Data and Artificial Intelligence
Cyber Shujaa Program

Week 12 Assignment
Generative AI, RAG, Agentic AI

Student Name: Deborah Kwamboka Omae

Student ID: CS-DA02-25075

Table of Contents

Table of Figures

Introduction

In this assignment, I was to apply my understanding of Natural Language Processing, demonstrating key concepts covered in class.

Objectives

- Apply a pre-trained BERT model for sentence encoding using Hugging Face Transformers.
- Extract token-level contextual embeddings.
- Demonstrate how BERT captures word meaning based on context (polysemy).
- Compute and interpret cosine similarity between token embeddings.
- Explain the importance of contextual embeddings in contrast to static word vectors.

Tasks Completed

Step 1: Importing the Required Libraries

In this step, I imported all the essential libraries needed for building a Retrieval-Augmented Generation (RAG) pipeline, including LangChain, Sentence-Transformers, FAISS, and Transformers. I did this to ensure I had all the tools required to load documents, split them into chunks, generate embeddings, store vectors, and integrate an LLM for question answering.

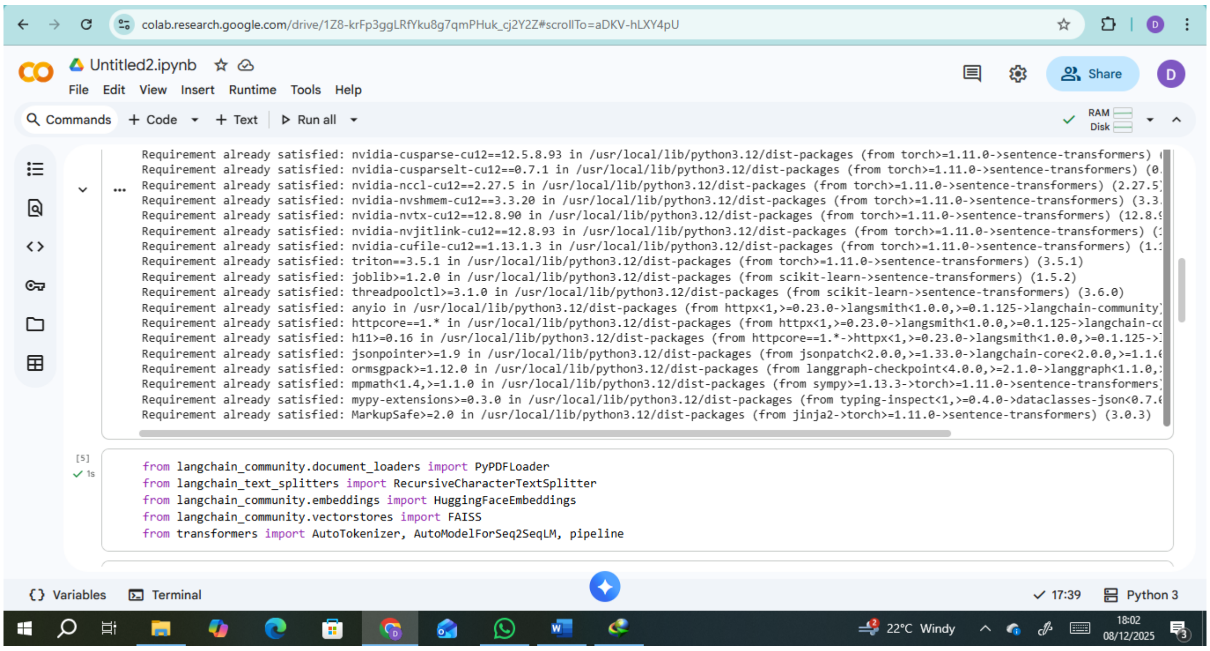
Code:

```
!pip install langchain langchain-community transformers sentence-transformers faiss-cpu pypdf

from langchain_community.document_loaders import PyPDFLoader

from langchain_text_splitters import RecursiveCharacterTextSplitter

from langchain_community.embeddings import HuggingFaceEmbeddings
```

from langchain_community.vectorstores import FAISS

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline

screenshot:



```
Requirement already satisfied: nvidia-cusparse-cu12==12.5.8.93 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers)
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (0.
Requirement already satisfied: nvidia-nccl-cu12==2.27.5 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (2.27.5
Requirement already satisfied: nvidia-nvshmem-cu12==3.3.20 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (3.3.
Requirement already satisfied: nvidia-nvtx-cu12==12.8.90 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (12.8.
Requirement already satisfied: nvidia-nvjitlink-cu12==12.8.93 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (
Requirement already satisfied: nvidia-cufile-cu12==1.13.1.3 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (1.
Requirement already satisfied: triton==3.5.1 in /usr/local/lib/python3.12/dist-packages (from torch>=1.11.0->sentence-transformers) (3.5.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn->sentence-transformers) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn->sentence-transformers) (3.6.0)
Requirement already satisfied: anyio in /usr/local/lib/python3.12/dist-packages (from httpx<1,>=0.23.0->langsmith<1.0.0,>=0.1.125->langchain-community
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.12/dist-packages (from httpx<1,>=0.23.0->langsmith<1.0.0,>=0.1.125->langchain-c
Requirement already satisfied: h11>=0.16 in /usr/local/lib/python3.12/dist-packages (from httpcore==1.*->httpx<1,>=0.23.0->langsmith<1.0.0,>=0.1.125->
Requirement already satisfied: jsonpointer>=1.9 in /usr/local/lib/python3.12/dist-packages (from jsonpatch<2.0.0,>=1.33.0->langchain-core<2.0.0,>=1.1.
Requirement already satisfied: ormsgpack>=1.12.0 in /usr/local/lib/python3.12/dist-packages (from langgraph-checkpoint<4.0.0,>=2.1.0->langgraph<1.1.0,
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch>=1.11.0->sentence-transformers
Requirement already satisfied: mypy-extensions>=0.3.0 in /usr/local/lib/python3.12/dist-packages (from typing-inspect<1,>=0.4.0->dataclasses-json<0.7.
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->torch>=1.11.0->sentence-transformers) (3.0.3)
```

```
[5]
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
```

colab.research.google.com/drive/1Z8-krFp3ggLRfYku8g7qmPHuk_cj2Y2Z#scrollTo=aDKV-hLXY4pU
Untitled2.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM Disk
Variables Terminal
17:39 Python 3
22°C Windy
18:02 08/12/2025

Figure 1: screenshot showing code on importing libraries

Step 2: Loading the PDF Document

In this step, I loaded the PDF file using PyPDFLoader. I did this to convert the PDF into text-based documents that can be processed by LangChain. Loading the document allowed me to extract the raw content that would later be chunked and embedded.

Code:

# --- Load PDF ---

loader = PyPDFLoader("Economies.pdf")

docs = loader.load()

Step 3: Splitting the Document into Chunks

In this step, I used RecursiveCharacterTextSplitter to divide the document into manageable chunks with a chunk size of 500 characters and an overlap of 50. I did this because splitting long documents into smaller sections improves embedding quality and ensures the retriever can return precise, context-specific matches.

Code:

```
# --- Split Documents into Chunks ---

splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)

chunks = splitter.split_documents(docs)
```

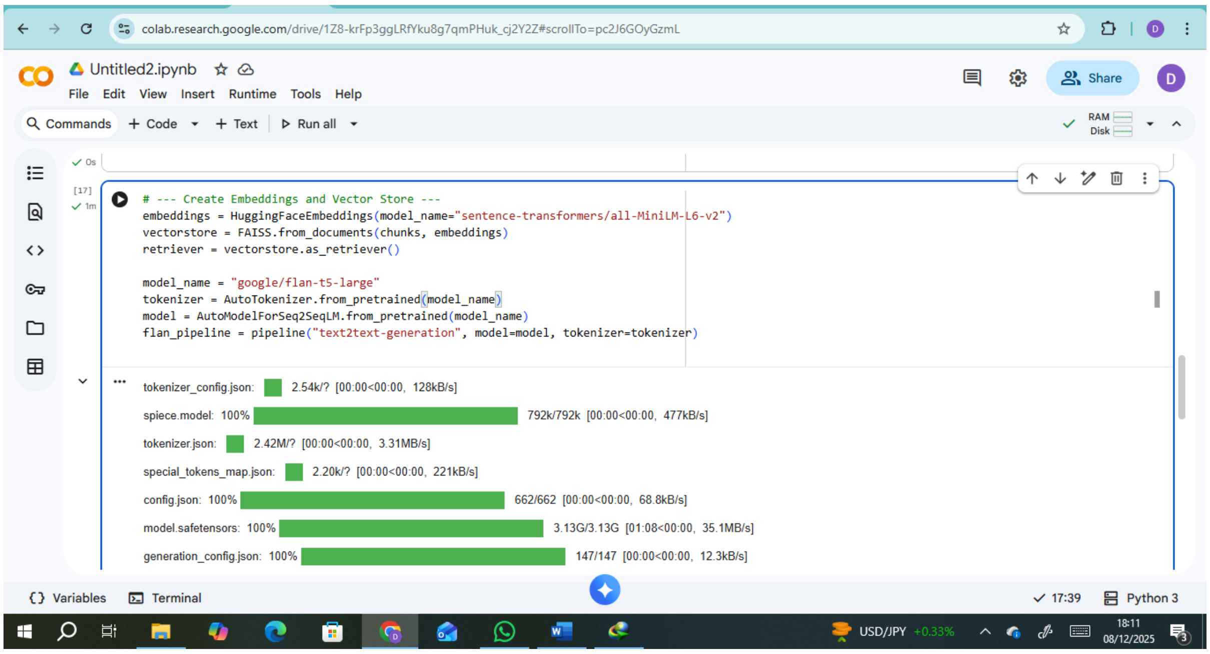
Step 4: Generating Embeddings and Creating the Vector Store

In this step, I created embeddings using the "all-MiniLM-L6-v2" Sentence-Transformer model and stored them in a FAISS vector database. I did this to convert each document chunk into numerical representations that allow for efficient similarity search. The vector store is what enables fast and accurate retrieval during question answering.

Code:

```
# --- Create Embeddings and Vector Store ---

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

vectorstore = FAISS.from_documents(chunks, embeddings)

retriever = vectorstore.as_retriever()
```

screenshot:



```
# --- Create Embeddings and Vector Store ---
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
vectorstore = FAISS.from_documents(chunks, embeddings)
retriever = vectorstore.as_retriever()

model_name = "google/flan-t5-large"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
flan_pipeline = pipeline("text2text-generation", model=model, tokenizer=tokenizer)
```

```
tokenizer_config.json:  2.54k/? [00:00<00:00, 128kB/s]
spiece.model: 100%  792k/792k [00:00<00:00, 477kB/s]
tokenizer.json:  2.42M/? [00:00<00:00, 3.31MB/s]
special_tokens_map.json:  2.20k/? [00:00<00:00, 221kB/s]
config.json: 100%  662/662 [00:00<00:00, 68.8kB/s]
model.safetensors: 100%  3.13G/3.13G [01:08<00:00, 35.1MB/s]
generation_config.json: 100%  147/147 [00:00<00:00, 12.3kB/s]
```

Figure 2: screenshot showing code for embeddings

Step 5: Initializing the Language Model (LLM)

In this step, I loaded the "google/flan-t5-large" model along with its tokenizer, and created a text-generation pipeline. I did this because the LLM is responsible for generating final answers using both the retrieved context and the user's question. FLAN-T5 is suitable for instruction-following and question-answering tasks.

Code:

model_name = "google/flan-t5-large"

tokenizer = AutoTokenizer.from_pretrained(model_name)

model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

flan_pipeline = pipeline("text2text-generation", model=model, tokenizer=tokenizer)

Step 6: Creating the RAG Query Function

In this step, I wrote the query_rag() function, which retrieves relevant chunks based on the user's question, constructs a context-aware prompt, and sends it to the FLAN-T5 model for generation. I did this to implement a complete RAG workflow where retrieval and generation work together—retrieval provides grounded facts, and the LLM synthesizes a final answer.

Code:

```
def query_rag(question):
    # Retrieve relevant chunks
    relevant_docs = retriever.invoke(question)

    # Combine into context
    context = "\n".join([doc.page_content for doc in relevant_docs])

    # Create prompt
    prompt = f"Answer the question using only the context:\n\nContext:\n{context}\n\nQuestion: {question}\n\nAnswer:"

    # Generate answer
    response = flan_pipeline(
        prompt,
        max_new_tokens=200,
        temperature=0.9,
        top_k=50,
        top_p=0.9,
        do_sample=True
    )
```

return response[0]['generated_text']

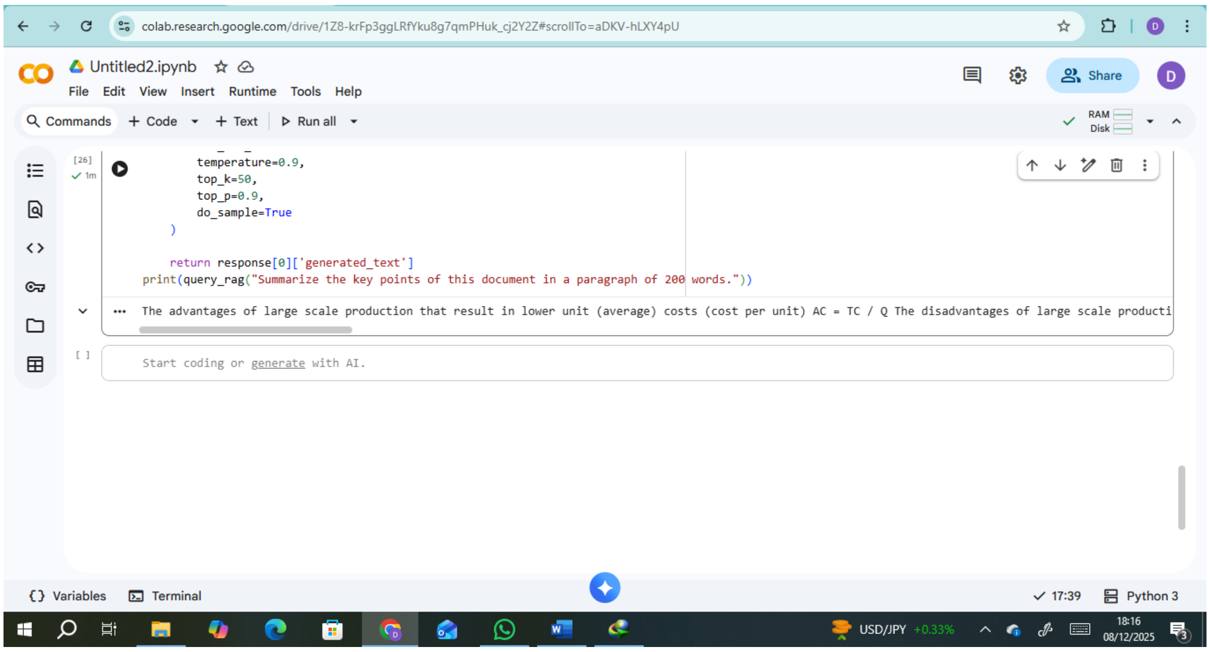
print(query_rag("Summarize the key points of this document in a paragraph of 200 words."))

Step 7: Running a Sample Query Using RAG

In this step, I tested the pipeline by asking a question:

"Summarize the key points of this document in a paragraph of 200 words."

I did this to verify that the RAG system works correctly, retrieves relevant document content, and generates an answer grounded strictly in the PDF context. This step validated the entire pipeline from document ingestion to final answer generation.


colab.research.google.com/drive/1Z8-krFp3ggLRfYku8g7qmPHuk_cj2Y2Z#scrollTo=aDKV-hLXY4pU
Untitled2.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
Variables
Terminal
17:39
Python 3
USD/JPY +0.33%
18:16
08/12/2025

```
        temperature=0.9,
        top_k=50,
        top_p=0.9,
        do_sample=True
    )

    return response[0]['generated_text']
print(query_rag("Summarize the key points of this document in a paragraph of 200 words."))
```

```
The advantages of large scale production that result in lower unit (average) costs (cost per unit) AC = TC / Q The disadvantages of large scale producti
```

```
Start coding or generate with AI.
```

Figure 3: screenshot showing the result of step 7

Link to Code

Link to Code: https://colab.research.google.com/drive/1Z8-krFp3ggLRfYku8g7qmPHuk_cj2Y2Z?usp=sharing

Conclusion

In this assignment, I successfully implemented a complete Retrieval-Augmented Generation (RAG) pipeline that integrates document processing, embedding, vector storage, and generative question answering. By splitting a PDF into meaningful chunks, generating embeddings using Sentence-Transformers, and storing them in a FAISS vector store, I learned how retrieval systems enable efficient and accurate context lookup. I also integrated the FLAN-T5 model to generate answers grounded in retrieved content, demonstrating how retrieval significantly improves the relevance and reliability of generative AI responses.

This project strengthened my understanding of generative AI concepts, vector databases, embeddings, and prompt engineering.