`context`), is relayed to the LLM. This process guarantees that the chatbot can maintain context throughout a conversation, resulting in more precise and tailored responses.

12. Send Answer Back to User

Finally, the answer received from the LLM is forwarded to the user via the HTTP API. Users can continue to ask different questions in subsequent requests by providing the same `user_id`. The system then recognizes the user's chat history and includes it in the information sent to the LLM, along with the new semantic search results. This process ensures a seamless and contextually aware conversation, enriching the user experience with each interaction.

Implementation

The complete implementation of this ChatBot is detailed below. The full source code of the ChatBot agent is available for access and review on GitLab.

1. Configurations

In the `config.py` file, I have defined various configurations used in the ChatBot. These configurations are read through environment variables in adherence to the principles of 12-factor apps.

```
import os

# define init index
INIT_INDEX = os.getenv('INIT_INDEX', 'false').lower() == 'true'

# vector index persist directory
INDEX_PERSIST_DIRECTORY = os.getenv('INDEX_PERSIST_DIRECTORY', "./data/chromadb"

# target url to scrape
```

```
TARGET_URL =  os.getenv('TARGET_URL', "https://open5gs.org/open5gs/docs/")

# http api port
HTTP_PORT = os.getenv('HTTP_PORT', 7654)

# mongodb config host, username, password
MONGO_HOST = os.getenv('MONGO_HOST', 'localhost')
MONGO_PORT = os.getenv('MONGO_PORT', 27017)
MONGO_USER = os.getenv('MONGO_USER', 'testuser')
MONGO_PASS = os.getenv('MONGO_PASS', 'testpass')
```

2. HTTP API

The HTTP API implementation is carried out in `api.py`. This API includes an HTTP POST endpoint `api/question`, which accepts a JSON object containing a `question` and `user_id`. The `user_id` is utilized for demonstration purposes. In a real-world application, this could be managed with an `Authorization header` (e.g., `JWT Bearer token`) in the HTTP request. When a question request is received from the user, it is forwarded to the `chat` function in the ChatBot model.

```
from flask import Flask
from flask import jsonify
from flask import request
from flask_cors import CORS
import logging
import sys
from model import init_index
from model import init_conversation
from model import chat
from config import *

app = Flask(__name__)
CORS(app)

logging.basicConfig(stream=sys.stdout, level=logging.INFO, format='%(asctime)s -

@app.route('/api/question', methods=['POST'])
def post_question():
    json = request.get_json(silent=True)
```

```
    question = json['question']
    user_id = json['user_id']
    logging.info("post question `%s` for user `%s`", question, user_id)

    resp = chat(question, user_id)
    data = {'answer':resp}

    return jsonify(data), 200

if __name__ == '__main__':
    init_index()
    init_conversation()
    app.run(host='0.0.0.0', port=HTTP_PORT, debug=True)
```

3. Model

Below is the implementation of the Model. It includes a function, `init_index`, which scrapes data from a given web URL and creates the vector store. An environment variable, `INIT_INDEX`, is used to determine whether to create the index. The `init_conversation` function initializes the `ConversationalRetrievalChain`, with Ollama's `Llama2` LLM which available through the Ollama's model REST API `<host>:11434` (Ollama provides a REST API for interacting with the LLMs. For detailed instructions and more information on how to use this feature, refer to the `Run Ollama Llama2` section). The `chat` function is responsible for posting questions to the LLM.

```
from langchain_community.llms import Ollama
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import DirectoryLoader
from langchain.document_loaders.recursive_url_loader import RecursiveUrlLoader
from langchain.embeddings import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import Chroma
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.llms import OpenAI
```

```
from langchain.embeddings import HuggingFaceEmbeddings
from bs4 import BeautifulSoup as Soup
from langchain.utils.html import (PREFIXES_TO_IGNORE_REGEX,
                                  SUFFIXES_TO_IGNORE_REGEX)

from config import *
import logging
import sys

logging.basicConfig(stream=sys.stdout, level=logging.INFO, format='%(asctime)s -


global conversation
conversation = None


def init_index():
    if not INIT_INDEX:
        logging.info("continue without initializing index")
        return

    # scrape data from web
    documents = RecursiveUrlLoader(
        TARGET_URL,
        max_depth=4,
        extractor=lambda x: Soup(x, "html.parser").text,
        prevent_outside=True,
        use_async=True,
        timeout=600,
        check_response_status=True,
        # drop trailing / to avoid duplicate pages.
        link_regex=(
            f"href=[\"']{PREFIXES_TO_IGNORE_REGEX}((?:{SUFFIXES_TO_IGNORE_REGEX}
            r"(?:[\#'\"]|\/[\#'\"])"
        ),
    ).load()

    logging.info("index creating with `%d` documents", len(documents))

    # split text
    # this chunk_size and chunk_overlap effects to the prompt size
    # execeed promt size causes error `prompt size exceeds the context window si
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overla
    documents = text_splitter.split_documents(documents)

    # create embeddings with huggingface embedding model `all-MiniLM-L6-v2`
    # then persist the vector index on vector db
    embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
    vectordb = Chroma.from_documents(
        documents=documents,
```

```
            embedding=embeddings,
            persist_directory=INDEX_PERSIST_DIRECTORY
        )
        vectordb.persist()


    def init_conversation():
        global conversation

        # load index
        embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
        vectordb = Chroma(persist_directory=INDEX_PERSIST_DIRECTORY,embedding_functi

        # llama2 llm which runs with ollama
        # ollama expose an api for the llam in `localhost:11434`
        llm = Ollama(
            model="llama2",
            base_url="http://localhost:11434",
            verbose=True,
        )

        # create conversation
        conversation = ConversationalRetrievalChain.from_llm(
            llm,
            retriever=vectordb.as_retriever(),
            return_source_documents=True,
            verbose=True,
        )


    def chat(question, user_id):
        global conversation

        chat_history = []
        response = conversation({"question": question, "chat_history": chat_history}
        answer = response['answer']

        logging.info("got response from llm - %s", answer)

        # TODO save history

        return answer
```

Run Application

Below are the main steps to operate the ChatBot application and interact with it. Questions can be submitted using the HTTP API, and responses will be received accordingly.

1. Install Dependencies

In this application, I have utilized a number of Python packages that need to be installed using Python's pip package manager before running the application. The `requirements.txt` file lists all the necessary packages.

```
huggingface-hub
sentence-transformers

Flask==2.0.1
Werkzeug==2.2.2
flask-cors

langchain==0.0.352
chromadb==0.3.29
tiktoken
unstructured
unstructured[local-pdf]
unstructured[local-inference]
```

I have used python virtual environment to setup these dependencies. These packages can be easily installed by executing the command `pip install -r requirements.txt`.

```
# create virtual environment in `ollama` source directory
❯❯ cd ollama
❯❯ python -m venv .venv

# enable virtual environment
❯❯ source .venv/bin/activate
```

```
# install dependencies
❯❯ pip install -r requirements.txt
```

2. Run Ollama Llama2

Ollama offers versatile deployment options, enabling it to run as a standalone binary on macOS, Linux, or Windows, as well as within a Docker container. This flexibility ensures that users can easily set up and interact with LLMs on their preferred platform. Ollama supports both command-line and REST API interactions, allowing for seamless integration into a variety of workflows and applications. An example of its utility is running the `Llama2` model through Ollama, demonstrating its capability to host and manage LLMs efficiently. Below is an illustrated method for deploying Ollama with Docker, highlighting my experience running the Llama2 model on this platform.

```
# run ollama with docker
# use directory called `data` in current working as the docker volume,
# all the data in the ollama(e.g downloaded llm images) will be available in tha
❯❯ docker run -d -v $(PWD)/data:/root/.ollama -p 11434:11434 --name ollama ollam

# connect to ollama container
❯❯ docker exec -it ollama bash

# run llama2 llm
# this will download the llm image and run it
# if llm image already exists it will start the llm image
root@150bc5106246:/# ollama run llama2
pulling manifest
pulling 8934d96d3f08... 100%
pulling 8c17c2ebb0ea... 100%
pulling 7c23fb36d801... 100%
pulling 2e0493f67d0c... 100%
pulling fa304d675061... 100%
pulling 42ba7f8a01dd... 100%
verifying sha256 digest
writing manifest
removing any unused layers
```

```
success
>>>

# exit from llm console
>>> /bye
root@c96f4fc1be6f:/#

# list running llms
root@150bc5106246:/# ollama list
NAME          ID           SIZE   MODIFIED
llama2:latest 78e26419b446 3.8 GB 10 hours ago

# reconnect to the llm console
root@c96f4fc1be6f:/# ollama run llama2
>>>

# ask question via llm console
root@c96f4fc1be6f:/# ollama run llama2
>>> what is docker
Docker is an open-source platform that enables you to create, deploy, and run ap
environments without worrying about compatibility issues. Docker provides a cons
```

```
# ollama exposes REST API(`api/generate`) to the llm which runs on port `11434`
# we can ask question via the REST API(e.g using `curl`)
# ask question and get answer as streams
# `"stream": true` will streams the output of llm(e.g send word by word as strea
❯❯ curl http://localhost:11434/api/generate -d '{
  "model": "llama2",
  "prompt": "what is docker?",
  "stream": true
}'
{"model":"llama2","created_at":"2024-03-17T10:41:53.358162047Z","response":"\n",
{"model":"llama2","created_at":"2024-03-17T10:41:53.494021698Z","response":"D","
{"model":"llama2","created_at":"2024-03-17T10:41:53.630381369Z","response":"ocke
{"model":"llama2","created_at":"2024-03-17T10:41:53.766590368Z","response":" is"
{"model":"llama2","created_at":"2024-03-17T10:41:53.902649027Z","response":" an"
{"model":"llama2","created_at":"2024-03-17T10:41:54.039338585Z","response":" ope
{"model":"llama2","created_at":"2024-03-17T10:41:54.175494123Z","response":"-","
{"model":"llama2","created_at":"2024-03-17T10:41:54.311130558Z","response":"sour
{"model":"llama2","created_at":"2024-03-17T10:41:54.447809241Z","response":" pla
{"model":"llama2","created_at":"2024-03-17T10:41:54.585971524Z","response":" tha
{"model":"llama2","created_at":"2024-03-17T10:41:54.723769251Z","response":" ena
{"model":"llama2","created_at":"2024-03-17T10:41:54.862244297Z","response":" you
{"model":"llama2","created_at":"2024-03-17T10:41:54.999796889Z","response":" to"
```

```
{"model":"llama2","created_at":"2024-03-17T10:41:55.136406278Z","response":" cre
{"model":"llama2","created_at":"2024-03-17T10:41:55.273430683Z","response":",","
{"model":"llama2","created_at":"2024-03-17T10:41:55.411326998Z","response":" dep
{"model":"llama2","created_at":"2024-03-17T10:41:55.54792922Z","response":",","d
{"model":"llama2","created_at":"2024-03-17T10:41:55.68550623Z","response":" and"

# ask question and get answer without stream
# that will wail till getting full response from llm and output
❯❯ curl http://localhost:11434/api/generate -d '{
  "model": "phi",
  "prompt": "Why is docker?",
  "stream": false
}'
{"model":"phi","created_at":"2024-03-16T23:42:34.140800795Z","response":" Docker
```

3. Run RAG Application

The RAG application can be initiated through `api.py` as outlined below. Prior to running it, it's necessary to set a few configurations via environment variables. Once `api.py` is executed, it will start the HTTP API, enabling users to post their questions.

```
# enable virtual environment in `ollama` source directory
❯❯ cd ollama
❯❯ source .venv/bin/activate

# set env variabl INIT_INDEX which determines weather needs to create the index
❯❯ export INIT_INDEX=true

# run aplication
❯❯ python api.py
2024-03-16 20:54:05,715 - INFO - index creating with `18` documents
2024-03-16 20:54:06,682 - INFO - Load pretrained SentenceTransformer: all-MiniLM
2024-03-16 20:54:09,036 - INFO - Use pytorch device_name: mps
2024-03-16 20:54:09,373 - INFO - Anonymized telemetry enabled. See https://docs.
2024-03-16 20:54:09,467 - INFO - loaded in 361 embeddings
2024-03-16 20:54:09,468 - INFO - loaded in 1 collections
2024-03-16 20:54:09,469 - INFO - collection with name langchain already exists,
2024-03-16 20:54:11,177 - INFO - Persisting DB to disk, putting it in the save f
2024-03-16 20:54:11,197 - INFO - Load pretrained SentenceTransformer: all-MiniLM
2024-03-16 20:54:12,418 - INFO - Use pytorch device_name: mps
2024-03-16 20:54:12,476 - INFO - Anonymized telemetry enabled. See https://docs.
```

```
2024-03-16 20:54:12,490 - INFO - loaded in 722 embeddings
2024-03-16 20:54:12,490 - INFO - loaded in 1 collections
2024-03-16 20:54:12,491 - INFO - collection with name langchain already exists,
 * Serving Flask app 'api' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
   Use a production WSGI server instead.
 * Debug mode: on
2024-03-16 20:54:12,496 - INFO - WARNING: This is a development server. Do not u
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:7654
 * Running on http://192.168.0.110:7654
```

4. Post Question

Once the RAG application is running, I can submit questions related to the Open5GS documentation via the HTTP API.

```
# post question
❯❯ curl -i -XPOST "http://localhost:7654/api/question" \
--header "Content-Type: application/json" \
--data '
{
  "question": "what is open5gs",
  "user_id": "kakka"
}
'


# ConversationalRetrievalChain generate following prompt with question, semantic
> Entering new LLMChain chain...
Prompt after formatting:
Use the following pieces of context to answer the question at the end. If you do

Open5GS     Sukchan Lee  acetcom@gmail.com        GitHub  open5gs      Open5GS i

Open5GS     Sukchan Lee  acetcom@gmail.com        GitHub  open5gs      Open5GS i

Question: what is open5gs
Helpful Answer:

> Finished chain.
```

```
> Finished chain.
2024-03-16 20:55:05,843 - INFO - got response from llm - Based on the provided c


# response
{
  "answer": "Based on the provided context, Open5GS appears to be an open-source
}



---



# post next question
❯❯ curl -i -XPOST "http://localhost:7654/api/question" \
--header "Content-Type: application/json" \
--data '
{
  "question": "what is EPC",
  "user_id": "kakka"
}
'


# ConversationalRetrievalChain generate following prompt with question, semantic
> Entering new LLMChain chain...
Prompt after formatting:
Use the following pieces of context to answer the question at the end. If you do

Open5GS     Sukchan Lee  acetcom@gmail.com        GitHub  open5gs      Open5GS i

Open5GS     Sukchan Lee  acetcom@gmail.com        GitHub  open5gs      Open5GS i

Question: what is EPC
Helpful Answer:

> Finished chain.
2024-03-16 20:56:24,053 - INFO - got response from llm - EPC stands for "Evolved


# response
{
  "answer": "EPC stands for \"Evolved Packet Core.\" It is a component of the 5G
}
```

What's Next

In next post, I have discussed building the same RAG application using a different tool called `LlamaIndex` which is a comprehensive framework designed for constructing production-level RAG applications. Build RAG Application Using a LLM Running on Local Computer with Ollama and LlamaIndex.

Reference

1. https://medium.com/rahasak/build-rag-application-using-a-llm-running-on-local-computer-with-gpt4all-and-langchain-13b4b8851db8
2. https://medium.com/@abonia/ollama-and-langchain-run-llms-locally-900931914a46
3. https://cheshirecat.ai/local-models-with-ollama/
4. https://fossengineer.com/selfhosting-llms-ollama/
5. https://abvijaykumar.medium.com/ollama-build-a-chatbot-with-langchain-ollama-deploy-on-docker-5dfcfd140363
6. https://sgoel.dev/posts/langchain-applications-using-ollama/
7. https://medium.com/@vndee.huynh/build-your-own-rag-and-run-it-locally-langchain-ollama-streamlit-181d42805895
8. https://mlabonne.github.io/blog/posts/Quantize_Llama_2_models_using_ggml.html
9. https://medium.com/@ingridwickstevens/quantization-of-llms-with-llama-cpp-9bbf59deda35
10. https://towardsdatascience.com/quantize-llama-models-with-ggml-and-llama-cpp-3612dfbcc172

Llm Ollama AI Langchain Gpt

Written by (λx.x)eranga

2.4K Followers · Editor for Effectz.AI

Follow

Ego = 1/Knowledge

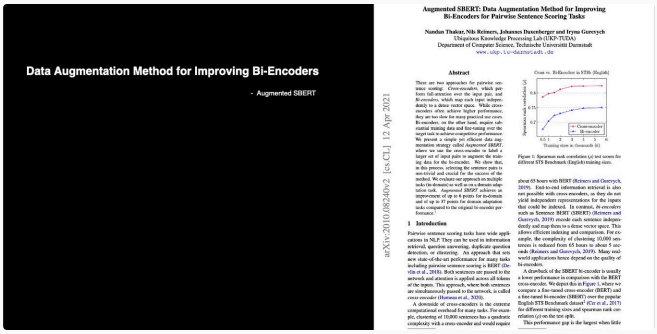
More from (λx.x)eranga and Effectz.AI

(λx.x)eranga in Effectz.AI

Build RAG Application Using a LLM Running on Local Computer with...

Privacy-preserving LLM without GPU

17 min read · Mar 24, 2024



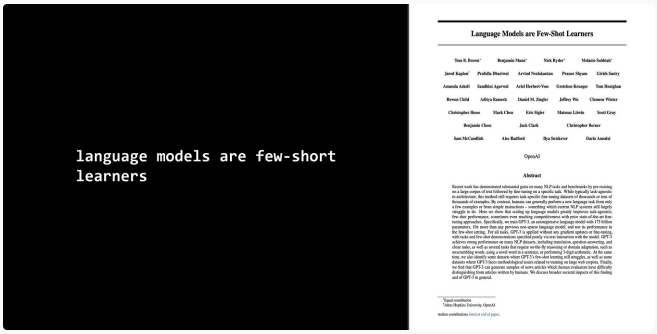
Deltaaruna in Effectz.AI

Optimizing RAG, Fine-tuning Embedding and Reranking model...

1. Introduction

13 min read · Apr 16, 2024

539 3



(λx.x)eranga in Effectz.AI

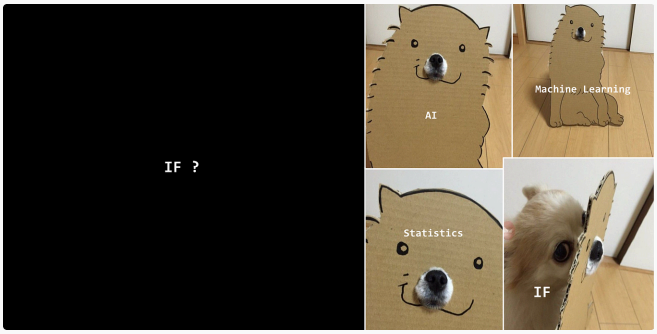
Creating Custom ChatGPT with Your Own Dataset using OpenAI...

Happy LLM

10 min read · Aug 22, 2023

129 2

142



(λx.x)eranga in Effectz.AI

Build RAG Application Using a LLM Running on Local Computer with...

Privacy-preserving LLM without GPU

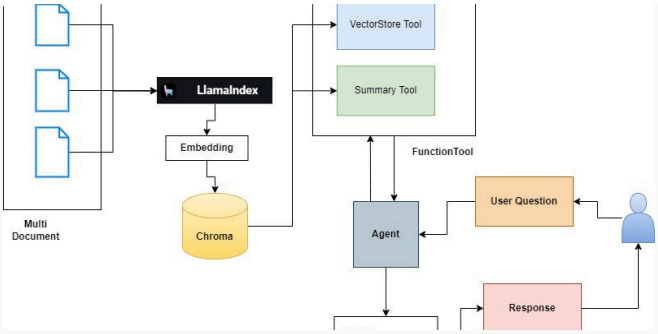
21 min read · Mar 10, 2024

561 6

See all from (λx.x)eranga

See all from Effectz.AI

Recommended from Medium



Duy Huynh

Build your own RAG and run it locally: Langchain + Ollama +...

With the rise of Large Language Models and its impressive capabilities, many fancy...

6 min read · Dec 5, 2023

758 7

Plaban Nayak in The AI Forum

Multi-document Agentic RAG using Llama-Index and Mistral

Introduction

15 min read · May 12, 2024

259 4

Lists

Natural Language Processing

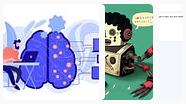
1464 stories · 975 saves

Generative AI Recommended Reading

52 stories · 1063 saves

What is ChatGPT?

9 stories · 357 saves



The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 381 saves

(λx.x)eranga in Effectz.AI

Build RAG Application Using a LLM Running on Local Computer with...

Privacy-preserving LLM without GPU

17 min read · Mar 24, 2024

539 3

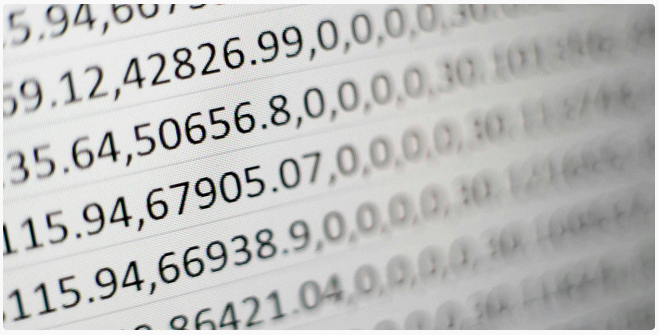
Lars Wiik

GPT-4o vs. GPT-4 vs. Gemini 1.5 ⭐ —Performance Analysis

Measuring English Language Understanding of OpenAI's New Flagship Model

5 min read · May 13, 2024

1.3K 24



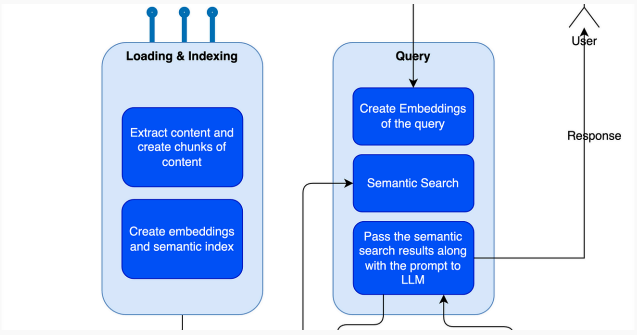
Intel in Intel Tech

Tabular Data, RAG, & LLMs: Improve Results Through Data...

How to ingest small tabular data when working with LLMs.

10 min read · May 14, 2024

93 1



A B Vijay Kumar

Retrieval Augmented Generation(RAG) — Chatbot for...

Implement the RAG technique using Langchain, and LlamaIndex for conversation...

7 min read · Feb 6, 2024

341 7

See more recommendations