Individual Report

Name: Raghav Agarwal

GWID: G32909283

DoctorGPT: An AI-Based Medical Q&A Chatbot

1. Problem Selection

The objective of **DoctorGPT** is to develop an AI-driven chatbot designed specifically for answering medical-related questions. This chatbot aims to provide preliminary healthcare guidance, which can be particularly useful in:

- Regions with limited access to healthcare services.
- Situations where long wait times delay medical consultations.

By leveraging AI to handle common medical inquiries, **DoctorGPT** can enhance healthcare accessibility, provide immediate information, and empower users with reliable, preliminary medical advice before they consult a healthcare professional.

2. Dataset

For this project, we will utilize the **"AI Medical Chatbot"** dataset available on Hugging Face. This dataset consists of:

- **Description**: Context or scenario of the medical inquiry.
- **Patient Question**: The questions asked by patients.
- **Doctor Answer**: Responses provided by medical professionals.

This dataset offers realistic doctor-patient dialogues, providing a rich and diverse set of training data to develop and fine-tune our chatbot.

Project Timeline

- *Week 1*: Data acquisition and exploration, topic modeling on the description column.

- *Week 2*: Wikipedia data scraping and building an initial encoder model.

- *Week 3*: Fine-tuning the encoder model with Wikipedia data.

- *Week 4*: Model training on Q&A format with the Hugging Face dataset.

- *Week 5*: Performance evaluation, metric analysis, and final tuning.

- *Week 6*: Documentation, project review, and final adjustments for presentation.

Shared Work

Raghav and Tanmay initiated the process by conducting topic modeling to identify a specific focus area for model specialization. They further explored and gathered relevant data sources for this topic.

Subsequently, the fine-tuning phase was undertaken by Parv, Raghav, and me, refining the model using the selected dataset.

Development of the Retrieval-Augmented Generation (RAG) system was done by Tanmay, Parv, and Raghav, while the user interface was designed and implemented on Streamlit by Parv and Tanmay.

Individual Contributions by Raghav Agarwal

- Understanding Topics of the present data in frame.
- Extracting relevant topics for RAG Vector-Database
- Creating Pinecone vector database.
- Fine tuning of Data using partitions.

Description of work in detail

Understanding Topics of the present data in Frame.
(Code/data/topic_modelling/topic_modelling.py)

This script processes a medical chatbot dataset, performs text preprocessing, and applies Latent Semantic Analysis (LSA) for topic modeling on three columns: **Description**, **Doctor**, and **Patient**.

1. Imports and Initialization

The script imports various libraries for text processing, topic modeling, and visualization:

- **Pandas** for handling tabular data.
- **Regex (re)** for pattern-based text cleaning.
- **TF-IDF Vectorizer** for converting text data into numerical feature vectors based on term frequency.
- **TruncatedSVD** for dimensionality reduction to identify latent topics.
- **Matplotlib** for plotting frequency distributions and topic importance.
- **NLTK** for text preprocessing tasks like lemmatization and stopword removal.
- **Numpy** for handling arrays and numerical operations.
- **Warnings** to suppress unnecessary warnings during execution.
- The **Hugging Face datasets library** for loading the medical chatbot dataset.

2. Dataset Loading

The dataset is loaded using Hugging Face's load_dataset function, specifically the "AI Medical Chatbot" dataset. The dataset is then converted into a DataFrame for easy manipulation. It contains three key columns:

- **Description**: Contextual information or scenario related to the medical inquiry.
- **Doctor**: Responses provided by medical professionals.
- **Patient**: Questions posed by patients.

3. Frequency Distribution Visualization

A function cleans and plots the frequency distribution of words in each of the three columns. This involves:

1. **Combining all text** from the column into a single string.
2. **Removing special characters** and keeping only alphanumeric text.
3. **Calculating word frequency** using NLTK's FreqDist.
4. **Plotting the 200 most common words** to visualize their distribution.

The frequency distribution helps identify which words are most frequently used in each column.

4. Filtering Frequent Words

Based on the frequency distributions, words that appear more frequently than specified thresholds are filtered out. These frequently occurring words are combined with standard English stopwords to create a **custom stopword list**. This helps in removing common, non-informative words during text preprocessing.

5. Text Preprocessing

The preprocessing function performs several tasks to clean the text data:

1. **Lowercasing** the text.
2. **Removing patterns** like initial question markers or trailing punctuation.
3. **Stripping special characters**, numbers, extra spaces, and URLs.
4. **Removing custom and standard stopwords** identified earlier.
5. **Lemmatizing** words to reduce them to their base form.

This function is applied to each of the three columns to prepare the text for analysis.

6. Latent Semantic Analysis (LSA) for Topic Modeling

The script applies LSA to identify latent topics in each of the three columns. Key steps include:

1. **TF-IDF Vectorization** to convert the cleaned text into numerical features.
2. **Truncated SVD** to reduce dimensions and extract 5 latent topics.

3. **Identifying the top 10 terms** for each topic based on their importance.
4. **Plotting the top terms** for each topic to visualize their significance.

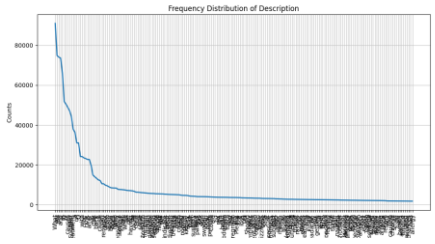
The LSA process is performed separately for the **Description**, **Doctor**, and **Patient** columns.

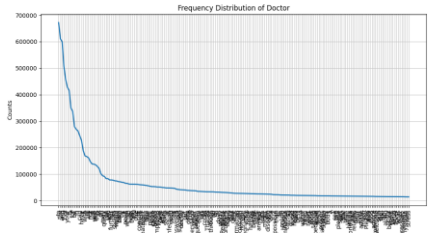
7. Exporting the Results

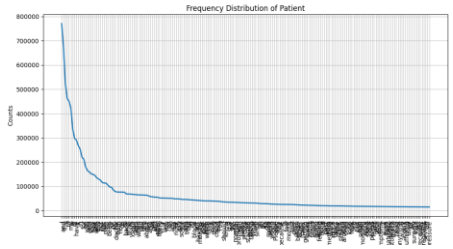
The identified topics and their top terms are saved into an Excel file named **lsa_topics.xlsx**. This file contains the topics for each column, making it easy to review and analyze the results.

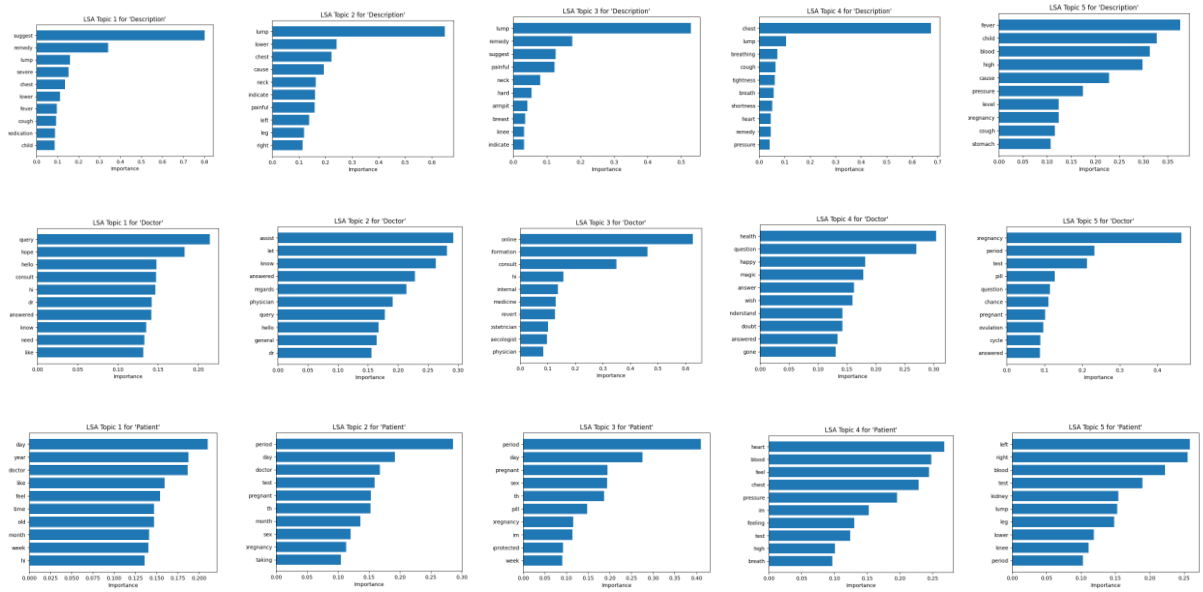
Summary of Workflow

1. **Load and clean** the medical chatbot dataset.
2. **Visualize word frequency distributions** for insights into common terms.
3. **Filter frequent words** to create a custom stopword list.
4. **Preprocess the text** through cleaning and lemmatization.
5. Apply **LSA topic modeling** to discover key themes in each column.









WikiScraping after finalizing Topic

(Code/Data/wikipedia/wiki_scrape.py)

Explanation of the Code

This script searches for Wikipedia articles related to a given query, retrieves the content, filters specific sections, and saves the important text to .txt files in a specified folder.

1. Imports and Initialization

The script imports necessary libraries:

- Pandas: For potential DataFrame operations (not explicitly used here).
- NLTK: For natural language processing tasks (not used in this code).
- String and Regex (re): For string processing.
- Numpy: For numerical operations (not used here).
- WikipediaAPI: For interacting with the Wikipedia API.
- LangChain's WikipediaLoader: For loading Wikipedia articles.
- Warnings: To suppress warnings.
- OS: For file system operations, such as creating directories and saving files.

The current working directory is printed to verify the execution environment.

2. Wikipedia Query and Document Loading

The query "pregnancy" is specified. The WikipediaLoader is used to fetch up to 50 Wikipedia articles related to this query:

- query specifies the search term.
- lang="en" ensures articles are retrieved in English.
- The articles are loaded into the documents variable.

The script then extracts the titles of the retrieved Wikipedia pages. Each title is appended to a titles list, ensuring no errors occur if metadata is missing.

3. Wikipedia API Initialization

A wikipediaapi.Wikipedia object (wiki_wiki) is created to interact with the Wikipedia API. The configuration includes:

- user_agent: Custom user agent for identification.
- language='en': Requests content in English.
- extract_format: Extracts plain text format.

4. Saving Wikipedia Pages to .txt Files

The function save_wikipedia_pages_to_txt saves the filtered content of each Wikipedia page to .txt files in a specified folder (../wiki_texts). Here's the process:

1. Create Directory: The specified folder is created if it doesn't exist.
2. Retrieve Page: For each title in titles, the corresponding Wikipedia page is fetched.
3. Filter Sections: Only sections not titled "History," "References," "See Also," or "External Links" are included.
4. Save Content: If filtered content exists:
    - The text is saved to a .txt file named after the Wikipedia title (underscores replace spaces).
    - The file is saved in the specified folder with UTF-8 encoding.
5. Feedback: The script prints messages indicating whether the content was saved or if no relevant content was found.

5. Execution

The function save_wikipedia_pages_to_txt(titles) is called to perform the saving operation for the list of Wikipedia titles retrieved.

Summary of Workflow

1. Load Wikipedia articles related to the query using WikipediaLoader.

2. Extract titles of the retrieved articles.
3. Fetch and filter content using wikipediaapi, excluding specific sections.
4. Save filtered content to .txt files in the ../wiki_texts directory.
5. Provide feedback on successful or unsuccessful operations.

Creating Pinecone vector database
(Code/rag/wiki_data_extractor.py)

This script processes a set of text files (previously saved Wikipedia content), prepares the data using a custom data processing pipeline, and stores the processed documents in a vector database (Pinecone) for future retrieval and similarity search.

1. Imports and Initialization
The script imports necessary libraries and tools:

- os: For interacting with the file system.
- pandas and numpy: For data manipulation (though not explicitly used here).
- tqdm: For displaying progress bars.
- Custom Modules:
    - preprocessor: Contains DataProcessor and Document classes for data processing.
    - vector_db: Contains the VectorDB class for interacting with the Pinecone vector database.
- dotenv: To load environment variables from a .env file.
  The .env file is expected to contain configuration parameters for Pinecone, such as API keys and index details.

2. Load Wikipedia Text Files
The script sets the path to the folder containing the Wikipedia .txt files and retrieves a list of all text files within that directory.

1. Get the absolute path of the ../data/wiki_texts directory.
2. List all .txt files in the directory.

3. Document Processing
For each .txt file in the directory:

1. Read the file content.
2. Create a Document object with:
    - page_content: The text content of the file.
    - metadata: The title of the document (derived from the file name).
3. Process the document using the DataProcessor class.
    - This likely involves tasks such as tokenization, text cleaning, or splitting into smaller chunks.
4. Store the processed documents in a list named processed_docs.

The progress bar provided by tqdm gives a visual indication of processing progress.

4. Pinecone Vector Database Configuration

The script loads Pinecone configuration details from environment variables:

- PINECONE_API_KEY: API key for authentication.
- PINECONE_ENV: Environment (region) for the Pinecone index.
- PINECONE_CLOUD: Cloud provider.
- PINECONE_INDEX_NAME: The name of the Pinecone index to use.
- DIMENSION: The dimension of the embedding vectors.
- METRIC: The similarity metric (e.g., cosine similarity).

A VectorDB instance (embedding_manager) is created using these parameters.

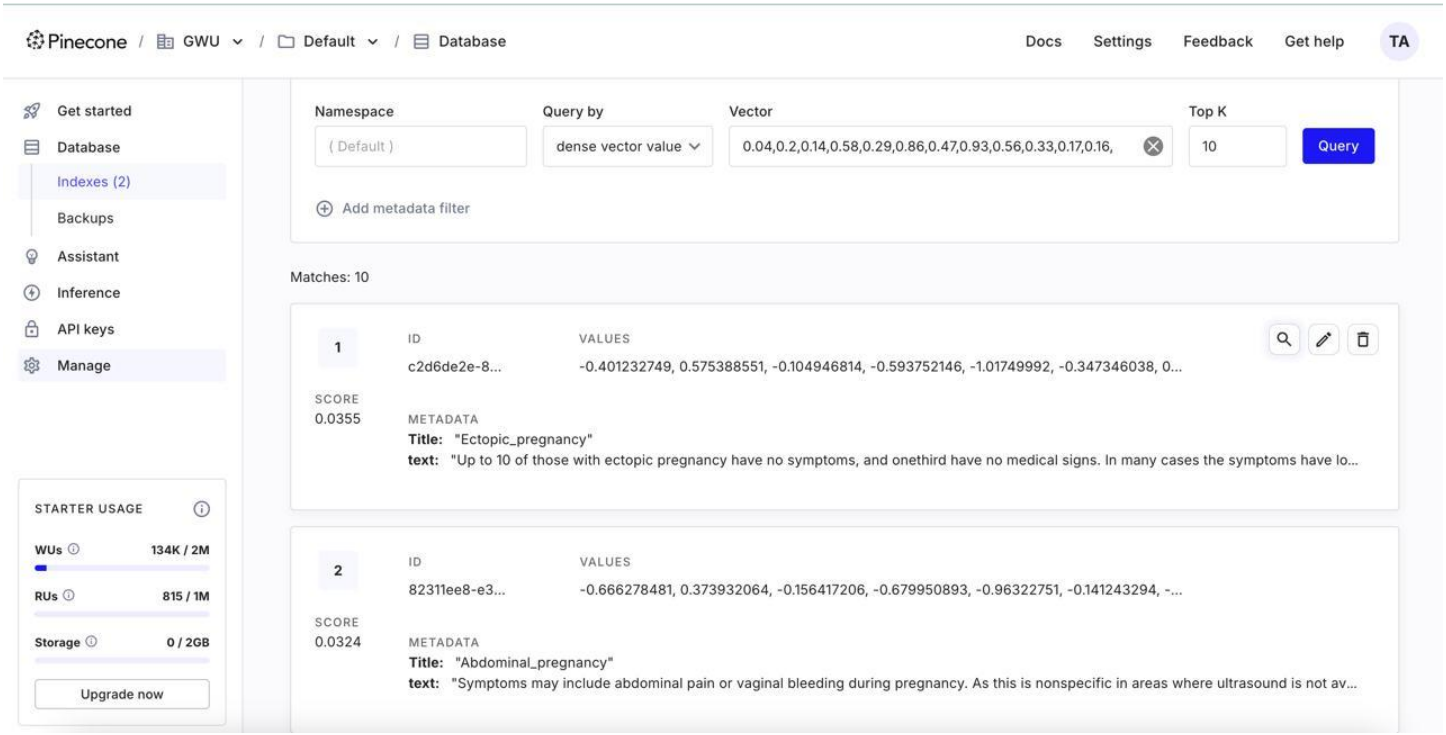
5. Store Processed Documents in Pinecone

The script prints the model and platform information used by the VectorDB instance. It then calls process_and_store_documents to:

1. Embed the processed documents using a model (likely an embedding model like OpenAI's or HuggingFace's).
2. Store the embeddings in the Pinecone index.

Summary of Workflow

1. Load Wikipedia text files from the specified directory.
2. Process the text documents to prepare them for embedding.
3. Configure Pinecone using environment variables.
4. Embed and store the processed documents in Pinecone for efficient similarity search and retrieval.



Fine tuning of Data using partitions.
(Code/finetune/finetune_Raghav.py)
This script fine-tunes a **LLaMA (Llama-3.2-1B)** language model using LoRA (Low-Rank Adaptation) on a medical chatbot dataset. The code performs data preprocessing, tokenization, training with gradient accumulation, evaluation, and saving the fine-tuned model.

1. Imports

- **PyTorch**: For model training and data handling.
- **Transformers**: For loading tokenizer and language models (LLaMA).
- **BitsAndBytesConfig**: For model quantization to reduce memory usage.
- **Datasets**: For loading and partitioning the dataset.
- **TQDM**: For progress bars.
- **PEFT**: For applying LoRA to the model.
- **huggingface_hub.login**: To authenticate with the Hugging Face Hub.
- **torch.nn.utils.rnn.pad_sequence**: For padding sequences to the same length.

2. Device Configuration

- The script checks for a CUDA-enabled GPU and sets the computation device accordingly.

3. Load Dataset and Partitioning

- **Dataset**: The "AI Medical Chatbot" dataset from Hugging Face (ruslanmv/ai-medical-chatbot).
- **Partitions**: The dataset is split into **250 partitions** to manage large datasets efficiently. Each partition is a subset of the training data.

4. Preprocessing and Tokenization Functions

1. **preprocess_data**:
    - Prepares input-output pairs where:
        - **Input**: Patient's question.
        - **Output**: Doctor's response.
2. **tokenize_data**:
    - Tokenizes the input and output texts.
    - Pads/truncates sequences to a maximum length of **512 tokens**.
    - Adds labels for training the language model.
3. **collate_fn**:
    - Custom function to pad sequences in a batch for:
        - input_ids
        - attention_mask
        - labels

5. Training and Evaluation Functions

1. **train_epoch**:
    - Runs one epoch of training.
    - Implements **gradient accumulation** to simulate a larger batch size.
    - **Gradient clipping** is applied to stabilize training.
2. **evaluate_model**:
    - Evaluates the model on the validation set.
    - Computes average loss for evaluation.
3. **save_model**:
    - Saves the model and tokenizer to a specified directory.

6. Login and Model Loading

- Logs into the Hugging Face Hub using an authentication token.
- Loads the **LLaMA-3.2-1B** model and tokenizer.
- Adds a padding token if not already present.
- Configures **8-bit quantization** using BitsAndBytesConfig to reduce memory usage.
- Applies **LoRA** for efficient fine-tuning.

LoRA Configuration:

- **Rank (r)**: 8
- **Alpha (lora_alpha)**: 32
- **Dropout**: 10%

7. Training Loop

For each partition:

1. **Empty CUDA Cache**: To manage GPU memory.
2. **Preprocess and Tokenize** the partition.

3. **Train-Test Split**: 90% for training, 10% for validation.
4. **Data Loaders**: For training and validation batches.
5. **Train for 3 Epochs**:
    - Train the model and evaluate after each epoch.
    - Save the fine-tuned model after each epoch.
6. **Save the Model** for the current partition.

Summary of Workflow

1. **Load and partition** the medical chatbot dataset.
2. **Preprocess and tokenize** the data.
3. Apply **LoRA** to the LLaMA model for efficient fine-tuning.
4. **Train and evaluate** the model on each partition with gradient accumulation.
5. **Save the fine-tuned model** after each epoch and partition.

Summary

The DoctorGPT project focuses on developing an AI-powered chatbot for answering medical-related inquiries. The primary goal is to provide immediate and reliable healthcare guidance in regions with limited access to medical services or where wait times are long. By leveraging the power of AI, DoctorGPT aims to bridge the gap in preliminary healthcare information, offering users a tool to access medical knowledge before consulting healthcare professionals.

Project Timeline and Work Distribution

- Timeline: The project was completed in six weeks, covering data acquisition, model development, fine-tuning, evaluation, and documentation.
- Raghav's Contributions:
    1. Topic Modeling: Identified key topics from the dataset to aid in the Retrieval-Augmented Generation (RAG) approach.
    2. Wikipedia Data Scraping: Scraped relevant Wikipedia pages to enhance the knowledge base for DoctorGPT.
    3. Vector Database Creation: Built a Pinecone vector database to store and retrieve relevant information efficiently.
    4. Data Fine-Tuning: Fine-tuned the LLaMA language model using the medical chatbot dataset with a partitioned training approach.

Key Technical Components

1. Topic Modeling

- Applied Latent Semantic Analysis (LSA) on the "Description," "Doctor," and "Patient" columns of the dataset to identify key themes.
- Preprocessed data by cleaning text, removing stopwords, and lemmatizing words.
- Results were exported to an Excel file for analysis.

2. Wikipedia Data Scraping
    - Fetched Wikipedia pages related to key topics.
    - Filtered and saved relevant sections, excluding sections like "History" and "References," to .txt files.
    - This data enriched the knowledge base for the chatbot.
3. Vector Database with Pinecone
    - Processed Wikipedia .txt files into embeddings.
    - Stored the embeddings in Pinecone to enable fast and accurate retrieval for the RAG system.
4. Model Fine-Tuning
    - Utilized the LLaMA-3.2-1B model with LoRA (Low-Rank Adaptation) for efficient fine-tuning.
    - Split the dataset into 250 partitions to manage large-scale training.
    - Implemented gradient accumulation to optimize memory usage during training.
    - Fine-tuned the model over 3 epochs per partition, evaluated performance, and saved the model periodically.

Conclusion

The DoctorGPT project successfully achieved its goal of developing an AI-based medical chatbot. By combining robust data preprocessing, topic modeling, and model fine-tuning techniques, the chatbot is capable of providing preliminary medical guidance efficiently. The project utilized cutting-edge methods like LoRA and quantized models to handle resource constraints effectively, ensuring scalability and performance.

Key takeaways include:

1. Data Diversity: Leveraging both the medical chatbot dataset and Wikipedia data enriched the model's knowledge base.
2. Efficiency: Using LoRA for fine-tuning allowed for efficient model adaptation without excessive computational costs.
3. Scalability: Partitioning the dataset facilitated the handling of large datasets in manageable chunks.

4. Practicality: The chatbot can provide immediate medical advice, enhancing healthcare accessibility.

This project demonstrates the potential of AI to address real-world challenges in healthcare accessibility and offers a solid foundation for further development and improvement of medical Q&A systems.

Line Code Percentage:

Total Lines of code: 414

Code from internet: 120

Modified lines of code: 40

Added Lines of code: 294

Percentage:

(120 – 40) / (120 + 294) -> 0.19

References:

UAE-Large-V1 Embedding Model: [Hugging Face]. https://huggingface.co/WhereIsAI/UAE-Large-V1

Vector Database: Pinecone Documentation. https://docs.pinecone.io/guides/get-started/overview