Project Workbook: Local Deployment of a Retrieval-Augmented Generation (RAG) System on Windows

---

Welcome to Your RAG Project!

In this project, you will build a fully local **Retrieval-Augmented Generation (RAG) system** on your Windows machine. This system allows you to upload, search, and retrieve insights from your meeting transcripts. By leveraging **Weaviate** (a vector database) and **Hugging Face models** (for text generation), you will create a powerful tool to ask questions about past meetings, action items, and topics discussed.

By the end of this project, you will:

- Understand how to **retrieve** and **index** documents using a vector database.
- Run **Hugging Face models** locally to generate responses based on document retrieval.
- Use **vector embeddings** to enable semantic search on your meeting transcripts.

Who is This For?

This workbook is designed for **absolute beginners**. Each step is explained in detail, and we'll guide you through every task, assuming no prior knowledge of machine learning, Python, or Docker.

What You Will Learn:

- How to **set up** your environment for running Weaviate and Hugging Face models locally.
- How to **upload** and **index** your meeting transcripts into a vector database.
- How to use **local models** to generate responses to questions based on your meeting transcripts.
- How to **chunk and vectorize large transcripts** to ensure they are usable in the RAG system.

Modules Overview

1. **Module 1: Setting Up Your Environment**
   Learn how to install the tools and set up your local environment.
2. **Module 2: Installing and Running Weaviate Locally**
   Install and configure Weaviate for storing and retrieving documents.

3. **Module 3: Defining the Schema and Uploading Meeting Transcripts**
   Learn how to structure and upload your meeting transcripts.
4. **Module 4: Vectorizing Your Transcripts Locally**
   Convert transcripts into vector embeddings for fast and efficient retrieval.
5. **Module 5: Installing and Using Hugging Face Models**
   Use Hugging Face models locally to generate responses based on document retrieval.
6. **Module 6: Retrieval and Generation Process**
   Query your transcripts using Weaviate and generate responses using Hugging Face models.
7. **Module 7: Chunking and Vectorizing Large Transcripts**
   Learn how to split large transcripts into smaller chunks and vectorize them for effective storage and retrieval.

Module 1: Setting Up Your Environment

Objective:

Set up the local environment by installing **Python**, **Docker**, and **Git**. Create a folder for your project and prepare everything to run Weaviate and Hugging Face models.

Steps:

1. **Install Python**:
   - Go to python.org/downloads.
   - Download the latest Python 3.x version for Windows.
   - **Important**: During installation, check the box to "Add Python to PATH."

Verify the installation:
bash
Copy code
```
python --version
```

   -
2. **Install Docker**:
   - Download **Docker Desktop** from docker.com/products/docker-desktop.
   - Follow the installation instructions.

Check if Docker is working by running:
bash
Copy code
```
docker --version
```

   -
3. **Install Git**:
   - Download Git from git-scm.com.
4. **Create a Project Folder**:

- **Create a folder** on your preferred drive (e.g., `A:\MyRAGSystem`).

Open **Command Prompt** or **PowerShell**, and navigate to the folder:
bash
Copy code
```
cd A:\MyRAGSystem
```

5. **Create a Virtual Environment**:

In your project folder, create a Python virtual environment:
bash
Copy code
```
python -m venv my_rag_env
```

Activate it:
bash
Copy code
```
.\my_rag_env\Scripts\activate
```

6. **Install Required Libraries**:

Install Weaviate and Hugging Face libraries:
bash
Copy code
```
pip install weaviate-client sentence-transformers transformers
```

Module 2: Installing and Running Weaviate Locally

Objective:

Install Weaviate to store and retrieve documents.

Steps:

1. **Set Up Weaviate Using Docker**:

Create a Docker Compose file in your project folder (`docker-compose.yml`):
yaml

Copy code
```
version: '3'
services:
  weaviate:
    image: semitechnologies/weaviate:latest
    ports:
      - "8080:8080"
    environment:
      QUERY_DEFAULTS_LIMIT: 100
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
```

- ○

Run Weaviate:
bash
Copy code
```
docker-compose up -d
```

2.
3. **Verify Installation**:
   - Open a browser and go to:
     `http://localhost:8080/v1/schema`

Module 3: Defining the Schema and Uploading Meeting Transcripts

Objective:

Define the structure for storing meeting transcripts and upload them into **Weaviate**.

Steps:

Define Your Schema:
python
Copy code
```
import weaviate

client = weaviate.Client("http://localhost:8080")

schema = {
    "classes": [
        {
            "class": "Transcript",
```

```
            "properties": [
                {"name": "date", "dataType": ["date"]},
                {"name": "content", "dataType": ["text"]}
            ]
        }
    ]
}
client.schema.create(schema)
```

1.

Upload Your Transcripts:
python
Copy code

```
transcript = {
    "date": "2024-01-15",
    "content": "We discussed project milestones and next steps."
}
client.data_object.create(transcript, "Transcript")
```

2.

Module 4: Vectorizing Your Transcripts Locally

Objective:

Generate vector embeddings for your meeting transcripts to enable efficient retrieval in Weaviate.

Steps:

Install Sentence Transformers:
bash
Copy code

```
pip install sentence-transformers
```

1.

Generate Vector Embeddings:
python
Copy code

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
content = "Meeting transcript content to be vectorized."
```

```
vector = model.encode(content)
```

2.

Store the Vectors in Weaviate:
python
Copy code

```
client.data_object.create(
    {"date": "2024-01-15", "content": content},
    "Transcript", vector=vector
)
```

3.

Module 5: Installing and Using Hugging Face Models for Document Retrieval

Objective:

Use Hugging Face models locally to generate responses based on the stored meeting transcripts.

Steps:

Install Transformers Library:
bash
Copy code

```
pip install transformers
```

1.

Generate Responses Using Hugging Face Models:
python
Copy code

```
from transformers import pipeline

generator = pipeline("text-generation", model="EleutherAI/gpt-neo-125M")
prompt = "What milestones were discussed during the January 15 meeting?"
response = generator(prompt, max_length=100)
print(response[0]["generated_text"])
```

2.

Module 6: Retrieval and Generation Process

Objective:

Retrieve a transcript from Weaviate using semantic search and generate a response using Hugging Face.

Steps:

python
Copy code

```
import weaviate
from transformers import pipeline
from sentence_transformers import SentenceTransformer

client = weaviate.Client("http://localhost:8080")

query = "What milestones were discussed during the January 15 meeting?"
model = SentenceTransformer('all-MiniLM-L6-v2')
query_vector = model.encode(query)

results = client.query.get("Transcript", ["content",
"date"]).with_near_vector({
    "vector": query_vector,
}).do()

if results['data']['Get']['Transcript']:
    transcript_content = results['data']['Get']['Transcript'][0]['content']
    print(f"Retrieved Transcript: {transcript_content}")
else:
    print("No relevant transcript found.")
    transcript_content = "No relevant content found."

generator = pipeline("text-generation", model="EleutherAI/gpt-neo-125M")
prompt = f"Using the transcript content: {transcript_content}, what milestones
were discussed?"
response = generator(prompt, max_length=100)
print(f"Generated Response: {response[0]['generated_text']}")
```

Module 7: Chunking and Vectorizing Large Transcripts

Objective:

Learn how to split large meeting transcripts into smaller chunks and vectorize them for effective storage and retrieval.

Why Chunking is Necessary:

Large transcripts (e.g., >512 tokens) are difficult to process efficiently. Chunking splits large textIt looks like my response got cut off, but here is the continuation of **Module 7**, which focuses on **chunking large transcripts** and vectorizing them:

Module 7: Chunking and Vectorizing Large Transcripts

Objective:

Learn how to split large meeting transcripts into smaller chunks and vectorize them for effective storage and retrieval.

Why Chunking is Necessary:

Large transcripts (e.g., more than 512 tokens) are difficult to process efficiently by models, and certain models have token limits. Chunking the text into smaller parts ensures that each part of the document is small enough to be processed effectively and vectorized for semantic search.

Steps for Chunking and Vectorizing Large Transcripts:

1. **Define a Chunking Function**:
    - You can define a function to split large transcripts into smaller chunks based on a token limit (e.g., 512 tokens). Here's a simple Python function that chunks text based on a word limit:

python
Copy code

```
def chunk_text(text, chunk_size=512):
    words = text.split()
    for i in range(0, len(words), chunk_size):
        yield ' '.join(words[i:i+chunk_size])
```

2.
    - This function takes a transcript and splits it into chunks of 512 words (you can adjust the size as needed).
3. **Chunk the Transcript**:
    - Use the `chunk_text` function to chunk large transcripts:

python
Copy code
```
content = "Large meeting transcript content to be chunked and vectorized..."
chunks = list(chunk_text(content, chunk_size=512))
```

4. Each chunk will contain part of the transcript, ready to be processed.
5. **Vectorize and Store Each Chunk**:
    - For each chunk, generate vector embeddings and store them in Weaviate:

python
Copy code
```
model = SentenceTransformer('all-MiniLM-L6-v2')

for i, chunk in enumerate(chunks):
    vector = model.encode(chunk)
    transcript_chunk = {
        "date": "2024-01-15",  # Same date for all chunks of this transcript
        "content": chunk,
        "chunk_id": i  # Keep track of the chunk order
    }
    client.data_object.create(transcript_chunk, "Transcript", vector=vector)
```

6.
    - Each chunk is stored in Weaviate with its own vector, allowing for efficient semantic search.
7. **Querying with Chunks**:
    - When querying Weaviate, the similarity search will return relevant chunks based on the query. You can retrieve the closest chunks and reconstruct the relevant sections of the original transcript.

Key Takeaways for Chunking:

- **Efficiency**: Chunking ensures that large documents are processed within the token limits of models and enables efficient retrieval.
- **Context Preservation**: When chunking, ensure that each chunk has enough context (e.g., complete sentences) to remain meaningful.
- **Storing in Order**: Include a `chunk_id` to preserve the order of the chunks if you need to reassemble them during retrieval.

Conclusion

By following this guide, you have built a complete **Retrieval-Augmented Generation (RAG) system** on your local machine. This system allows you to upload, index, and retrieve meeting transcripts, and generate responses based on semantic search and text generation using local models.

Key accomplishments:

- **Set up the environment** with Python, Docker, and Weaviate.
- **Uploaded and vectorized transcripts** for fast retrieval.
- **Generated responses** using Hugging Face models.
- **Chunked large transcripts** for efficient processing and retrieval.

Feel free to extend this project by adding more transcripts, optimizing the model, or exploring new use cases.

// ISAAC v1.3 (2024-09-15) by ADEPTUS - Contact: ken.elwell@gmail.com