INTELLIGENT MULTIFORMAT DOCUMENT SUMMARIZATION AND Q&A

Name: Monisha Patro

Contact: monishaapatro@gmail.com

USING LLAMA 3.1 (8B) VIA GROQCLOUD

ABSTRACT

In today's data-driven landscape, vast volumes of text files—ranging from PDF research papers to corporate DOCX reports and everyday TXT documents—necessitate efficient methods of extraction, summarization, and question-answering. This project, **Intelligent Multiformat Document Summarization and Q&A Using Llama 3.1 (8B) via GroqCloud**, aims to unify three core functions within a single application: file uploading and text extraction, chunk-based summarization with parallel processing, and intuitive Q&A based on an entire document context.

A defining element of this solution is the integration of the **Llama 3.1 (8B)** language model via **GroqCloud**. Rather than hosting large language models locally and wrestling with GPU memory constraints, the project leverages GroqCloud's production-ready API. The synergy of open-source toolkits (LangChain for chunking, pdfplumber for PDF reading, docx for DOCX files, and Python's built-in methods for TXT) provides a high-performance pipeline that can parse diverse document types. Sensitive data redaction is built into every step, ensuring privacy through advanced regex detection. Finally, deployment on Streamlit's Cloud platform offers a free, browser-accessible demonstration, enabling real-time interactions and easy scaling to additional users.

This project stands at the intersection of convenience, speed, and reliability. By showing how a user can upload any large textual document, automatically

produce a meaningful summary, and receive context-sensitive Q&A answers, we demonstrate how state-of-the-art large language model capabilities can be applied practically. From academic research and corporate analyses to personal notes and beyond, the system addresses the growing need to handle massive textual data intelligently and securely.

INTRODUCTION

In an era where information is both abundant and critical, retrieving relevant data can be just as difficult as creating it. Whether you are dealing with thousands of pages of research publications, extensive financial spreadsheets (converted into PDF or DOCX), or simpler textual logs, the effort to manually summarize and glean insights can be enormous. Traditional search engines and manual reading approaches fall short when documents run into tens or hundreds of pages, each requiring meticulous reading and highlight extraction.

Recent breakthroughs in Large Language Models (LLMs) have broadened horizons for automatic document processing. LLMs can analyze unstructured textual data, identify main ideas or themes, and even engage in question-answering by referencing specific parts of the text. However, straightforward application of LLMs can face bottlenecks, such as the model's limited context window or the high cost of inference when dealing with extensive texts in a single pass.

This project introduces an **intelligent pipeline** that marries chunk-based approaches with parallel execution, thereby accommodating large documents while maintaining performance. We adopted an open-source mindset for the majority of the workflow, employing pdfplumber for PDF reading, python-docx for DOCX parsing, basic Python I/O for TXT files, and **LangChain's text-splitting** to chunk up massive text. We also integrated **sensitive data redaction** through regex, ensuring that private information remains protected during summarization or Q&A tasks.

The standout piece is the **Llama 3.1 (8B)** model, a robust LLM hosted by **GroqCloud**, which handles all the actual NLP tasks. By outsourcing the model inference to GroqCloud, we offload the complexities of GPU provisioning, memory management, and concurrency handling, letting us focus on the user-facing aspect and chunk-based logic. The final layer—**Streamlit**—presents everything in

a streamlined, user-friendly interface where individuals can upload a file, summarize it, and pose queries about the text with minimal friction.

Why?

1. **Rising Volume of Textual Data**

   Nearly every domain is affected by the exponential growth in textual data. Analysts, researchers, and professionals across sectors continually need to parse large archives of documents. A robust summarization and Q&A solution drastically shortens the time it takes to glean critical information.

2. **Bridging Usability & Advanced NLP**

   While advanced NLP models exist, bridging the gap between cutting-edge research and end-user accessibility remains a challenge. We wanted to create a pipeline that anyone, even those without deep ML or devops backgrounds, could appreciate and operate.

3. **Encouraging Responsible AI Use**

   Automated solutions often risk exposing private or sensitive data, especially in confidential documents. Our built-in regex-based redaction ensures an ethical dimension to summarization, demonstrating how a project can incorporate privacy measures from the ground up.

4. **Showcasing a Full Stack Approach**

   From reading multi-format inputs to chunk-splitting and parallel requests, from advanced model calls to front-end design—this project demonstrates a comprehensive end-to-end approach, valuable for students, developers, or companies seeking a template for large-scale text processing.

Llama 3.1 (8B) MODEL

Llama 3.1 (8B) is one of the more recent and balanced LLMs, delivering efficient and high-quality inferences without requiring enormous hardware overhead. To provide better context, we compared multiple potential solutions:

1. **GPT-4**

- **Pros**: Excellent reasoning, widely recognized name, easy integration with OpenAI's API.
- **Cons**: Potentially higher costs, rate limiting, some usage restrictions.
- **Why We Didn't Choose It**: For large documents with chunk-based parallel calls, cost can grow quickly. Also, fine-grained control can be more limited.

2. **BART or T5**
    - **Pros**: Open-source, decent summarization capabilities.
    - **Cons**: Typically smaller context windows, may require local hosting if we want to avoid huggingface or other hosting solutions.
    - **Why We Didn't Choose It**: They don't always produce as robust Q&A as Llama, and handling large documents might require more manual chunking logic.
3. **Llama 2 (7B or 13B)**
    - **Pros**: High-quality open-source model by Meta, widely recognized in the open community.
    - **Cons**: Setting it up to run efficiently might still require specialized GPU infrastructure. Licensing and sign-up processes can be more involved.
    - **Why We Didn't Choose It**: We found a *newer*, more context-friendly option via GroqCloud in Llama 3.1, which gave us a faster environment and 8B parameter scale.
4. **Llama 3.1 (8B)**
    - **Pros**: Balanced parameter count for speed and accuracy; 8B is big enough for multi-domain knowledge, while still performing well in near real-time. GroqCloud offers immediate production-level hosting.
    - **Cons**: Larger than some minimal models, so concurrency might need attention.
    - **Conclusion**: We selected Llama 3.1 (8B) because it merges the best of both worlds—strong performance and manageable inference overhead, plus direct support from GroqCloud's infrastructure.

Hence, **Llama 3.1 (8B)** is well-suited for a demonstration of summarizing big text while answering queries with good reasoning. It stands at a sweet spot between smaller (less capable) and massive (costly or slow) models.

GROQCLOUD

GroqCloud is a platform for deploying and running large language models at scale, offering:

- **API Endpoints**: Instead of building local GPU clusters, we simply call a hosted model with a straightforward REST interface.
- **Production Tier**: Our model ID—"llama-3.1-8b-instant"—is known to be stable, with potential for concurrency and higher context windows, if needed.
- **Competitive Performance**: By focusing on specialized hardware and efficient inference frameworks, GroqCloud can return chat completions quickly.
- **Reduced Operational Complexity**: We only keep track of an API key and an endpoint, letting us focus on the summarization logic rather than GPU memory usage or concurrency locks.

In essence, GroqCloud's environment is what makes the entire pipeline feasible without huge overhead. We can parallelize chunk requests and still rely on the platform's ability to handle concurrency gracefully (subject to rate limits).

PROCESS

Below is a thorough, step-by-step explanation of how the application works, from uploading a file to returning the final summary or answer. We'll break down each Python function and the overarching architecture.

1. Project Setup and Import Statements

We begin our code (`app.py`) by importing:

- **Streamlit** for the web interface.
- **pdfplumber**, **docx**, and Python's built-in file I/O to handle PDF, DOCX, and TXT files, respectively.
- **Re** for regex redaction.

- **Requests** for making API calls to GroqCloud.
- **ThreadPoolExecutor** from `concurrent.futures` to run summarization tasks concurrently.
- **dotenv** for loading environment variables (the GroqCloud API key).
- **RecursiveCharacterTextSplitter** from LangChain to handle chunk-based splitting.

2. Loading the Environment Variable

```
from dotenv import load_dotenv
load_dotenv()
```

This snippet ensures we read `GROQCLOUD_API_KEY` from either a `.env` file locally or from environment variables set in Streamlit Cloud's "Secrets." If the key is not found, we raise an error to avoid undefined behavior later.

3. Sensitive Data Redaction

We define a dictionary:

```
SENSITIVE_PATTERNS = {
  "PHONE": r"(\+?\d[\d\-\(\) ]{7,}\d)",
  "EMAIL": r"([a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)",
}
```

And a function `detect_and_redact(text)` which uses Python's `re.sub()` to replace matches with a placeholder like `<REDACTED:PHONE>` or `<REDACTED:EMAIL>`. This ensures that personal info does not appear in the final summary or Q&A context.

4. Reading Different File Formats

We define:

- `read_pdf_file(file_path)` with **pdfplumber**, iterating over each page's extracted text.
- `read_docx_file(file_path)` using the **docx** library's `Document` object.
- `read_txt_file(file_path)` with a simple open/read approach.

When the user uploads a file in Streamlit, we temporarily save it locally, check its extension, and call the appropriate function to produce a single large string of text.

5. GroqCloud Chat Inference (`call_groqcloud_chat()`)

A single function handles the entire cycle of sending a request to the model and retrieving the response:

```
def call_groqcloud_chat(user_prompt, system_prompt="", max_tokens=300, t
emperature=0.7):
    api_key = os.getenv("GROQCLOUD_API_KEY")
    # if not found, raise ValueError

    url = "https://api.groq.com/openai/v1/chat/completions"
    headers = {"Authorization": f"Bearer {api_key}", "Content-Type": "applicatio
n/json"}

    # Build an array of messages with optional system prompt
    messages = []
    if system_prompt:
        messages.append({"role": "system", "content": system_prompt})
    messages.append({"role": "user", "content": user_prompt})

    payload = {
        "model": "llama-3.1-8b-instant",
        "messages": messages,
        "max_completion_tokens": max_tokens,
        "temperature": temperature
    }

    response = requests.post(url, json=payload, headers=headers)
    if response.status_code != 200:
        # Show an error in Streamlit and print to logs
        st.error(f"GroqCloud call failed: status={response.status_code}, body={r
esponse.text}")
```

```
        print("GroqCloud error details:", response.status_code, response.text)
        raise RuntimeError(f"GroqCloud API call failed with code {response.statu
s_code}")

    data = response.json()
    try:
        return data["choices"][0]["message"]["content"]
    except (KeyError, IndexError):
        return "[Error: No text returned from GroqCloud response]"
```

Key Observations:

- We rely on **chat-based** completions, so we pass a list of messages with roles like “system” and “user.”
- We handle any non-200 status code by raising an exception. This ensures the user sees an appropriate error in Streamlit if, for example, the key is invalid or we exceed some concurrency limit.

6. Summarizing a Document (`summarize_document()`)

1. **Redaction**: The entire document text is first passed to `detect_and_redact(text)`.
2. **Chunk Splitting**:
    - We use `RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=200)`. This ensures each chunk is around 2,000 characters, with a 200-character overlap to preserve context continuity.
    - Why chunk? We surpass LLM context windows more easily and can parallelize the summarization calls, avoiding a single massive request.
3. **Parallel Summaries**:
    - We define `summarize_chunk(chunk)` which calls `call_groqcloud_chat()` with a short system prompt: “You are a helpful assistant. Summarize the user content briefly.”
    - We run these chunk operations in a `ThreadPoolExecutor(max_workers=4)`. For each chunk, we retrieve the partial summary.
4. **Final Combination**:

- We join all partial summaries into `combined_text`.
- Then we make a second call to refine them with a prompt like “Combine and refine these partial summaries...”.
- The final summary is returned to the user.

7. Q&A Workflow (`answer_question()`)

1. We redact the doc text.
2. Provide a system prompt “You are a helpful assistant. Use the user’s text to answer their question.”
3. Concatenate the entire doc with the question in a single user prompt.
4. Use `call_groqcloud_chat()` to get the direct answer from GroqCloud.
5. Because the doc can be long, we might later incorporate chunk-based retrieval (RAG). But for now, we rely on the entire doc as a single context, assuming it is within feasible context limits or that chunk splitting is not absolutely required for Q&A if the doc is not too large.

8. Streamlit UI

Finally, we have `main()`:

1. **Page Config**: `st.set_page_config(...)` sets the page layout and title.
2. **Uploader**: `st.file_uploader(...)` allows the user to pick a file.
3. We temporarily save the file to disk and detect if it ends with “.pdf,” “.docx,” or “.txt.” The appropriate reading function is called.
4. We remove the temporary file.
5. The **Preview**: We display up to the first 500 characters.
6. **Summarize Button**: If pressed, calls `summarize_document(doc_text)` and displays the results with `st.success(summary)`.
7. **Q&A Section**: The user enters a question in `st.text_input(...)`, and upon submission, we use `answer_question(...)` to produce a response.
8. We wrap each action in `st.spinner` so the UI indicates “Summarizing...” or “Generating answer...”.

DEPLOYMENT

1. Local Testing

We initially tested the script locally by:

- Installing dependencies from `requirements.txt`
- Setting `GROQCLOUD_API_KEY` in `.env`
- Running `streamlit run app.py`

We confirmed correct behaviors in a local environment. Typical issues that can arise here include Python library versions, concurrency issues, or missing API keys. Once all worked locally, we proceeded.

2. Streamlit Cloud

For a **public** demonstration, we used the free-tier hosting on Streamlit Cloud:

1. **Push** the project to GitHub (ensuring `.env` and any secrets remain local).
2. **Create** a Streamlit Cloud app, referencing the main `app.py`.
3. **Add** our `GROQCLOUD_API_KEY` in the "Secrets" panel, under the same name, so it's accessible via `os.getenv("GROQCLOUD_API_KEY")`.
4. **Deploy**. Once completed, we get a shareable URL.
5. **Test**: The same file upload logic works in the hosted environment. If concurrency or rate-limit errors arise, we see them in the logs.

Thus, the entire pipeline is now publicly accessible. Anyone with the link can upload a file and see near real-time summarization or Q&A.

FUTURE IMPROVEMENTS

1. **RAG (Retrieval-Augmented Generation)**

   For massive documents (hundreds of pages) or entire knowledge bases, chunk-based retrieval with embedding stores (FAISS, ChromaDB) can drastically reduce the context overhead and improve Q&A precision. Summaries and answers become more targeted as only the relevant chunks feed into each LLM call.

2. **Fine-tuning / Domain Tuning**

   If a specific domain—e.g., legal or scientific—were the focus, we could incorporate domain adaptation or fine-tuning (if GroqCloud's environment or another service supports custom training). That yields more accurate summaries with domain jargon and context.

3. **Advanced Redaction**

   We currently target phone/email. This can be extended to detect and sanitize addresses, names, credit card patterns, or region-specific sensitive data like social security numbers.

4. **Enhanced UI**

   - A more sophisticated file manager to handle multiple documents at once.
   - Real-time Q&A without clicking a "submit" button, possibly with streaming tokens.
   - A comprehensive interactive session featuring memory or conversation history.

5. **Scaling Concurrency**

   If many users simultaneously want to summarize large docs, we might adapt to a queue or specialized concurrency management. That could also involve a paid plan on GroqCloud with higher rate limits.

6. **Metadata Extraction**

   We might augment the approach with "metadata-first" extraction from PDF documents—like table of contents or footnotes. Summaries then become more structured, referencing specific sections or page ranges.

CONCLUSION

Intelligent Multiformat Document Summarization and Q&A Using Llama 3.1 (8B) via GroqCloud provides a model for how advanced NLP tasks can be democratized and streamlined. Instead of building and maintaining large local infrastructure, we utilize a powerful hosted LLM, paralleling chunk-based requests for large documents, and safeguarding user data with regex-based redaction. Deployed on Streamlit Cloud, the entire solution is accessible to anyone with the

link, showcasing how swiftly an AI-driven summarization and Q&A system can be spun up, tested, and utilized in real-world settings.

From commercial applications (like scanning business contracts) to academic pursuits (summarizing lengthy research papers), the system's ability to yield immediate, concise overviews and handle user queries has immense value. More critically, the solution's modular design—redaction, chunking, API calls, and a lightweight UI—means it can readily evolve. As the field of large language models advances, we can swap in new models or add retrieval-based pipelines to expand capabilities further. Ultimately, this project stands as a solid foundation to highlight how practical and beneficial LLM-based summarization/Q&A can be when carefully integrated and responsibly deployed.