Large Language Models and Their Applications (UE22AM343BB4): Case Study

NAME:

SOURABH R KAKRANNAYA (PES1UG22AM164)

Vallapuri Jagapathi (PES1UG22AM183)

SEC: C SEM:6

DATE: 11-03-2025

NUMBER OF DOCUMENTS USED: 10

GITHUB REPO : https://github.com/Jagapathi-Vallapuri/LLMCaseStudy2

QUESTIONS USED FOR EVALUATION:

1. How much did Apple spend on Research and Development in fiscal year 2018, and how did it change compared to 2017?
2. What were the total iPhone sales figures (in units) for Apple in 2018, and how did this compare to the previous year?
3. How many shares did Microsoft repurchase in fiscal year 2016, and what was the total amount spent?
4. What was Apple's net sales figure for the Americas region in 2018, and what percentage of total net sales did this represent?
5. When did Microsoft acquire LinkedIn Corporation according to the quarterly information?
6. What was the dividend per share declared by Microsoft in September 2015?
7. What factors contributed to the increase in iPad net sales during 2018 compared to 2017?
8. How much did Apple's Services segment contribute to total net sales in 2018, and what was the year-over-year growth percentage?
9. What were the main components of Microsoft's "Other Income (Expense), Net" for fiscal year 2018?
10. What was Apple's gross margin percentage range anticipated for the first quarter of 2019?

OUTPUT:

```
Helpful Answer: In fiscal year 2018, Apple spent $11,988 million on
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: The total iPhone sales figures (in units) for Apple in 2018 were 2
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: Microsoft repurchased 148 million shares in fiscal year 2016, and
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: In 2018, Apple's net sales for the Americas region were $1
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: Microsoft acquired LinkedIn Corporation on December 8, 2016.
```

```
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: The dividend per share declared by Microsoft in September 2015 was $0.3
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: The increase in iPad net sales during 2018 compared to 2017 was
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: In 2018, Apple's Services segment contributed $37,190
--------------------------------------------------------------------------------
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.
Helpful Answer: The main components of Microsoft's 'Other Income (Expense), Net' for fiscal year
--------------------------------------------------------------------------------
Helpful Answer: The anticipated gross margin percentage range for the first quarter of 2019 is between 3
--------------------------------------------------------------------------------
```

PREPROCESSING STAGE:

1. `clean_text(text)` – Text Cleaning Function

Purpose: Cleans raw text extracted from the PDF.

- **Removes page numbers (e.g., "Page 1", "Page 2").**
- **Removes non-ASCII characters (e.g., special symbols that may not be useful).**
- **Normalizes whitespace (removes unnecessary spaces).**

How It Works:

```
def clean_text(text):
    text = re.sub(r"Page\s?\d+", "", text)          # Remove "Page X"
    text = re.sub(r"[^\x00-\x7F]+", " ", text)      # Remove non-ASCII
characters
    text = re.sub(r"\s+", " ", text).strip()        # Normalize whitespace
    return text
```

Example Input & Output:

```
Page 1
This is a test document.    It contains   unnecessary spaces.
© 2024 Some Symbol
```

Output After Cleaning:

```
This is a test document. It contains unnecessary spaces.
```

---

2. `extract_text(pdf_path)` – Extract Text from PDF

Purpose: Reads a PDF file and extracts text from all pages.

- **Uses `pdfplumber` to open the PDF.**
- **Iterates through each page and extracts text.**
- **Cleans the extracted text using `clean_text()`.**
- **Preserves page numbers in the output to maintain document structure.**

How It Works:

```
def extract_text(pdf_path):
    full_text = []

    with pdfplumber.open(pdf_path) as pdf:  # Open PDF file
        for page_num, page in enumerate(pdf.pages, start=1):
            page_text = page.extract_text() or ""  # Extract text from
page
            cleaned_text = clean_text(page_text)  # Clean extracted text
            full_text.append(f"Page {page_num}:\n{cleaned_text}\n")  #
Preserve page numbers

    return "\n".join(full_text)  # Return extracted text as a single
string
```

Example Input & Output:

PDF Content:

```
Page 1
Company Report 2024
Revenue: $1M
```

```
Page 2
Expenses: $500K
Net Profit: $500K
```

Output (Extracted & Cleaned Text):

```
Page 1:
Company Report 2024 Revenue: $1M

Page 2:
Expenses: $500K Net Profit: $500K
```

---

3. `extract_tables(pdf_path)` **– Extract Tables from PDF**

Purpose: Extracts tables from a PDF and saves them as CSV files.

- **Uses** `tabula.read_pdf()` **to detect and extract tables.**
- **If tables are found, they are saved as CSV files in the output directory.**
- **If no tables are found, it logs a warning.**
- **If an error occurs, it logs an error message.**

How It Works:

```
def extract_tables(pdf_path):
    table_paths = []
    try:
        tables = read_pdf(pdf_path, pages="all", multiple_tables=True)  # Extract tables from all pages

        if not tables:
```

```
            logging.warning(f"No tables found in {pdf_path}")  # Log
warning if no tables are found


        for i, table in enumerate(tables):
            # Generate output CSV file path
            output_csv_path = os.path.join(output_dir,
f"{os.path.basename(pdf_path).replace('.pdf', '')}_table_{i + 1}.csv")
            table.to_csv(output_csv_path, index=False)  # Save table as
CSV
            table_paths.append(output_csv_path)  # Store CSV path


    except Exception as e:
        logging.error(f"Error extracting tables from {pdf_path}: {e}")  #
Log error if table extraction fails


    return table_paths  # Return list of extracted table paths
```

Example Input & Output:

PDF Table Content:

Year	Revenue	Expenses	Profit
2022	$1M	$500K	$500K
2023	$1.5M	$600K	$900K

Output (CSV File Saved):

```
Year,Revenue,Expenses,Profit
2022,$1M,$500K,$500K
```

```
2023,$1.5M,$600K,$900K
```

---

4. `process_pdf(pdf_path)` – Main PDF Processing Function

Purpose:

- **Calls `extract_text()` and saves the extracted text to a `.txt` file.**
- **Calls `extract_tables()` and saves the extracted tables as CSV files.**
- **Logs the results to track progress.**

How It Works:

```
def process_pdf(pdf_path):
    logging.info(f"Processing: {pdf_path}")  # Log start of processing

    # Extract and save text
    extracted_text = extract_text(pdf_path)
    text_output_path = os.path.join(output_dir,
f"{os.path.basename(pdf_path).replace('.pdf', '')}.txt")

    with open(text_output_path, "w", encoding="utf-8") as text_file:
        text_file.write(extracted_text)  # Save extracted text to file

    logging.info(f"Text saved to: {text_output_path}")  # Log saved text
file

    # Extract and save tables
    table_paths = extract_tables(pdf_path)
```

```
    logging.info(f"Extracted {len(table_paths)} tables from: {pdf_path}")
# Log table count


    if not table_paths:
        logging.warning(f"No tables were extracted from {pdf_path}")  #
Log warning if no tables are extracted
```

Execution Flow:

1. **Logs "Processing PDF..."**
2. **Extracts text → Saves as `.txt` file**
3. **Extracts tables → Saves as `.csv` file(s)**
4. **Logs the output locations and warnings (if any)**

Building the RAG System:

1. Load Extracted Text and Tables

Step 1: Load Extracted Text from File

```
with open(text_file_path, "r", encoding="utf-8") as file:
    extracted_text = file.read()
```

- **Reads the previously extracted text from the `.txt` file.**
- **This text was extracted using `pdfplumber` in an earlier step.**

Step 2: Load Extracted Tables and Convert to Text

```
table_texts = []
for filename in os.listdir(output_dir):
```

```
    if filename.startswith("combined_document_10_table_") and
filename.endswith(".csv"):
        csv_path = os.path.join(output_dir, filename)
        try:
            df = pd.read_csv(csv_path)
            table_text = f"Table: {filename}\n" +
df.to_string(index=False) + "\n"
            table_texts.append(table_text)
        except Exception as e:
            logging.error(f"Error reading {csv_path}: {e}")
```

- **Reads all extracted table files from the output directory.**
- **Converts CSV data into string format for embedding.**
- **Each table is stored as a formatted string in `table_texts[]`.**

Step 3: Combine Text and Table Data

```
full_text = extracted_text + "\n\n" + "\n\n".join(table_texts)
```

- **Merges extracted text and tables into a single document.**
- **This ensures that both textual content and structured table data can be used for retrieval.**

---

2. Convert Text to Embeddings

Step 4: Initialize Hugging Face Embeddings

```
device = "cuda" if torch.cuda.is_available() else "cpu"
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2", model_kwargs={"device": device})
```

- Uses Hugging Face sentence embeddings (`MiniLM-L6-v2`) to convert text chunks into vector representations.
- Uses GPU if available, otherwise defaults to CPU.

---

3. Text Chunking for Retrieval

Step 5: Split Text into Smaller Chunks

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
text_chunks = text_splitter.split_text(full_text)
```

- Divides the extracted text into manageable segments for better retrieval.
- Each chunk is 500 characters long with 100-character overlap to maintain context.
- This is essential because LLMs have token limits and work better with smaller inputs.

---

4. Vector Indexing Using FAISS

Step 6: Create FAISS Index for Fast Retrieval

```
faiss_index_path = os.path.join(output_dir, "faiss_index")
```

```
vector_store = FAISS.from_texts(text_chunks, embedding_model)
vector_store.save_local(faiss_index_path)
```

- **Converts text chunks into vector embeddings and stores them in FAISS.**
- **FAISS (Facebook AI Similarity Search) is an efficient vector database used for fast retrieval of similar text chunks.**

Step 7: Load FAISS Index for Querying

```
vector_store = FAISS.load_local(faiss_index_path,
embedding_model, allow_dangerous_deserialization=True)
```

- **Reloads the FAISS index so that the retrieval system can access it.**

---

5. Load LLM for Response Generation

Step 8: Load Mistral-7B LLM

```
pipe = pipeline(
    "text-generation",
    model="mistralai/Mistral-7B-Instruct-v0.3",
    model_kwargs={"torch_dtype": torch.bfloat16},
    device="cuda",
)
```

- **Loads Mistral-7B, a lightweight open-source LLM, optimized for factual response generation.**
- **Uses `torch.bfloat16` to reduce memory usage while maintaining accuracy.**
- **Runs on GPU if available.**

Step 9: Wrap the Model for LangChain

```
llm = HuggingFacePipeline(pipeline=pipe)
```

- **Converts the Mistral-7B model into a LangChain-compatible LLM pipeline.**

---

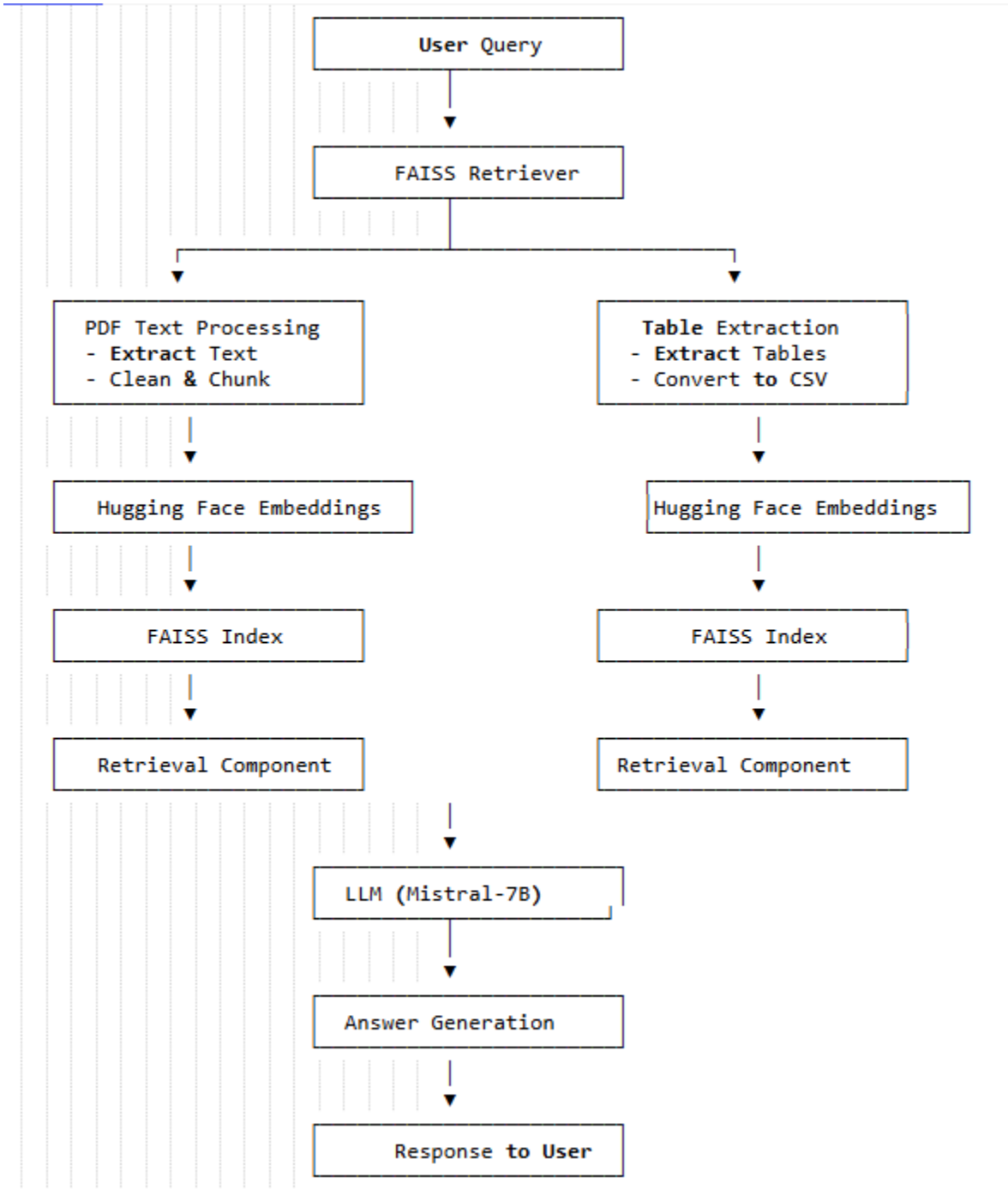
6. Build the Retrieval-Augmented Generation (RAG) Pipeline

Step 10: Create RAG Pipeline

```
qa_chain = RetrievalQA.from_chain_type(llm=llm,
retriever=vector_store.as_retriever())
```

- **Combines the retriever (FAISS) and the generator (Mistral-7B LLM).**
- **Retrieval Component: Finds relevant document chunks from FAISS based on user queries.**
- **Generation Component: Uses Mistral-7B to generate context-aware responses using the retrieved documents.**

DIAGRAM:


User Query
FAISS Retriever
PDF Text Processing
- Extract Text
- Clean & Chunk
Table Extraction
- Extract Tables
- Convert to CSV
Hugging Face Embeddings
Hugging Face Embeddings
FAISS Index
FAISS Index
Retrieval Component
Retrieval Component
LLM (Mistral-7B)
Answer Generation
Response to User

Retrieval-Augmented Generation (RAG) System Using LangChain

1. Theory of Retrieval-Augmented Generation (RAG)

Introduction

Retrieval-Augmented Generation (RAG) is an advanced AI framework that enhances the response generation capabilities of Large Language Models (LLMs) by incorporating external document retrieval. Unlike standalone generative models, RAG first searches for relevant context from a knowledge base before generating responses, ensuring improved accuracy, relevance, and factual consistency.

How RAG Works

A RAG system consists of two main components:

1. **Retriever – This module searches for the most relevant document chunks based on user queries. It uses vector embeddings and similarity search techniques to retrieve contextual information from a pre-indexed document database.**
2. **Generator – Once the retriever fetches the relevant context, an LLM generates responses using the retrieved information as additional context. This mitigates the problem of hallucinations in LLMs and ensures better factual correctness.**

Advantages of RAG Systems

- **Enhanced Accuracy: Responses are grounded in retrieved documents rather than solely relying on LLM's pre-trained knowledge.**
- **Scalability: Can process and search through large document sets quickly using vectorized indexing.**
- **Dynamic Knowledge Base: Unlike traditional models, RAG systems can be updated with new documents without retraining the entire model.**

---

2. Technologies Used in the RAG System

2.1 Data Ingestion and Preprocessing

- **PDF Processing:**
  - `PyPDF2` or `pdfplumber` for extracting text from PDFs.
  - `tabula-py` or `camelot-py` for extracting tabular data from structured reports.
- **Text Cleaning & Chunking:**
  - Removal of unwanted headers, footers, and formatting artifacts.
  - Segmentation of text into smaller chunks for efficient retrieval.

2.2 Embedding and Indexing

- **Vector Embeddings:**
  - Text chunks are converted into vector representations using models like OpenAI's `text-embedding-ada-002` or open-source alternatives such as `sentence-transformers`.
- **Vector Databases:**
  - `FAISS`, `Pinecone`, or `ChromaDB` for similarity search and efficient document retrieval.

2.3 RAG Pipeline Implementation

- **LangChain Framework:**
  - Provides an abstraction layer to connect retrievers and LLMs for structured RAG pipeline development.
- **LLMs Used:**
  - Open-source models like `LLaMA`, `Mistral`, `Gemma`, or Hugging Face's `Flan-T5`.
  - Optionally, `Groq API` can be used for faster inference.

2.4 Optimization and Deployment

- **Quantization for Efficiency:**
  - `bitsandbytes` library for reducing model memory consumption with 4-bit or 8-bit quantization.
- **Deployment on Cloud Resources:**
  - `Google Colab` or `Kaggle` for free GPU-based execution.

- **User Interface (Optional):**
  - `Streamlit` **for developing an interactive web-based query input and result display system.**

Conclusion

This RAG system implementation effectively combines document retrieval and language generation to enhance information extraction from large PDF datasets. By leveraging vector embeddings, open-source LLMs, and cloud-based GPU execution, the solution is scalable and optimized for performance, making it ideal for real-world applications in research, finance, and knowledge management.

Would you like any refinements to this?