REPORT

1. Data Handling

Approach:

- Data Extraction: Extracted resumes from a CSV file and corresponding PDFs. If the resume text was missing in the CSV, the corresponding PDF was used as a fallback.
- Text Extraction: Used the PyPDF2 library to extract text from PDF files, and read data from the CSV using Pandas.

Challenges and Solutions:

- **Handling Missing Data**: Not all CSV entries had complete resume text, necessitating the extraction from PDFs. We resolved this by implementing conditional logic to read from PDFs when CSV data was missing.
- **Inconsistent Text Formats**: Preprocessing was applied to clean up the text by removing unwanted spaces, punctuation, and converting to lowercase, ensuring consistent text for embedding generation.

2. Embedding Generation

Approach:

- **Embedding Model**: Used the HuggingFace Embeddings (all-mpnet-base-v2) to generate embeddings for the resumes.
- **FAISS Integration**: Embeddings were stored in a FAISS index for efficient similarity searches, which allows quick retrieval of relevant resumes based on the query.

Challenges and Solutions:

- **High Dimensional Data**: Embedding generation can be computationally expensive. To address this, FAISS was employed, optimized for fast similarity searches.
- **GPU Utilization**: Integrated FAISS with GPU support (faiss.StandardGpuResources) to improve search speed and handle larger datasets.

3. RAG (Retrieval Augmented Generation) Pipeline Development

Approach:

- **Dual Retrieval**: Combined FAISS and BM25Okapi to retrieve relevant documents. FAISS handled semantic similarity, while BM25 took care of keyword-based retrieval.
- **Result Merging**: Combined results from both FAISS and BM25 and ranked them by frequency to ensure the most relevant resumes were considered in the response.

Challenges and Solutions:

- **Balancing Semantic and Keyword Search**: FAISS and BM25 often retrieve different sets of documents. The solution was to combine their outputs and rank based on the frequency to leverage the strengths of both semantic and keyword retrieval.
- **Scalability**: Efficient indexing with FAISS and BM25 ensures that the pipeline can handle a growing number of resumes without significant performance degradation.

4. API Creation

Approach:

- **Backend Setup**: Developed a FastAPI backend to handle incoming queries and process them using the RAG pipeline.
- **Frontend Integration**: Created a simple frontend using Streamlit to interact with the API, allowing users to input questions and view responses.

Challenges and Solutions:

- **API Response Handling**: Ensured robust error handling by checking API response status codes and capturing any issues during the request.
- **Communication with External API**: Used HuggingFace's Inference API to generate responses from the retrieved texts. Stored the HuggingFace API token as an environment variable for secure access.

Frontend Code Explanation

- A Streamlit-based interface accepts user queries and displays responses.
- Sends requests to the backend API (http://localhost:8000/query/) with the user's input question.
- Handles errors gracefully, providing feedback for both empty inputs and failed API calls.

Backend Code Explanation

- **PDF and CSV Data Handling**: Extracts text data from PDF files and CSV rows.
- **Embedding Generation and Search**: Uses FAISS for vector search and BM25 for keyword matching.
- **Response Generation**: Combines retrieved results and generates an answer using HuggingFace's Inference API model (mistralai/Mistral-Nemo-Instruct-2407).

Infrastructure Challenges and Solutions in Building the RAG Pipeline

1. Initial Attempts with Milvus and AWS EC2

- **Milvus Exploration**: Initially, Milvus wa
- s chosen for vector storage. However, the free AWS EC2 instance was insufficient in terms of resources, necessitating a shift to a paid instance.
- **Outcome**: Despite upgrading to a paid instance, performance issues persisted, making Milvus impractical for this use case. Consequently, Milvus was abandoned, and FAISS was selected as an alternative.

2. Transition to FAISS and GPU Utilization

- **FAISS Implementation**: Running FAISS on a CPU proved to be extremely time-consuming, prompting a switch to GPU-based indexing.
- **Result**: The GPU transition resulted in significantly improved performance, allowing for seamless embedding search and retrieval.

3. Deployment on Hugging Face Spaces

- **Initial CPU Deployment**: Deployed the solution on Hugging Face Spaces using a CPU; however, this approach was inefficient and led to prolonged processing times.
- **Upgrade to GPU**: Opted for a paid GPU instance on Hugging Face Spaces to enhance performance. Despite this upgrade, technical challenges continued to arise.
- **Front-End Integration**: Ultimately, Streamlit was used to build the front end, offering a simple and effective user interface.

4. Local Deployment Attempts

- **Local Challenges**: Running the solution on a local computer encountered similar delays due to CPU limitations, reaffirming the need for GPU-based processing.

Key Challenge

The most significant challenge throughout the project was managing performance issues, especially the extended processing times when using CPUs for embedding generation and retrieval. Switching to GPU instances—despite the additional cost—proved to be the most effective solution to overcome these challenges and ensure smooth execution.