Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

CSEN 903: Advanced Computer Lab, Winter 2025
Lab 8 Manual: Retrieval Augmented Generation Using LangChain

## Part 1: Read and Watch Before the Lab

This lab introduces you to the exciting world of **Generative AI** and the critical role of prompt engineering in controlling generative models' outputs. You will also explore **Retrieval-Augmented Generation (RAG)**, an emerging technique that improves the performance and accuracy of language models by grounding responses in external knowledge. Using **LangChain**, you'll build a document-aware Q&A system that combines a **retriever** (search engine) with a **generative model** (like GPT-3.5) inside a Jupyter notebook.

You'll:

- Understand what prompt engineering is and why it matters.
- Understand how RAG improves LLM responses
- Use LangChain to index and query local documents
- Build a retriever+LLM pipeline for document-based question answering
- Experiment with queries to observe grounded vs hallucinated responses
- Create a mini-app that generates outputs based on user input, using streamlit

### Prompt Engineering Essentials

Prompt engineering is the art of crafting inputs to generative AI models to obtain high-quality, specific, and useful outputs. Whether you're working with large language models or image generators, the way you ask matters.

#### *Best Practices*

1. Be Specific: Avoid vague or generic instructions.

   Example: `"Write a 3-bullet summary of this article for high school students."`

2. Use Few-Shot Examples: Provide examples of the desired output.

Example: `"Q: 5+3?\nA: 8\nQ: 9+1?\nA:"`

3. Set the role: Assign a persona to the model.

   Example: `"You are a helpful programming tutor. Explain recursion."`

4. Add constraints: Control format, length, and style.

   Example: `"Respond in less than 100 words, use bullet points.`

*More Examples of Bad and Good Prompts*

| Use Case | ❌ Bad Prompt | ✅ Good Prompt |
|---|---|---|
| Text | `"Tell me a story"` | `"Write a story about a robot learning to cook Egyptian food in 3 paragraphs."` |
| Code | `"Give code"` | `"Write a Python function that returns the factorial of a number using recursion."` |
| Image | `"Draw a tree"` | `"A detailed digital painting of a cherry blossom tree under moonlight"` |

**Required Readings:**

1. Prompt Engineering Guide
2. What is Retrieval-Augmented Generation(RAG) in LLM and How it works?

**Required Watch List:**

1. What is Retrieval Augmented Generation?
2. Building RAG Using LangChain Python Project

## Part 2: In-Lab Task

We will implement three different retrievers to answer a simple query. One retriever will only use similarity search, the second retriever will use Generative LLM, and the final retriever will use RAG. Our External content source for RAG will be the MS1 Checklist available on the CMS (You need to download it and add it to the Kaggle notebook using 'Add Input'). Our query will be "What should be included in report Milestone 1?"
You will find the notebook ready on Kaggle.

1. Install required libraries.
    a. Install pypdf to read your PDF
    b. Install sentence-transformers for embeddings
    c. Install faiss-cpu for vector search
    d. Install huggingface-hub to call the free HF Inference API

```
!pip install pypdf sentence-transformers faiss-cpu huggingface_hub --quiet
```

2. Import required libraries

```
from pypdf import PdfReader
from sentence_transformers import SentenceTransformer
import faiss
from huggingface_hub import InferenceClient

import numpy as np
import textwrap
```

3. Load a PDF or text file.

```
pdf_path = "/kaggle/input/rag-lab8/Checklist.pdf"

reader = PdfReader(pdf_path)
all_text = ""

for page in reader.pages:
    text = page.extract_text()
    if text:
        all_text += text + "\n"

# Preview the start of the text
print("Extracted text preview:\n")
print(all_text[:500])
```

4. Split the text into chunks.

```
def chunk_text(text, max_length=400, overlap=100):
    chunks = []
    current = []

    for line in text.split("\n"):
        line = line.strip()
        if not line:
            continue

        # If adding the new line exceeds max_length → finalize chunk
        if sum(len(x) for x in current) + len(line) > max_length:
            chunk = " ".join(current)
            chunks.append(chunk)

            # create overlap
           # Convert chunk into characters and take the last `overlap` chars
            if overlap > 0:
                overlap_text = chunk[-overlap:]
                current = [overlap_text, line]  # start new chunk with
overlap + new line
            else:
                current = [line]

        else:
            current.append(line)

    # Add the final chunk
```

```
    if current:
        chunks.append(" ".join(current))

    return chunks


chunks = chunk_text(all_text)
print(f"Total chunks created: {len(chunks)}")
print("Sample chunk:\n", chunks[0][:300], "...")
```

5. Load the embedding model and encode the created chunks.
   Text embedding is the process of converting text (words, sentences, or documents) into numerical vectors. You have been introduced to this concept in Lab 5 using TFIDF vectorization. In this lab, we perform sentence embedding using the all-MiniLM-L6-v2 embedding model

   - A small, fast, high-quality sentence embedding model.
   - Created for semantic similarity & RAG.
   - It converts text → numerical vectors (embeddings).

```
#Load embedding model
embedder = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

#Encode chunks
print("Encoding chunks... this may take ~10–20 seconds.")
embeddings = embedder.encode(chunks, convert_to_numpy=True)

embeddings.shape
```

6. Build FAISS Index.
   FAISS (Facebook AI Similarity Search) is an open-source library developed by Meta AI for fast similarity search and efficient vector indexing. It stores high-dimensional vectors and searches for similar vectors

```
#Build FAISS Index
d = embeddings.shape[1]
index = faiss.IndexFlatL2(d)

# Add vectors to index
index.add(embeddings)

print("FAISS index built successfully!")
```

```
print("Number of vectors in the index:", index.ntotal)
```

Now, each chunk of text is stored as a vector in FAISS. FAISS can now search among them to find the **most similar chunks to a query**

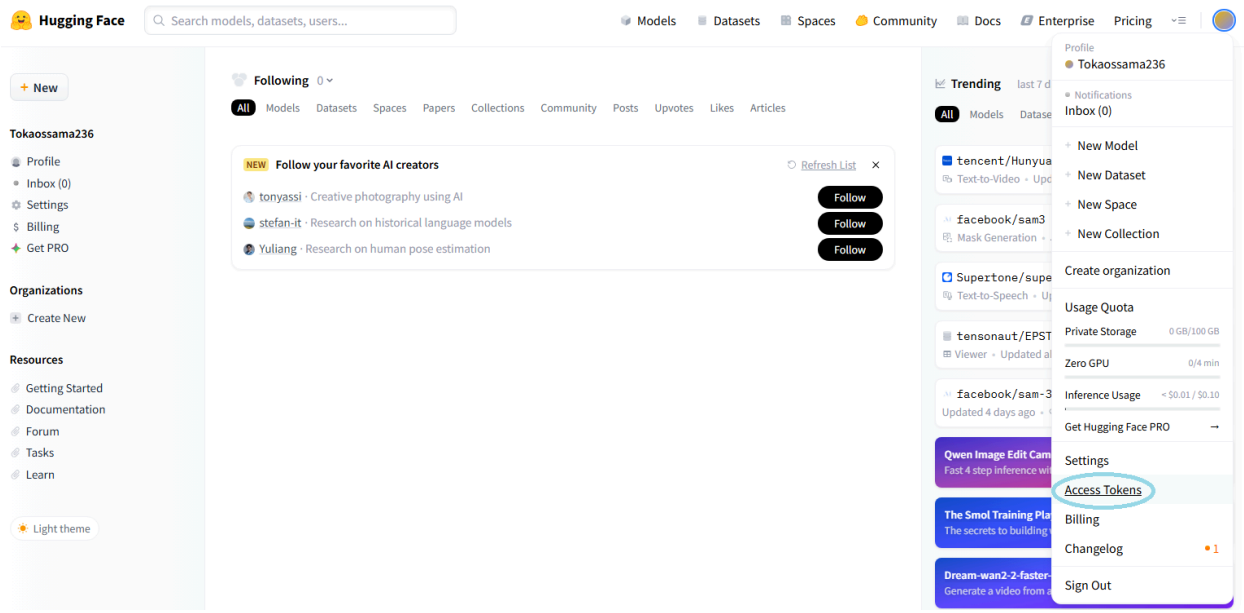
7. Our first retriever will try answering a sample query using only the FAISS index; i.e, using only distances between embeddings.

```
def retrieve(query, k=3):
    # Embed the user query using the same SentenceTransformer model used for
document chunks
    query_emb = embedder.encode([query], convert_to_numpy=True)

    # Search for the query using the FAISS index of our chunks
    distances, indices = index.search(query_emb, k)

    print(distances, indices)
    print("**Meaning: Best match is chunk", indices[0][0], "with distance",
distances[0][0], ",and Second best is chunk", indices[0][1], "with distance
", distances[0][1], "**")

    # Return the retrieved chunks
    results = [chunks[i] for i in indices[0]]
    return results

# Test retrieval
sample_query = "What is the main topic discussed in the document?"
retrieved_chunks = retrieve(sample_query, k=2)


print("Top retrieved chunks:\n")
for i, c in enumerate(retrieved_chunks, 1):
    print(f"---- Chunk {i} ----")
    print(c[:400], "\n")
```

Observing the returned answer, the FAISS retriever does not provide the most meaningful answer to the provided query.

We will move to creating our second retriever using a large language model (LLM) through an API from HuggingFace.

Before you proceed to the next step, you need to create a ‘Read Only‘ access token on Hugging Face.




Then add it to Kaggle using Add-ons → Secrets → Add Secret

8. Load your **secret HuggingFace token** from Kaggle

```
from kaggle_secrets import UserSecretsClient
from huggingface_hub import InferenceClient

user_secrets = UserSecretsClient()
HF_TOKEN = user_secrets.get_secret("RAG-Lab8") #"RAG-Lab8" is your Secret
Label


client = InferenceClient(
    model="google/gemma-2-2b-it",
    token=HF_TOKEN
)

def call_llm(prompt, max_tokens=256):
    response = client.chat_completion(
        messages=[{"role": "user", "content": prompt}],
        max_tokens=max_tokens,
        temperature=0.2, #temperature: controls randomness of the provided
answer
    )
    return response.choices[0].message["content"]
```

9. Try a variety of questions that can and cannot be answered from the document.

```
print(call_llm("Explain what a knowledge graph is in one sentence."))
```

```
print(call_llm("What should be included in report Milestone 1?"))
```

The answers of the “call_llm” function do NOT read or retrieve any information from the provided PDF or the FAISS index. It only sends a plain prompt to Gemma and returns the model’s reply. AGAIN, the second query returns an answer that DOES NOT make sense (a generic answer), because the provided prompt to the API includes only the query without any context from the document.

We will move to creating our third retriever using RAG, an AI framework that enhances a large language model’s (LLM) responses by first retrieving relevant information from an external knowledge source before generating an answer. Meaning, instead of just relying on what the LLm knows, we are adding a content store (a retrieval-relevant content) and combining it with the user query.

We need to build a context from the text chunks extracted from the document and use this context in the Gemma prompt, along with the query.

10. Now use RAG to answer

```
def rag_answer(query, k=3, max_tokens=500):
    # Step 1: Retrieve relevant chunks
    retrieved = retrieve(query, k=k)

    # Step 2: Build the context
    context = "\n\n".join(retrieved)

    # Step 3: Build prompt for the LLM
    prompt = f"""
You are a helpful assistant using Retrieval-Augmented Generation.

Here is relevant context extracted from a document:

{context}

Based on this context, answer the following question:
```

```
Question: {query}

If the answer is not in the context, say: "The document does not contain the
answer."
"""

    # Step 4: Call Gemma 2B Instruct (conversational mode)
    response = client.chat_completion(
        messages=[{"role": "user", "content": prompt}],
        max_tokens=max_tokens,
        temperature=0.2,
    )

    return response.choices[0].message["content"]
```

```
print(rag_answer("What should be included in report Milestone 1?"))
```

Now, the provided query answer finally makes sense.

Next, let's implement RAG using LangChain, which means building the same Retrieval-Augmented Generation pipeline we just created manually — **but using LangChain's high-level tools** instead of writing everything ourselves.

What is LangChain?

LangChain is a framework that provides:

- Ready-made **text splitters**
- Easy **embedding wrappers**
- Plug-and-play **VectorStores** (FAISS, Chroma, Pinecone…)
- Ready LLM clients (HuggingFace, OpenAI…)
- A built-in **retrieval pipeline**
- A built-in **RAG chain** that does everything for you

11. Install LangChain

```
!pip install langchain langchain-community langchain-huggingface --quiet
```

12. Import the needed libraries

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS as LCFAISS
```

```
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain.chains import RetrievalQA
from langchain_huggingface import ChatHuggingFace
from langchain.llms.base import LLM
from typing import Optional, List, Any
from pydantic import BaseModel, Field
```

13. Create a **custom LangChain LLM wrapper**

LangChan has plenty of built-in models ready to use. However, for the purpose of this lab, we want to use the same LLM used above. LangChain does not know how to use the HuggingFace `InferenceClient` by default. Hence, this wrapper creates a bridge so that LangChain can call the HuggingFace Inference API (Gemma-2B Instruct) as if it were a normal LangChain language model.

LangChain → wrapper → HF Inference API (Gemma)

```
# Custom LLM wrapper for HuggingFace Inference Client (Gemma conversational)
class GemmaLangChainWrapper(LLM):
    client: Any = Field(...)
    max_tokens: int = 500 #sets a default max output length

    @property
    def _llm_type(self) -> str:
        return "gemma_hf_api" #Identify the LLM type

    #what LangChain calls when it needs the LLM to answer something
    def _call(self, prompt: str, stop: Optional[List[str]] = None) -> str:
        response = self.client.chat_completion( #call the HuggingFace API
            messages=[{"role": "user", "content": prompt}],
            max_tokens=self.max_tokens,
            temperature=0.2,
        )
        return response.choices[0].message["content"]


# Instantiate the wrapper
gemma_llm = GemmaLangChainWrapper(client=client)
```

14. Chunking using LangChain

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=400,
    chunk_overlap=100
)

documents = splitter.create_documents([all_text])
len(documents)
```

15. Embedding Using LangChain

```
embedding_model = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)
```

16. Creating the FAISS retriever using LangChain

```
from langchain_community.vectorstores import FAISS as LCFAISS

vectorstore = LCFAISS.from_documents(
    documents=documents,
    embedding=embedding_model
)
retriever = vectorstore.as_retriever(search_kwargs={"k": 10})
```

17. Building the RAG QA using LangChain

```
qa_chain = RetrievalQA.from_chain_type(
    llm=gemma_llm,
    retriever=retriever,
    chain_type="stuff") #concatenate all retrieved documents and feed them
to the LLM as one big prompt.
```

18. Use the LangChain RAG to answer the query.

```
response = qa_chain.run("What should be included in report Milestone 1?")
print(response)
```

## Part 3: Graded Task

### Task Description

Transform your notebook-based retrieval system into a simple interactive chatbot using Streamlit. The chatbot should answer user questions grounded in a fixed knowledge source.

Your chatbot must:

- Load and embed a fixed document.
- Use **Chroma** or **FAISS** as the vector store.
- Use **LangChain's RetrievalQA chain** for querying.
- Be deployed in a **Streamlit app** interface with:
    - A text input for the user's question
    - A display of the generated answer
    - (Optional but encouraged) a display of the source chunks retrieved

### Deliverables:

Submit an .py file with the completed task above to the form. The deadline for submission is **@11:59 PM a week from your lab.** I.e., If your lab is on Saturday,29/11, you need to submit by Friday, 5/12 at 11:59 pm.