LLAMA, MD: AI Specializing in Gynaecology

NLP-Final Project

Name: Parv Bhargava

I. Introduction

1.1 Overview

This project focuses on fine-tuning a Large Language Model (LLM) for application in the medical domain, specifically targeting expertise in a specialized subfield. The main components of the project are as follows:

1. **Topic Modelling**: We applied topic modelling techniques to identify the top 10 topics within the dataset. After performing frequency analysis, we selected "pregnancy" as the domain for specialization.
2. **Data Retrieval**: A curated collection of 10 comprehensive books on pregnancy was identified, and their digital (PDF) versions were obtained. This dataset serves as the knowledge base for the Retrieval-Augmented Generation (RAG) system.
3. **Fine-Tuning the LLM**: Llama 3.2 (1 billion parameters) was chosen as the base LLM due to its compatibility with our computational resources. The model was fine-tuned using the AI Medical Chatbot dataset to align it with medical conversational tasks.
4. **Retrieval-Augmented Generation (RAG)**: The fine-tuned model was integrated into a RAG framework, resulting in a specialized medical assistant. This system emulates a virtual gynaecology expert, providing accurate and contextually relevant responses related to pregnancy.

This project showcases the capability of domain-specialized LLMs to deliver expert-level assistance in critical healthcare areas. The AI Medical Chatbot dataset, which contains patient questions, doctor responses, and a brief description of the patients' queries, was used for fine-tuning.

1.2 Shared Work

The entire project was a group effort of 4 of us. All of us had to learn so many newer concepts and help each other out at various circumstances. Majorly each of us focused more on two or more aspects of the problem.

1. Raghav and Tanmay initiated the process by conducting topic modelling to identify a specific focus area for model specialization. They further explored and gathered relevant data sources for this topic.
2. Subsequently, the fine-tuning phase was undertaken by Sajan, Raghav, and me, refining the model using the selected dataset.
3. Development of the Retrieval-Augmented Generation (RAG) system was done by Tanmay, Raghav and me
4. While the user interface was designed and implemented on Streamlit by Tanmay and me. All of us worked together in setting up the code structure.
5. Apart from fine-tuning, Sajan worked on model inferencing and manual testing.

II. Overview of Individual Work

I started my work on the project by focusing on the Retrieval-Augmented Generation (RAG) framework. I wrote the code to fetch embeddings and set up and query the vector database, which is implemented in the files embeddings.py and vectordb.py. I also created a preprocessor script to handle the preprocessing of incoming documents. This step was really important because it allowed Tanmay and Raghav to upload documents into Pinecone and perform retrival effectively.

After that, I worked with Sajan on fine-tuning the model. Sajan had already written some fine-tuning code, and I built on top of that to perform instruction fine-tuning. I also spent some time on prompt engineering during the inference stage to make sure the model gave accurate and meaningful responses.

Lastly, I worked on creating a Streamlit app for the project. I set up the basic structure and wrote the initial chatbot code for the app, which Tanmay then improved and added more features to.

III. Explanation

1. Embedding Setup

I started by creating a class to generate vector embeddings from two platforms:

1) **Hugging Face**
2) **Bedrock**

This class allowed us to seamlessly generate vector embeddings regardless of the platform, ensuring flexibility and adaptability in our pipeline. By enabling compatibility with both platforms, it not only streamlined our workflow but also provided an option to switch or expand to other embedding sources in the future, offering significant versatility for the project.

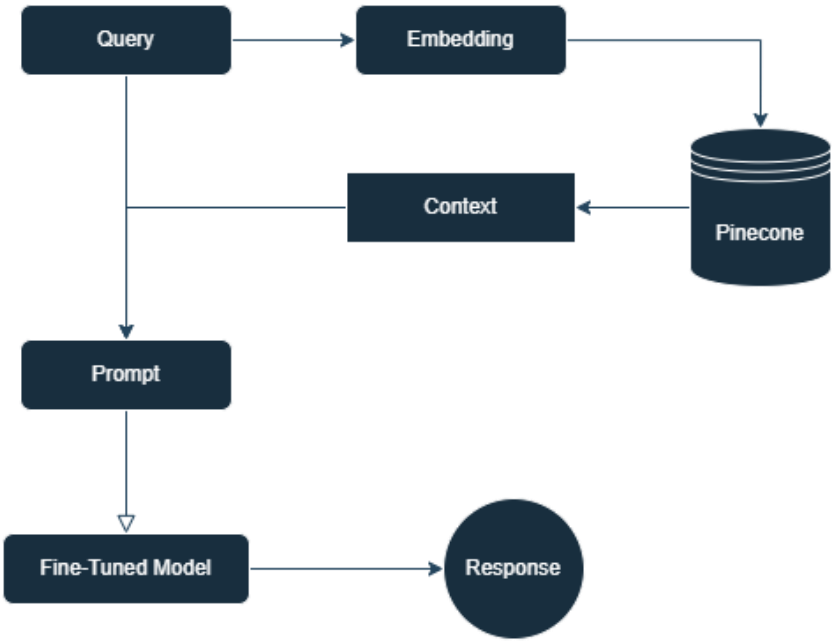
---

2. Vector Database Setup (Pinecone)

Next, I focused on setting up a vector index in **Pinecone**, which serves as the backbone of our RAG system. My contributions included:

1) Writing robust code to upsert vector embeddings along with their associated metadata into the Pinecone index. This step was critical for ensuring that our embeddings could be effectively stored and retrieved during the query process.
2) Developing and testing the functionality to query the vector index efficiently, enabling smooth integration with downstream components of our system.

By providing these scripts and creating a preprocessor for document preparation, I laid the foundation for our RAG system. This scaffolding enabled Tanmay and Raghav to populate the vector index with data and perform queries to retrieve relevant information.



RAG Architecture

The above architecture diagram explains working of our RAG system.

1. User gives a query and we create embedding for that query
2. Using that embedding we search the vector index for relevant context and return top 5 relevant vectors
3. We then append this relevant information with query in a single prompt.
4. We finally give this prompt to LM for generation.

---

3. Fine-Tuning

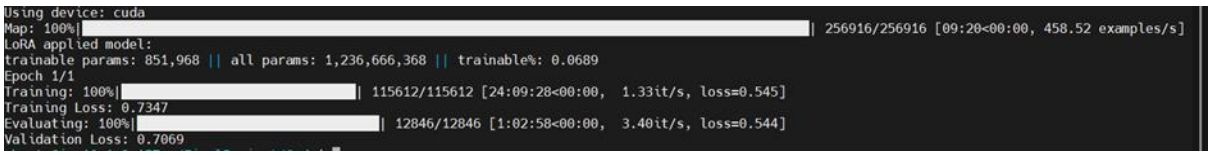
After completing the RAG setup, I transitioned to working on fine-tuning the model in collaboration with Sajan. Initially, we explored multiple LLM options and eventually decided on the **Llama 3.2 (1 billion parameters)** model. Although promising, our initial attempts at fine-tuning faced several challenges, such as inadequate performance and high resource consumption.

Through iterative trials and experimentation, Sajan identified that **LoRA (Low-Rank Adaptation)** offered the most effective approach for our specific use case. He finalized the fine-tuning script based on this technique, which significantly improved the model's performance while optimizing resource usage.

```
output_text = (f"<|begin_of_text|>"
               f"<|start_header_id|>"
               f"system"
               f"<|end_header_id|>"
               f"Your are a proficient doctor specializing in Gynaecology."
               f"<|eot_id|>"
               f"<|start_header_id|>"
               f"description"
               f"<|end_header_id|>"
               f"{example['Description']}"
               f"<|eot_id|>"
               f"<|start_header_id|>"
               f"patient"
               f"<|end_header_id|>"
               f"{example['Patient']}"
               f"<|eot_id|>"
               f"<|start_header_id|>"
               f"doctor"
               f"<|end_header_id|>"
               f"{example["Doctor"]}"
               f"<|eot_id|>"
               f"<|end_of_text|>")
```

Code Snippet for instruct-prompt for fine tuning

During this phase, I observed inefficiencies in the input-output handling of the model, where inputs and outputs were directly passed without any refinement. After conducting research, I discovered the concept of **Instruct Fine-Tuning**, which focuses on aligning the model to task-specific instructions. By integrating this approach into our subsequent fine-tuning iterations, we achieved better task alignment and improved response accuracy.



Snapshot of Model Training Process

I also suggested training models on a smaller subset of data to reduce training time as it was taking around 24 hrs of time for 1 epoch using LoRa when we were just training around a 10^6 (million) parameters. We ultimately decided to train our final models on 0.2% of the dataset, which surprisingly outperformed the models trained on the full dataset.

4. Inference Code

```
# Remove the input text from the generated output
generated_tokens = outputs[0]
input_length = inputs["input_ids"].shape[1]
output_final = generated_tokens[input_length:]
response = tokenizer.decode(output_final, skip_special_tokens=True)
```

Code to clean outputs

During the inference phase, I noticed that the model's output vectors often included unnecessary input tokens, which detracted from the quality of the results. To address this, I implemented a cleaning process to remove all input tokens from the output, ensuring a more concise and relevant response.

```
    prompt = (
    "<System Prompt>\n"
    "You are an expert medical doctor of the patient.\n"
    "Read the patient's query and provide a clear, concise, and medically sound response.\n\n"
    "Your answer should include:\n"
    "- A diagnosis\n"
    "- A recommended treatment plan or next steps\n\n"
    "Do not repeat the patient's question. Avoid unnecessary disclaimers.\n"
    "Keep your answer focused, authoritative, and helpful.\n"
    "</System Prompt>\n\n"
    "Query: {question}\n\n"
    "Your Response:".format(question=question)
)
```

Final prompt after performing prompt engineering

I also engaged in **prompt engineering**, experimenting with various techniques to enhance the model's output. For example, I tested the use of special tokens within prompts to guide the model's response generation. After extensive testing, we finalized a prompt template that consistently delivered the best results.

Furthermore, I conducted **hyperparameter tuning** by adjusting key parameters such as max_length and temperature during the generation process. This optimization significantly improved the fluency, coherence, and relevance of the model's responses, ensuring they met the project's requirements.

---

5. Streamlit Application

Finally, I contributed to the development of the **Streamlit** application, which serves as the user interface for our project. I wrote the initial script (app.py), providing the basic chatbot functionality and a user-friendly UI. This script established the foundation for the final version of the Streamlit app, which Tanmay and others later refined and enhanced.

The application's initial version included essential features such as input handling, response generation, and a simple interface. This early version allowed the team to test and iterate on the functionality while incorporating user feedback. The enhancements made by Tanmay and others built upon this structure, resulting in a polished and fully functional app that met our project's goals.

6. Code Maintenance

I also worked on the code maintenance part of the project. My responsibilities included creating a clear code structure and ensuring everything worked seamlessly. I set up branches based on features like data processing, RAG, and fine-tuning, allowing team members to make contributions in a more organized way. Members also created their own separate branches for specific tasks. In the end, I merged all the code from the feature branches into the main branch, making sure everything was functional and integrated correctly.

IV. Summary and Conclusion

To sum up, I worked on many important parts of the project, like setting up embeddings, connecting to a vector database, fine-tuning the model, improving how it gives answers, and building the Streamlit app. I created the basic setup for the RAG system so the team could easily add and use data from the

vector index. By working on fine-tuning with smaller amounts of data and trying new methods like instruct fine-tuning, I helped make the model perform better and train faster. I also cleaned up the outputs during inference and tried different ways to make the model's responses more accurate. Finally, I built the first version of the Streamlit app, giving the team a starting point to create the final app.

VI. Percentage Code: 66.6%

VII. References

1) **UAE-Large-V1**
2) **Pinecone**
3) **Recursive Splitter**
4) **Torch Chat Template**
5) **Instruction-Tuning**
6) **Streamlit**
