SRH HOCHSCHULE HEIDELBERG

MASTERS THESIS

AI-Driven Medical Chatbot for Medical Students and Patients: An LLM and Embedding-Based Approach

Author:

NATARAJU TUMAKURU SHREESHYLESHA

Matriculation Number: 11027742

Supervisors:

PROF. DR. CHANDNA SWATI

A thesis submitted in fulfilment of the requirements for the degree of
Master Of Science In Applied Data Science and Analytics

in the

School of Information, Media and Design

5th March, 2025

Declaration of Authorship

I, **Nataraju Tumakuru Shreeshylesha**, declare that this thesis titled, *"AI-Driven Medical Chatbot for Medical Students and Patients: An LLM and Embedding-Based Approach"* and the work presented in it is my own. I confirm that this work submitted for assessment is expressed in my own words and is my own. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are appropriately acknowledged at any point of their use. A list of the references employed is included.

- This work was done wholly for a research degree at this University.
- It has been clearly stated in this work if any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution.
- Where I have consulted the published work of others, credit is always given to the concerned work.
- Where I have quoted from the work provided by others, the source is always given unless it is entirely my work.
- I have acknowledged all main sources that supported me and helped during this work.

Signed:

Date:

Acknowledgements

I am deeply grateful for the support and guidance of numerous individuals who have contributed to the completion of this thesis, and I would like to express my heartfelt appreciation to all of them.

First and foremost, I would like to extend my sincere gratitude to my supervisor, Prof.Dr.Chandna Swati, for her invaluable guidance, expertise, and unwavering support throughout this research. Her mentorship has played a crucial role in shaping my ideas and refining my work.

I am also grateful to the faculty and staff of the School of Information, Media and Design at SRH Hochschule Heidelberg for their continuous encouragement and for providing an enriching environment for research and learning.

Special thanks to my colleagues and friends for their motivation, insightful discussions, and unwavering support. Their camaraderie has made this journey more fulfilling and enjoyable.

I am profoundly indebted to my family for their unconditional love, patience, and encouragement throughout this journey. Their unwavering belief in me has been my greatest source of strength and perseverance.

Lastly, I extend my gratitude to everyone who directly or indirectly contributed to the successful completion of this thesis. Your support and encouragement mean the world to me.

Abstract

Artificial intelligence (AI) is rapidly transforming the landscape of medical education and patient support. This thesis proposes an AI-driven medical chatbot system designed to serve two main user roles—medical students and patients—via distinct modes of operation. For medical students, the system provides an interactive knowledge platform with detailed explanations of complex topics, whereas for patients, it offers symptom analysis, healthcare provider recommendations, and even autonomous appointment booking through Agentic AI capabilities.

The architecture comprises several key components. First, textual data (such as PDFs of medical articles or learning materials) undergoes text extraction and chunking before being transformed into embeddings by a dedicated model. These embeddings are then indexed in a vector database, enabling efficient similarity-based retrieval. When a user submits a query, the system identifies and retrieves the most relevant text segments and subsequently augments the prompt for a Large Language Model (LLM), which can be tailored to specific roles: a learning AI assistant (e.g., Llama 2, 13B) for medical students and an agentic diagnostic AI (e.g., Mistral 13B) for patients.

In student mode, the chatbot focuses on explaining medical concepts and providing in-depth, evidence-based knowledge. In patient mode, the chatbot employs symptom analysis to generate preliminary diagnostic suggestions, seamlessly directing users to the HealthCare Provider Recommendation System. Harnessing its Agentic AI feature, the system can autonomously schedule appointments if necessary, reducing administrative overhead and expediting patient care. The entire process is orchestrated through a secure login mechanism, ensuring that only authorized users can access specific functionalities suited to their roles.

Experimental evaluations draw upon real-world medical texts and feedback from both medical students and healthcare practitioners. Early results indicate that this approach effectively balances accurate medical guidance, educational depth, and practical utility in assisting patients. Future enhancements include expanding multi-language support, improving the LLM's medical domain expertise, and integrating real-time health data streams to strengthen diagnostic accuracy.

Keywords: AI-driven Medical Chatbot, Role-based LLM, Patient Symptom Analysis, Healthcare Provider Recommendation, Autonomous Appointment Booking, Embedding Model, Vector Database

Contents

List of Figures

List of Tables

Listings

Abbreviations

CTM	Correlated Topic Model
LDA	Latent Dirichlet Allocation
LSA	Latent Semantic Analysis
PLSA	Probabilistic Latent Semantic Analysis
NMF	Non-Negative Matrix Factorization
TF-IDF	Term Frequency-Inverse Document Frequency
SVD	Singular Value Decomposition
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
HDBSCAN	Hierarchical Density-Based Spatial Clustering of Applications with Noise
HAC	Hierarchical Agglomerative Clustering
BERT	Bidirectional Encoder Representations from Transformers
NLP	Natural Language Processing
BIRCH	Balanced Iterative Reducing and Clustering Hierarchies
BOW	Bag-of-words
GloVe	Global Vectors for Word Representation
ML	Machine Learning
SBERT	Sentence-Bidirectional Encoder Representations from Transformers
GBERT	German-Bidirectional Encoder Representations from Transformers
T5	Text-To-Text Transfer Transformer
RoBERTa	Robustly Optimized BERT
UMAP	Uniform Manifold Approximation and Projection
PCA	Principal Component Analysis
T-SNE	T-Distributed Stochastic Neighbor Embedding
LLM	Large Language Models
GPT	Generative Pre-trained Transformer
LLaMA	Large Language Model Meta AI
JSON	JavaScript Object Notation

Chapter 1. Introduction

1.1 Introduction

AI (Artificial Intelligence) has seamlessly integrated with modern medicine and medical education, with the potential to improve patient outcomes and successfully train the next generation of physicians. Traditionally, patient-oriented solutions have either been rule-based systems or static information portals, and generally missed the capacity to comprehend context comprehensively, to adjust in a fluid manner, and to execute driving operations without human supervision. Likewise, learning tools in medicine have largely been passive, forcing users into static content repositories or low-interactivity platforms that fail to capture the complexity of medical concepts. This thesis addresses these limitations by proposing a role-based **AI-driven medical chatbot** capable of serving two distinct user groups:

- **Medical Students:** Learners find themselves in an interactive learning atmosphere, which allows students to ask about medical topics and get real-time, evidence-based answers.
- **Patients:** A simple interface for automatic appointment booking, healthcare provider suggestions, and symptom evaluation.

The system's architecture harmonizes several advanced technologies:

1. **Data Ingestion and Chunking:** Relevant medical texts (e.g., PDFs, articles, or curated training materials) are split into smaller, semantically consistent segments for informative retrieval and information reduction.

2. **Embedding and Vector Storage:** Embed every item under a model, say an LLM-based encoder employing numerical embedding and vector storage. By use of a vector database indexing, these embeddings exhibit fast similarity. Underline quickly sections from the database most likely appropriate for a user's search.

3. **Retrieval-Augmented Generation (RAG):** The system searches the top-matching chunks from the vector database upon user query submission, hence augmenting the generation. The language model (LLM) then makes use of these acquired segments as background, therefore improving the validity and accuracy of its produced answers.

4. **Role-based Large Language Models:** The chatbot distinguishes between "patient mode" (giving symptom-based guidance and possible next actions) and "student mode," which provides comprehensive medical knowledge and conceptual explanations. This role-specific strategy guarantees patients have easily available assistance and in-depth answers for pupils.

5. **Agentic AI for Appointment Booking:** The system combines an agentic component that enables autonomous scheduling or rescheduling of appointments. This function simplifies patient travels and lowers administrative overhead for physicians, therefore maybe improving access to care.

From profound academic support for medical students to pragmatic, real-world patient treatments, the suggested chatbot therefore covers a wide spectrum of capabilities. The architecture addresses current gaps in both user groups' experiences by combining embedding-based retrieval, role-aware LLMs, and autonomous scheduling: medical students acquire a dynamic tutor that can recall and contextualize complex material, while patients receive an accessible, on-demand assistant that not only provides initial information but also handles logistical tasks like finding the right specialist and booking an appointment.

Furthermore, ensuring a robust, user-centric design requires addressing core challenges. These include:

- **Data Quality and Scope:** The medical knowledge base must be reliable, up-to-date, and appropriately chunked to avoid misinformation.

- **Model Accuracy and Safety:** Large language models can exhibit “hallucination” or produce erroneous conclusions; thus, retrieval of verifiable source chunks is critical.

- **Ethical and Regulatory Considerations:** Especially with managing personal health information and schedule details, patient data protection, informed permission, and adherence to healthcare standards remain first priorities.

- **User Experience Across Roles:** The interface has to be able to easily differentiate between student and patient use, give students advanced conceptual detail and reduce medical language for patients.

This thesis assesses the performance of the suggested chat-bot architecture in fulfilling these objectives by means of methodical experimentation involving real-time input from both patients and healthcare practitioners. The next chapters cover the fundamental technologies, design approaches, system assessment criteria, and future developments that together define an advanced, role-based AI solution bridging medical education and patient-centric healthcare.

1.2 Motivation

Modern healthcare deals with a constant scarcity of medical experts as well as rising patient expectations at once. On one side, patients may need quick attention and easy scheduling systems to lower missed diagnosis, improve therapy compliance, and ease anxiety. To remain current, medical students and early-careers doctors must negotiate the complexity of specialized knowledge-based, hands-on practice and always changing policies. Static textbooks and passive lectures are among the conventional teaching strategies that might not be sufficient for equipping students to manage challenging, real-time situations.

Concurrently, traditional patient-facing instruments as rule-based triage systems or simple symptom checks seldom offer enough background or tailored advice. They also lack direct connection with hospital resources for appointment scheduling. Patients could thus be unsure about the severity of their symptoms or spend significant time researching several, dubious web sites.

These challenges point toward a critical need for a comprehensive solution that offers:

- **On-demand access to credible information:** This translates for patients into effective triage and clear symptom interpretation. It means interactive learning tools for pupils that replicate reasonable, sophisticated searches.

- **Reduced administrative barriers:** Automating chores like appointment scheduling guarantees that patients get timely treatment and frees the time of health professionals.

- **Seamless retrieval of validated data:** Whether it's localised healthcare provider information or current medical guidelines, the system has to combine pertinent data and provide it in context.

- **Adaptability for different user roles:** Advanced question-answering for students is somewhat different from simple patient scheduling and symptom analysis. One chatbot has to be able to manage these several modes fluidly.

Medical education and patient-centric services may be united on a single platform by using sophisticated embedding-based retrieval systems, agentic artificial intelligence capable of autonomous interaction with scheduling APIs, and large language models (LLMs). This coherent strategy provides quick advantages:

1. **Enhanced Learning:** At every degree of complexity, students may ask questions of the system and get thorough, evidence-based answers and case studies, therefore bridging the gap between theoretical knowledge and practical experience.

2. **Better Patient Outcomes:** Clear symptom expression and early appointment scheduling serve to minimize treatment delays, therefore lowering the possibility of issues and so decreasing patient stress.

3. **Administrative Efficiency:** Automated booking solutions simplify staff effort and reduce the chance of scheduling errors or gaps, therefore helping healthcare organizations to reduce their over-heads.

Developing a dual-mode medical chatbot is ultimately driven by transcending the constraints of single-purpose teaching platforms or basic symptom checks. Rather, an integrated AI-driven framework is ready to provide customized user experiences, guarantee flawless data flow, and create a transforming learning environment for medical students as well as better healthcare accessible for patients.

1.3 Problem Statement and Challenges

Two issues characterize modern healthcare: patients need consistent advice when confronted with contradicting symptoms; medical students need basic access to updated, accurate material to increase their core knowledge and be ready for clinical practice. Often lacking the timeliness, complexity, and contextual clarity required for complicated medical conditions are current solutions such as basic symptom checklists or static web-based systems. Moreover, these stand-alone technologies hardly provide for quick scheduling or referral, therefore leaving patients wondering about their next course of action on their medical path.

Regarding medical students particularly, the challenge is in Five Short Notes Names. Negotiating vast numbers of research publications, medical textbooks, and clinical case reports may be difficult, especially when doing so in real time or when referring to other fields (pharmacy and pathology, for example). Conventional e-learning systems lack effective Q and A systems suitable to user questions and hardly connect academic information with practical, scenario-based learning.

Timeliness and Accuracy: The pace of development of medical technology is so rapid that even conventional resources find it impossible to survive. Students take the risk of selecting obsolete techniques without fast, modern resources.

With respect to the patients, the struggles are equally important: people can sometimes struggle with knowing whether their symptoms warrant routine care or a trip to an urgent treatment center, which can lead to anxiety and potentially a delay in treatment.

Obstacles blocking healthcare access: Usually taking multiple phone calls or online searches, scheduling a specialist or follow-up visit slows patients from getting appropriate treatment especially those confused about the type of expert to see.

Unverified or contradicting information: Patients might be driven to self-diagnose via dubious internet sources as standard symptom-checking websites could be insufficient or not tailored.

Thus, the main challenge is designing a single, synthetic intelligence-powered system that satisfies both sets of needs:

1. A comprehensive teaching tool for medical students responding in-depth, contextually sensitively to specific questions derived from properly chosen clinical and scholarly sources. It allays any concerns, rapidly links patients to pertinent medical experts, and offers initial direction based on a patient-facing symptom analysis and triage mechanism.
2. Agentic appointment management helps to reduce administrative load and fill the information gap between pragmatic healthcare measures by enabling the system to autonomously organize or postpone patient appointments.

These underlines the requirement of including Large Language Models (LLMs), embedding-based data retrieval, and user role-aware interactions. One such strategy is: Abbreviations 6 covers the academic demands needed for medical school as well as the pragmatic, immediate concerns patients have when presented with health issues. The next parts will address how this integrated chatbot idea may provide medical students and patients accurate, ethically acceptable, contextually relevant advice.

1.4 Research Questions

Modern healthcare faces two major challenges. First, patients often receive inconsistent advice when confronted with ambiguous symptoms, which can lead to anxiety and delays in seeking proper care. Second, medical students frequently struggle to access the most current and accurate information needed to build their knowledge, especially when they have to sift through vast amounts of literature and clinical reports. In addition, existing systems rarely integrate practical features such as automated appointment scheduling or provider recommendations, leaving both patients and healthcare professionals with fragmented support.

This thesis aims to address these challenges by exploring the development of an integrated medical assistant tool that serves both patients and medical students. In particular, the study is guided by the following questions:

1. **Supporting Medical Education:** How can a system be built to provide medical students with timely, thorough, context-rich responses from current clinical literature thereby strengthening their basis for practice?

2. **Improving Patient Guidance:** How can the tool clearly advise whether a routine consultation or urgent care is suitable and fairly evaluate patient-reported symptoms?

3. **Healthcare Provider Recommendations:** Based on a patient's particular symptoms and location, how may the system efficiently suggest surrounding healthcare providers?

4. **Autonomous Appointment Management:** How can the technology be turned on to automatically arrange and control patient visits, therefore lowering administrative load and closing the distance between diagnosis and treatment?

Together, these questions aim to develop a unified solution that not only enhances the educational experience for future healthcare professionals but also offers practical, immediate support for patients.

1.5 Thesis Structure

This thesis is organized into six main chapters, as outlined below:

1. **Introduction:** This chapter defines the issue statement, introduces the topic, clarifies the driving force of the effort, and lists the research questions thereby setting the scene. It also describes the difficulties contemporary healthcare encounters with regard to medical education and patient treatment.

2. **State of the Art:** Reviewed in this chapter are present literature and accepted methods pertinent to the project. Topics cover traditional approaches and their constraints as well as the usage of Large Language Models and embedding techniques in healthcare. The backdrop this study offers helps one to grasp the setting of the suggested remedy.

3. **Design and Methodology:** This chapter describes the general system architecture and the techniques applied to build the medical chatbot. It addresses preprocessing and data collecting; it integrates language models with embedding-based data retrieval; it clarifies the agentic aspect of autonomous appointment management.

4. **Implementation:** The technological features of the project are then under discussion. The chapter details the tools, programming models, and procedures followed in system development. It also clarifies the way several modules were carried out and combined.

5. **Evaluation:** This chapter offers qualitative as well as numerical evaluations of the system. To show the success of the suggested strategy, it comprises performance criteria, user testing comments, and analogies with current solutions.

6. **Conclusion and Future Work:** The last chapter reviews the main conclusions of the studies, emphasizes the thesis's contributions, and addresses potential areas for next development.

Additional sections, such as the Appendices and the Bibliography, provide supplementary material, including source code, detailed experimental results, and a comprehensive list of references.

Chapter 2. State-of-the-Art

2.1 Introduction

Modern artificial intelligence, enormous data availability, and changing clinical demands taken together have revolutionized healthcare and medical education. Large language models (LLMs) and advanced embedding methods have become transforming technologies allowing dynamic patient involvement and inter-active educational support in recent years. These technologies today support conversational systems able to provide contextually complicated replies and manage challenging, unstructured clinical data. Covering historical advances, technical breakthroughs, distributed data structures, and assessment techniques, this chapter offers an all-inclusive survey of the state-of- the-art methods in the field. We build our dual-mode architecture meant to serve medical students and patients by combining ideas from two key studies—Paper 1 and Paper 2.

We arrange our review into various areas. Section 2.2 reviews the literature and traces the evolution of LLMs in healthcare; Section 2.3 explores the methodologies, including retrieval-augmentated generation, advanced embedding, dimensionality reduction, and clustering techniques; Section 2.4 addresses distributed architectures and privacy-enhancing technologies; Section 2.5 outlines evaluation strategies and benchmarks; and Section 2.6 discusses the system architectures that support these functionalities. Section 2.7, which looks at the difficulties and next avenues of study, closes us.

2.2 Literature Review

2.2.1 Evolving Role of LLMs in Healthcare and Medical Education

Early healthcare information systems used to be built on rigid databases with limited adaptation and rule-based expert systems. Data driven techniques took front stage when statistical language models and then deep learning approaches emerged. By allowing models like BERT and GPT to detect long-range relationships and contextual subtleties, transformer architectures introduced in 2017 transformed natural language processing (NLP). This development opened the path for models capable of producing very fluid and context-sensitive answers.

Recent innovations best shown by GPT-4 and open-source models such as Meta's LLaMa series have opened fresh opportunities for clinical decision assistance and medical education. Paper 2 shows how remarkably fluidly LLMs may now create teaching materials and patient-tailored discourse. To guarantee factual correctness, both publications do point out that such systems need thorough domain-specific adaption and strong retrieval methods.

2.2.2 Decentralized Architectures for Data Privacy

The major privacy issues present in conventional, centralized healthcare systems are discussed in this part together with how distributed solutions might help. Data breaches and illegal access expose centralized systems, therefore raising ethical and legal questions. By keeping their own sensitive medical data in a Personal Data Store (PDS), distributed architectures let every patient to keep control over it.

Processing and filtering data locally often via de-identification and edge-processing mechanisms only non-sensitive, anonymized information is communicated externally (for example, with huge language models used in chatbot systems). This approach guarantees compliance with strict rules including GDPR and HIPAA in addition to lowering the possibility of privacy invasions. Furthermore, these designs improve general data security by eliminating single points of failure and spreading data management, thereby building more user confidence.

2.2.3 Conversational AI for Patient Engagement and Education

Recent research has demonstrated the transformative potential of conversational AI systems in healthcare. Paper 2 outlines a multi-layered conversational framework where patient inputs are first processed by specialized modules—ranging from ad-hoc parsers for vital signs to intent recognition systems (e.g., Wit.AI)—before being passed to an LLM for free-form dialogue. This tiered approach is designed to protect sensitive clinical data while enabling rich, empathetic interactions.

For medical students, similar systems provide interactive tutoring that offers in-depth, evidence-based explanations and simulates realistic clinical scenarios. Tailoring responses based on user roles significantly enhances the system's utility, meeting the divergent needs of patients and students.

Both Paper 1 and Paper 2 highlight the use of retrieval-augmented generation (RAG) as a means to ground the generative process in verifiable sources. By indexing trusted medical texts in a vector database and retrieving contextually relevant passages, these approaches improve both accuracy and reliability in clinical applications.

2.3 Methodologies

The approaches followed in this part to create and assess AI based chatbot systems in the medical field. The method combines two complimentary points of view: one on data privacy and decentralization and the other on using large language models (LLMs) to improve patient involvement.

2.3.1 Decentralized Data Management for Enhanced Privacy

Building on current developments in distributed architectures, the suggested system uses a design whereby every patient has a Personal Data Store (PDS). This method guarantees that sensitive medical data is kept dispersed instead of in one, centralized repository, therefore reducing data breach and unauthorized access concerns. The essential actions consist in:

- **Data Ingestion and Local Processing:** Medical texts, vital sign measurements, and other clinical data are ingested and split into semantically consistent segments. Sensitive information is filtered and de-identified at the edge, ensuring that only non-sensitive, anonymized data is transmitted for further processing.

- **Personal Data Manager:** Each PDS is managed by a dedicated component that handles read/write operations and basic data aggregation (e.g., computing average values over specified time periods). This manager also enforces user control over access permissions, allowing patients to authorize healthcare professionals to view or modify their data.

- **Access Control Mechanisms:** Although not necessarily relying on blockchain in our implementation, the system uses a robust, smart-contract-based access control list to log and manage permissions transparently. This ensures compliance with stringent data protection regulations such as GDPR and HIPAA.

2.3.2 Leveraging Large Language Models for Patient Engagement

To improve patient engagement and support self-management, the system integrates advanced LLMs for natural language understanding and generation. The methodology here is structured around a multi-layered natural language processing pipeline:

- **Multi-tier NLP Pipeline:**
  - *Ad-hoc Interpretation:* Custom heuristics catered for the clinical setting help to first parse user inputs (e.g., vital sign data).
  - *Intent Recognition with Wit.AI:* Inputs not addressed by the ad-hoc parser are passed to Wit.AI for domain-specific intent identification, in which the system is trained on utterances connected to healthcare.
  - *Fallback via LLMs:* For searches that remain unsolved or call for a more conversational approach, a high-capacity LLM (such as GPT-4 or an open-source version) is called upon to provide sympathetic, context-aware re-responses.
- **Case Study Implementations:** Several case cases illustrating the LLM-based approach show:
  - Examining conversations on mental health to spot risk factors and language trends.
  - Creating customized chatbots for senior cognitive engagement.
  - Pairwise assessment frame-works help to summarize medical discussions.

  - Creating a patient interaction solution driven by artificial intelligence that connects with healthcare processes for automatic appointment scheduling

- **Evaluation and Ethical Considerations:** Using both quantitative measures such as victory rates from paired model comparisons and qualitative assessments by professional evaluators a strong evaluation methodology is applied. Throughout the development process, ethical issues like data privacy, bias, openness, and regulatory compliance are taken under discussion.

2.3.3 Integration and Synergy

Combining distributed data management with sophisticated conversational artificial intelligence helps the entire system design to fulfill two primary goals:

1. **Enhanced Privacy and Trust:** Patients have ownership of their own data, therefore guaranteeing compliant, safe, and open data handling.

2. **Improved Patient Engagement:** By use of LLMs, the system may provide dynamic, sympathetic replies, present individualized information, and offer timely advice depending on real-time patient contacts.

This combined approach not only solves the technical issues of privacy protection and safe data handling but also makes use of LLM transforming power to support improved therapeutic results and patient empowerment.

2.4 System Architecture

Our system design harmonizes strong data privacy with cutting-edge conversational artificial intelligence to enable doctor supervision and patient self-management. Inspired by approaches from distributed data management and digital health chatbots, the architecture is arranged into the following main elements:

2.4.1 Decentralized Data Management

The system uses a distributed method to protect patient privacy and follow laws including GDPR and HIPAA:

- **Personal Data Store (PDS):** Every patient receives a PDS, a safe haven for their private medical information. This architecture reduces the chances connected to centralized data leaks.

- **Personal Data Manager:** A dedicated component manages data ingestion, de-identification, and aggregation (e.g., computing average vital sign measurements over time). This manager ensures that only non-sensitive, anonymized data is transmitted for further processing.

- **Access Control Mechanisms:** Robust, policy-driven controls are implemented to allow only authorized healthcare professionals to access or update patient data, thus reinforcing user sovereignty over personal information.

2.4.2 Conversational AI Engine

The core of the system is its conversational AI engine, which leverages large language models (LLMs) to engage with patients effectively:

- **Multi-Tier Natural Language Processing:**
  - *Ad-hoc Interpretation:* Custom heuristics catered for the clinical setting help to first parse user inputs (e.g., vital sign data).

  - *Intent Recognition with Wit.AI:* Inputs not addressed by the ad-hoc parser are passed to Wit.AI for domain-specific intent identification, in which the system is trained on utterances connected to healthcare.

  - *Fallback via LLMs:* For searches that remain unsolved or call for a more conversational approach, a high-capacity LLM (such as GPT-4 or an open-source version) is called upon to provide sympathetic, context-aware re-responses.

- **Personalized Interaction:** The AI engine tailors its responses based on patient profiles, ensuring that conversations are both clinically relevant and emotionally supportive.

2.4.3 Integration with Clinical Workflows

The architecture is designed to bridge patient interactions with clinician oversight:

- **Patient Interface:** Easily entered data, health monitoring, and real-time chatbot engagement are made possible by the patient interface accessible via mobile applications, SMS, or web platforms.

- **Clinician Dashboard:** To enable quick clinical interventions, healthcare providers use a web-based dashboard, summary reporting, and real-time patient data analysis.

2.4.4 Data Flow and Security

First data is gathered via patient inputs and handled locally within the PDS. After that, securely sent non-sensitive, de-identified data goes to the conversational artificial intelligence engine to generate responses. Strong encryption and access control policies are used all throughout the system to guarantee data integrity, confidentiality, and regulatory standard compliance.

Using the benefits of distributed data management and cutting-edge conversational artificial intelligence, this integrated architecture creates a safe, patient-centric platform that increases clinical decision support, engagement, and finally helps to provide improved health outcomes.

2.5 Challenges

Using a safe and efficient AI-driven healthcare chatbot system presents various technological, ethical, and operational difficulties across several spheres. These difficulties result from the distribution of sophisticated large language models (LLMs) as well as from the distributed data management architecture:

- **Data Privacy and Security:** First and most importantly is safeguarding private patient information. To stop unwanted access and breaches, distributed architectures call for strong encryption, efficient data de-identification, and exact access limits. Following laws like HIPAA and GDPR adds even another level of complication.

- **Scalability and Performance:** Maintaining low latency for real-time interactions is difficult when the system expands to serve a rising user population and increased data volume. Sustaining performance depends critically on effective processing, storage, and retrieval of high dimensional embeddings.

- **Integration Complexity:** Combining conversational artificial intelligence engine with distributed data management calls for careful system architecture. Perfecting data flow among Personal Data Stores (PDS), natural language processing modules, patient interfaces, and clinician dashboards calls for strong APIs and exact synchronizing.

- **Reliability and Accuracy of LLMs:** Although LLMs like as GPT-4 have strong natural language processing and generating ability, their outputs have to be therapeutically appropriate and contextually precise. Especially in high-stakes healthcare environments, it is imperative to address problems including factual errors, possible biases, and erratic responses.

- **Regulatory Compliance and Ethical Considerations:** The system has to negotiate moral questions and complicated legal systems. This covers guaranteeing data sovereignty, getting informed permission, keeping openness in AI-driven judgments, and assigning unambiguous responsibility for automated activities.

- **User Trust and Adoption:** Patients and healthcare professionals alike have to see the system as dependable, safe, and user-friendly if adoption is successful. Establishing this confidence not only solves technological problems but also efficiently presents the data protection policies and artificial intelligence decision-making procedures of the system.

2.6 Evaluation Strategies

2.6.1 Quantitative Metrics

A comprehensive evaluation employs multiple quantitative metrics:

- **Semantic Similarity Scores:** Cosine similarity measures the closeness between generated responses and gold-standard references.

- **Retrieval Metrics:** Metrics like Precision@k and Mean Reciprocal Rank (MRR) assess the effectiveness of the retrieval module.

- **Clustering Validity Indices:** Silhouette Score, Davies-Bouldin Index, and Calinski-Harabasz Index evaluate the coherence and separation of clusters.

- **Pairwise Comparison:** Evaluate user satisfaction and system performance over time to appraise the long-term effects.

2.6.2 Human-Centric Evaluations

Human evaluations are important, in addition to numerical metrics:

- **User Studies and Surveys:** Ask medical students, patients, and doctors about clarity, empathy, and clinical relevance.

- **Expert Panels:** Have medical experts verify the accuracy of the chatbot outputs and usefulness

- **Task-Based Evaluations:** Design informative clinical scenarios and training simulations for performance evaluation

- **Longitudinal Studies:** Monitor user satisfaction and system performance over time to evaluate long-term impact.

2.6.3 Comparative Benchmarking

Comparative studies are essential to contextualize performance:

- **Traditional vs. Modern Approaches:** Benchmark the LLM-based approach against conventional methods (e.g., LDA, NMF).

- **Cross-Model Evaluations:** Compare outputs from different LLMs (e.g., GPT-3.5, Llama2-70B, Mistral-7B) under identical conditions.

2.7 Future Directions

Building on the current system architecture and methodologies, several avenues for future work can further enhance both the technical performance and clinical utility of AI-driven healthcare chatbots:

- **Multimodal Data Integration:** Future systems might combine electronic health records (EHRs), sensor data from wearable devices, and medical imaging to provide a more complete picture of patient health. Using real-time physiological data would provide more accurate and tailored responses.

- **Enhanced Model Fine-Tuning:** LLMs must be kept constantly fine-tuned on specific medical corpora and clinical conversations. Particularly for important decision-making in healthcare, this involves domain-specific training to increase the factual correctness, contextual relevance, and empathy of created replies.

- **Improved Data Privacy Techniques:** Data privacy is still a major issue, so more study on advanced de-identification techniques, federated learning, and safe multi-party computing can assist to guarantee that private patient data stays encrypted without endangering system performance.

- **Scalability and Real-Time Performance:** Future research should concentrate on maximizing scalability of system components. Examined are methods including distributed computing, pruning, and model compression to keep real-time responsiveness as the user base and data volume increase.

- **User-Centric Design and Usability Studies:** It is important to have constant comments from patients as well as from doctors. Larger and more varied user groups might be part of future research to hone user interfaces, increase conversational dynamics, and raise general system usability and accessibility.

- **Integration with Broader Healthcare Ecosystems:** Increasing the system's compatibility with current clinical management systems and healthcare IoT devices would help to provide a more flawless integration into daily clinical procedures.

This covers standardizing communication channels and data formats for more general use.

- **Ethical and Regulatory Frameworks:** Further research is needed to create and improve ethical rules and regulatory requirements as artificial intelligence-driven products find increasing presence in clinical practice. Establishing clear assessment criteria and responsibility systems should be the main priorities of further studies to guarantee responsible application.

2.8 Summary

The state-of- the-art in LLM driven systems for healthcare and medical education is fully reviewed in this chapter. From early transformer models to contemporary systems providing improved fluency and contextual awareness, we tracked the development of LLMs. Along with an in-depth study of distributed architectures and privacy-enhancing technologies as described in Paper 1, further talks on retrieval-augmented generation, sophisticated embedding techniques, dimensionality reduction, and resilient clustering

Our suggested dual-mode system is built on the multi-layered system architecture integrating data intake, embedding generation, retrieval, and LLM-driven response generating. Our thesis tries to close the distance between clinical practice and medical education by tackling important issues such factual correctness, data privacy, computational efficiency, and assessment complexity.

Future avenues of study include improving model dependability, merging multimodal data sources, enlarging distributed architectures, and standardizing assessment techniques. Development of next generation AI-driven medical chatbots that enhance patient involvement and educational results depends on these initiatives.

Chapter 3. System Architecture

This chapter includes a comprehensive review of the design and methods used adopted for developing the AI-driven medical chatbot. The system is tailored to cater to two distinct cohorts: patients and students of medicine, approached from an educational standpoint and pragmatic support including symptom evaluation and independent management of appointments. The complete system flow and design are shown in Figure 3.1.

3.1 High-Level Architecture

The system is structured into several interconnected modules, as shown in Figure 3.1.

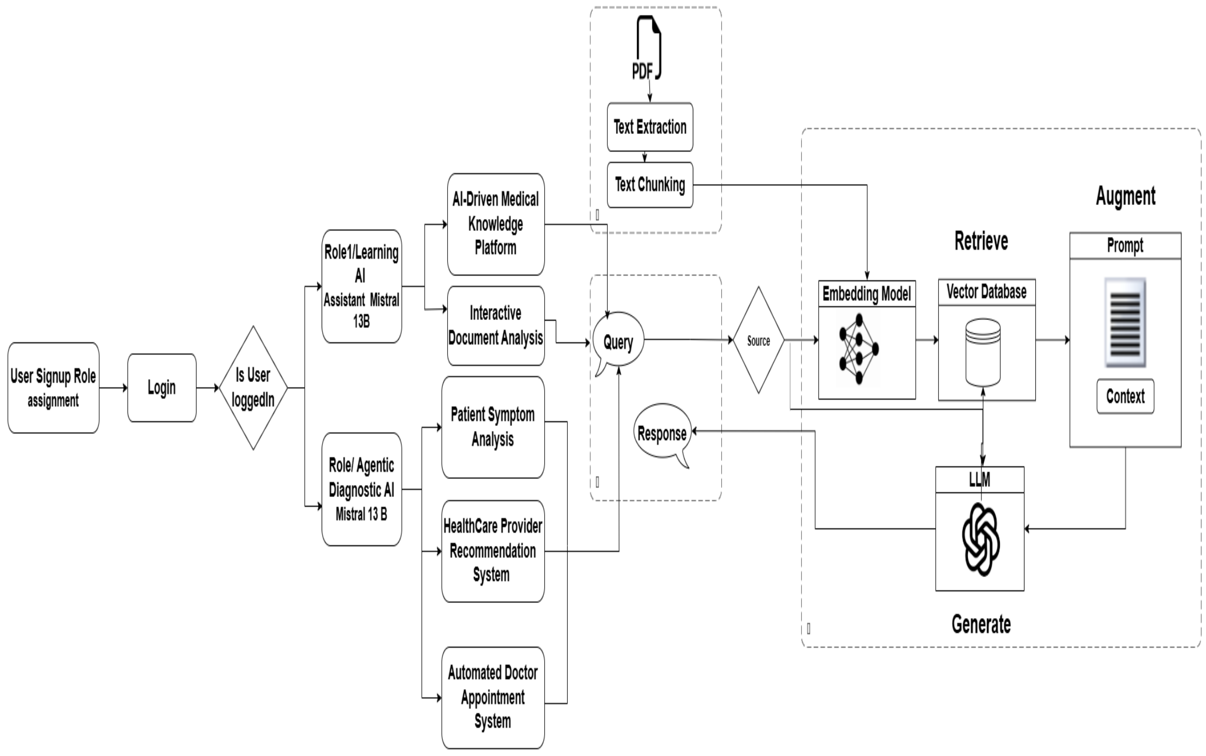
PDF
Text Extraction
Text Chunking
Augment
Retrieve
Prompt
AI-Driven Medical Knowledge Platform
Role1/Learning AI Assistant Mistral 13B
Interactive Document Analysis
Query
Source
Embedding Model
Vector Database
Context
User Signup Role assignment
Login
Is User loggedIn
Patient Symptom Analysis
Response
Role/ Agentic Diagnostic AI Mistral 13 B
LLM
HealthCare Provider Recommendation System
Automated Doctor Appointment System
Generate

FIGURE 3.1: Thesis implementation phases

1. **User Authentication and Role Identification:**
   - Upon user login, the system authenticates and decides the individual's role as either medical student or patient.
   - Role-based access controls the assignment of users to a particular dashboard and capabilities.

2. **Role-Based Interaction:**
   - *Medical Student Mode:* Medical Student Mode: Provides exhaustive study aids such as PDF files with scholarly question-and-answer sessions.
   - *Patient Mode:* Gives a symptom analysis with suggestions for location based doctor recommendation with agentic appointment system.

3. **PDF Processing and Text Chunking:**
   - Extracts text from uploaded PDFs and segments them into manageable chunks for embedding and retrieval.

4. **Embedding and Vector Storage:**
   - Uses `all-mpnet-base-v2` for semantic embeddings.
   - Effective similarity-based retrieval depends on stores of embeddings in a vector database (Pinecone).

5. **Retrieval-Augmented Generation (RAG):**
   - To get pertinent pieces, user inquiries are incorporated and compared against stored vectors.
   - Using the **Mistral 13B Quantized Model**, correct replies are then produced from the recovered context.

6. **LLM Response Generation:**
   - The Mistral 13B Quantized Model generates context-aware, role-specific answers.
   - Responses are tailored differently for educational depth (students) and practical advice (patients).

7. **Agentic Appointment Booking:**
    - For patients, the system autonomously schedules appointments by integrating healthcare provider data.

3.2 Mistral 13B Quantized Model

The **Mistral 13B Quantized Model** serves as the core LLM for generating responses. It is chosen for its balance between performance and computational efficiency.

3.2.1 Overview and Architecture

The Mistral 13B Quantized Model is a variant of the Mistral architecture known for:

- **Parameter Efficiency:** With 13 billion parameters, it delivers great performance while keeping reasonable resource needs owing to quantization.

- **Quantization Technique:** Uses Q3_K_M quantization to rapidly infer without appreciable loss of accuracy by lowering memory footprint.

- **Transformer Architecture:** Built on a decoder-only design with multi-head self-attention and feed-forward layers, tailored for generative workloads, transformer architecture is suited for

3.2.2 Contextual Response Generation

- The model responds using the context obtained from the RAG pipeline.
- Role-specific prompts guide the model in generating:
    - *Educational Explanations* For medical students with thorough references, educational justifications abound.
    - Easy, practical advice for patients guarantees clarity and reduces jargon.

3.2.3 Role-Specific Prompting

Different prompts are crafted based on user roles:

- **For Medical Students:**
  - Requests comprehensive explanations with references to academic sources.
  - Example: "Explain the mechanism of action of beta-blockers with clinical implications."

- **For Patients:**
  - Prompts the model to generate concise, easy-to-understand guidance.
  - Example: "What should I do if I experience chest pain?"

3.2.4 Additional Explanation of the Mistral 13B Quantized Model

This system has especially selected the Mistral 13B Quantized Model because of its outstanding balance between computational efficiency and high-quality response generating. Delivering real-time replies in a production setting depends on the model's memory footprint being much reduced by the quantization procedure, which also accelerates inference times. The model is well-suited for both the exact, practical counsel required by patients and the thorough instructional needs of medical students as it retains strong performance across many tasks despite its small size.

3.3 Embedding and Vector Database

3.3.1 Embedding Model: `all-mpnet-base-v2`

Text chunks and queries are turned into 768-dimensional embeddings using all-mpnet-base-v2. Important traits consist in:

- **Contextual Semantic Representation:** adds contextual relevance to queries.

- **Performance:** often beats more recent models (such as BERT) in semantic search challenges.

- **Compatibility:** Pinecone stores Embeddings for quick cosine similarity-based searches.

3.3.2 Vector Database: Pinecone

Pinecone's flawless integration into the retrieval process and strong performance in maintaining high-dimensional embeddings define it as the vector database. Important qualities include:

- **Scalability:** Pinecone is made to effectively index and control millions of embeddings, hence guaranteeing low-latency retrieval even as data volume increases. Applications with fast growing datasets will find this scalability perfect.

- **Metric Support:** Pinecone provides very accurate and significant search results by using cosine similarity as the main measure for relevance ranking; these are vital for guaranteeing that the most contextually relevant information is obtained.

- **High Performance:** The database's performance optimization allows real-time searches and changes. For uses like the medical chatbot system that need for quick answers, this is absolutely vital.

- **Integration with Retrieval Pipelines:** Pinecone simplifies the process of embedding storage and retrieval by deftly interacting with frameworks like LangChain. From query embedding to final answer production, this close connection helps to keep a seamless flow.

- **Reliability and Consistency:** For systems managing sensitive and vast data, Pinecone provides a dependable infrastructure guaranteed by durability and consistency of data.

- **Ease of Use:** Pinecone helps developers to construct and run the vector database with little overhead by means of a user-friendly API and strong documentation, therefore promoting quicker development cycles.

3.4 Agentic Appointment Booking

3.4.1 Triggering Conditions and Workflow

The system autonomously handles appointment booking by:

1. Detecting user intent through keywords (e.g., "book an appointment").
2. Collecting details like location and specialty.
3. Generating a booking link via Doctolib and sending it through Twilio SMS integration.

3.5 User Interface Approaches

3.5.1 Flask UI

Flask is used for:

- **User Authentication:** Role-based access control for students and patients.
- **File Upload:** Allowing PDF uploads for medical students.
- **Query Form:** Basic input form for submitting questions.

The Flask UI is made to easily manage file uploads, user registration, and login. It also links with another Chainlit process with a chat-centric UI. Successful login causes Flask to launch Chainlit in a fresh thread sending the user's email via environment variables so Chainlit may retrieve role related information from the database. This connection guarantees that following authentication users are swiftly sent to a dynamic and responsive chatbot interface.

3.5.2 Chainlit UI

Chainlit provides a chat-centric interface designed to enhance user engagement through interactive and dynamic conversation flows. Key features include:

- **Role-Specific Chat Profiles:** The technology lets patients and medical students communicate in numerous ways that fit them. Whether it implies in-depth academic discussions or simple, attainable health advice, every profile provides customised questions and answers to ensure the conversation remains relevant to the user′s situation.

- **Interactive Conversation Elements:** Just a few of the components Chainlit UI offers include real-time feedback, flexible chat threads, and dynamic content loading. This allows users to interact dynamically with the chatbot since the UI varies depending on user input and inquiry context.

- **Enhanced User Engagement:** Chainlit works to provide an interesting experience by combining interactive buttons and multimedia components. Users may, for instance, click on recommended subjects or follow-up questions, which simplifies browsing difficult material.

- **Seamless Integration with Back-End Systems:** The UI is made to operate in concert with the underlying retrieval and generating processes. It lets users modify or expand their searches interactively and gets context-aware results from the Mistral 13B Quantized Model that show in an easily consumable way.

- **Customization and Scalability:** Chainlit′s modular architecture lets one quickly change chat features and discussion patterns. As new features are included or user demands change, this adaptability helps to promote scalability and continuous improvement.

3.6 Summary

This chapter presents in general a strong concept for an artificial intelligence-driven medical chat bot that efficiently serves two different user groups: patients and medical students. The modular design of the system combines user identification, PDF processing, semantic embedding using all-mpnet-based-v2, and a vector database via Pinecone for fast retrieval. To provide context-aware replies, a retrieval-augmented generating pipeline deftly combines the potent Mistral 13B Quantized Model. Moreover, the system offers specific features like autonomous appointment booking and role-based interaction, therefore assuring that practical and instructional demands are both effectively met. Key component is the way the Chainlit chat interface and the Flask UI are integrated to provide dynamic, real-time interaction using user-specific data to customize answers and services. The implementation specifics, system deployment, and other integration techniques will cover in the future chapter.

Chapter 4. Implementation

In this chapter, a comprehensive overview of the libraries utilized throughout the project is presented, alongside the roles each library plays in different sections of the code. By offering deeper insight into how each component works and how the algorithms are implemented, this thesis is brought to its conclusion.

For the implementation of the thesis, I have used platforms such as VS Code (Visual Studio Code) and Jupyter Notebook. While VS Code is an integrated development environment (IDE) suitable for a variety of languages and extensions, Jupyter Notebook offers a web-based environment that is well-suited for interactive coding and data visualization.

4.1 Libraries

In this section, we focus on the various libraries used in the project and how they contribute to the overall model implementation process.

sqlite3:

The `sqlite3` module is part of Python's standard library, enabling lightweight database operations. It is used here to store and retrieve user information, roles, and session data. This is particularly helpful for managing authentication details in a local, file-based database, without requiring an external database server Python Documentation [2023].

httpcore (TimeoutException):

`httpcore` is a low-level HTTP library used to manage connection pooling, timeouts, and retries. In this thesis, it is mainly leveraged to handle `TimeoutException` cases, ensuring the application does not hang indefinitely during network calls Python Documentation [2023].

Selenium:

Selenium automates browsers, enabling dynamic web scraping. It is used in conjunction with the Edge WebDriver to navigate pages, locate elements, and extract data. Relevant classes such as `Service`, `Options`, and `WebDriverWait` help in configuring and controlling the browser for scraping tasks Python Documentation [2023].

BeautifulSoup (bs4):

`BeautifulSoup` parses HTML (and XML) content, transforming the raw webpage data into a navigable tree. Used together with Selenium, it allows the program to extract specific tags, links, or other structured elements from the loaded pages Python Documentation [2023].

time:

This built-in Python module provides sleep intervals and basic time management. Incorporating small delays can prevent overwhelming target servers, especially when performing repetitive automated scraping Python Documentation [2023].

json:

The `json` module encodes and decodes data in JSON (JavaScript Object Notation). It is both human-readable and language-independent, making it ideal for transmitting and storing structured data. In this project, JSON formats are used to manage intermediate results, user sessions, or to pass data between components Python Documentation [2023].

chainlit:

`chainlit` provides a framework for building chat-based interfaces around large language models (LLMs). It handles session management, real-time conversation flows, and prompt configuration, giving users an intuitive platform to interact with AI-driven functionalities Python Documentation [2023].

src.helper:

While not a standard library, this local module contains custom utilities such as `download_hugging_face_embeddings`. It automates loading the required embeddings from Hugging Face, ensuring consistent embedding generation across the system Python Documentation [2023].

langchain_community.vectorstores (Pinecone):

This integration connects LangChain's data processing workflows with the Pinecone vector database. By creating embeddings and storing them in Pinecone, the system retrieves similar text chunks or documents quickly, enhancing response accuracy in the chatbot Python Documentation [2023].

pinecone:

The `pinecone` SDK communicates with the Pinecone service, a specialized vector database built for high-speed similarity search. `PineconeClient` handles index creation and queries, while `ServerlessSpec` configures the underlying infrastructure Python Documentation [2023].

langchain.prompts (PromptTemplate):

`PromptTemplate` structures how queries are fed into large language models. By separating static and dynamic parts of a prompt, it creates systematic instructions for LLM-based generation Python Documentation [2023].

langchain_community.llms (CTransformers):

`CTransformers` provides an interface for loading and running transformer models within the LangChain ecosystem. It can be more memory-efficient and can help speed up inference in certain configurations Python Documentation [2023].

langchain.chains (RetrievalQA):

`RetrievalQA` coordinates the search of relevant vectors/documents before the final language model call. This leads to more accurate, context-specific responses to user questions, making the chatbot more robust Python Documentation [2023].

langchain.document_loaders (PyPDFLoader):

`PyPDFLoader` helps parse PDF documents into text chunks, enabling easy ingestion of content into the retrieval system. This is crucial when dealing with PDF-based data sets Python Documentation [2023].

langchain.text_splitter (CharacterTextSplitter):

Splits text into manageable chunks based on character count. By doing so, the system can embed and retrieve smaller, more relevant pieces of information during user queries Python Documentation [2023].

dotenv (load_dotenv):

Loads environment variables from a `.env` file, preventing sensitive credentials (like API keys) from being exposed in the source code. This approach enhances security and deployment flexibility Python Documentation [2023].

sentence_transformers:

This library provides state-of-the-art models for sentence and paragraph embeddings. By embedding text into vector representations, the system can compare semantic similarities and rank documents according to relevance Python Documentation [2023].

twilio.rest (Client):

Twilio's Python SDK handles sending SMS messages or other communications. The chatbot uses this to send appointment reminders or booking confirmations to users, enhancing the system's real-world utility Python Documentation [2023].

os:

A built-in Python module for OS-level interactions, such as file path handling and environment variable lookups. Used to coordinate local file structure and manage `.env` variables Python Documentation [2023].

logging:

`logging` is the standard Python library for capturing and recording logs, warnings, and error messages. It helps in debugging and monitoring the application's performance Python Documentation [2023].

warnings:

The `warnings` module is used to handle non-critical warnings in Python. In this context, it can hide or filter out messages from dependencies that are not relevant to the end-user Python Documentation [2023].

asyncio:

`asyncio` facilitates concurrent execution of tasks, leveraging an event loop to handle long-running or I/O-bound processes without blocking other functionalities. Particularly useful for background tasks in chatbots Python Documentation [2023].

Flask:

Flask is a lightweight Python framework used to build web applications and RESTful APIs. It manages server-side processes, routes, and user sessions, playing an important role in bridging the front-end user interface with back-end logic Python Documentation [2023].

database:

This local Python module contains functions like `init_db`, `get_user`, and `add_user`, supporting user registration, retrieval, and management. It leverages `sqlite3` to store and fetch user credentials Python Documentation [2023].

subprocess:

The `subprocess` module lets you spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This functionality is essential when starting or stopping auxiliary services like a Chainlit server Python Documentation [2023].

threading:

`threading` supports concurrency by running multiple threads in parallel. In this project, it helps avoid blocking the main application by delegating tasks (such as server checks) to background threads Python Documentation [2023].

requests:

`requests` simplifies making HTTP requests. It is used to communicate with external APIs (e.g., Pinecone, Twilio, or third-party services) and verify that local services like Chainlit are up and running Python Documentation [2023].

numpy:

`numpy` provides a multi-dimensional array object and routines for fast array manipulations, serving as the fundamental package for numerical computations in Python Python Documentation [2023].

plotly.graph_objects:

`plotly` is a plotting library that generates interactive data visualizations. The `graph_objects` module provides granular control over chart elements, suitable for in-depth analysis of chatbot performance data Python Documentation [2023].

HTML/CSS:

The front-end user interface is structured using HTML and styled with CSS. HTML forms gather user input, while CSS ensures the layout is visually appealing and user-friendly. This combo supports usability and meets the requirements for practical deployment.

4.2 Code Implementation

In this chapter, we map the major steps of our **architecture** (see Figure **??**) to their respective **code components**. Each section follows the flow of data and logic from user authentication all the way to generating a final answer.

4.3 User Authentication and Role Identification

User management begins with:

- **Signup & Role Assignment:** A new user provides an email and password. The system assigns the role (*e.g., patient* or *medical student*) by storing it in a lightweight database (e.g., using `sqlite3`).
- **Login:** On login, the code checks the role from the database to serve the correct chatbot mode.
- **Session Handling:** Flask or Chainlit stores session data (e.g., user email and role) so each request can correctly route to patient or student logic.

4.4 PDF Processing (Text Extraction and Chunking)

PDF documents (e.g., medical articles) are uploaded by students:

- **Extraction:** The code typically uses a library like `PyPDFLoader` to read and convert PDF pages to raw text.
- **Chunking:** Long text is split into manageable pieces via `CharacterTextSplitter` or custom splitting logic, ensuring each chunk remains semantically coherent.
- **Preprocessing:** Code may further remove stopwords or apply cleaning to each chunk before embedding.

4.5 Embedding Generation

Each text chunk is converted into a numerical representation (an *embedding*):

- **Model Setup:** A pretrained model (e.g., `all-mpnet-base-v2` or `SentenceTransformer`) is loaded.
- **Batch Embedding:** The code loops through chunks and feeds them into the embedding model. This step often includes concurrency or batch processing to optimize runtime.
- **Storage:** Embeddings are passed either to an in-memory structure or a vector database (such as Pinecone) for fast retrieval.

4.6 Pinecone Vector Database Integration

After the system generates embeddings for each text chunk (Section 4.5), the next step is to store them in a **vector database** for fast retrieval. Here, we integrate with *Pinecone* to manage and query these embeddings efficiently:

- **Index Creation**: The system checks whether an index (e.g., `"zf"`) already exists in Pinecone. If not, it creates one with a specified dimension (e.g., 768) and metric (e.g., `cosine`). This ensures compatibility with the selected embedding model (e.g., `all-mpnet-base-v2`).
- **Batch Upload**: Each text chunk is embedded (transformed into a numerical vector) and then stored in Pinecone with a unique identifier. Doing so in batches reduces overhead when dealing with large datasets.
- **High-Speed Retrieval**: When the user asks a question or enters a search query, the system converts their text into an embedding and performs a similarity search against the Pinecone index. The top-k most relevant chunks are returned to provide context for LLM-based responses (Section 4.15).
- **Scalability & Flexibility**: Pinecone accommodates millions of embeddings and quickly returns nearest neighbors, allowing the system to handle ever-growing corpora of PDF documents, scraped data, or other text sources without reprocessing older content.

By coupling embeddings (Section 4.5) with Pinecone's vector database, the system can efficiently surface **contextual insights** and pass them to the Large Language Model for retrieval-augmented generation. This approach significantly boosts the accuracy and relevance of chatbot answers, particularly in a medical or academic setting where **precise factual references** are paramount.

4.7 Retrieval-Augmented Generation Pipeline

To handle user queries, we blend retrieval (based on embeddings) with LLM generation:

- **Query Handling:** Once a user query arrives, the system transforms it into an embedding (using the same model as above).
- **Similarity Search:** This embedding is compared against stored embeddings in Pinecone (or another vector store). The top-k relevant chunks are returned.

- **Prompt Augmentation:** These chunks are appended to the user's query to form a "context window" for the Large Language Model (LLM).
- **Response Generation:** The LLM (e.g., Mistral 13B) produces an answer grounded in the retrieved text. The code also applies any prompt template rules or role-specific instructions.

4.8 Role-Based Logic (Learning Assistant vs. Diagnostic Agent)

In the code, a conditional block or chain-level branching typically handles:

- **Student Mode (Learning AI Assistant)**: Provides detailed, in-depth explanations with references.
- **Patient Mode (Agentic Diagnostic AI)**: Focuses on simpler language, symptom triage, or booking logic. When users mention booking an appointment, an agentic step can automatically interact with a scheduling API.

4.9 Symptom Analysis, Provider Recommendation, and Appointment Booking

For patient-focused tasks, code sections may:

- **Symptom Analysis**: Use a rule-based approach or LLM prompts fine-tuned on medical data to interpret user symptoms.
- **Provider Recommendation**: Parse a local database (or external service) of doctors. The code filters by specialty or location, returning suitable recommendations.
- **Appointment Booking**: Integrate with an external API (e.g., Twilio for SMS, Doctolib for scheduling). The code triggers a request, confirms details, and notifies the user.

4.10 Front-End Integration

- **Flask or Chainlit Routes**: A Python route handles file uploads (for PDFs) or query input. These routes pass data to the back-end logic (embedding, retrieval, LLM).

- **User Interface**: HTML/CSS or a Chainlit-based chat bubble interface. The code updates the chat context in real-time and displays responses from the LLM.
- **Templating**: If using Flask, Jinja templates might render dynamic pages with user-specific data.

4.11 Error Handling and Logging

Throughout the system:

- **Try-Except Blocks**: Key places (e.g., PDF loading, retrieval queries, LLM calls) are wrapped in `try-except` to gracefully handle failures.
- **Logging**: The Python `logging` module or a custom logger records warnings, debug info, and errors. This helps in diagnosing issues without exposing them to end users.
- **Notifications**: In critical scenarios (like an inability to retrieve from the vector database), the code might send an alert or fallback to a simpler pipeline.

4.12 User Authentication and Role Identification

Below is an excerpt of Python/Flask code illustrating how user login, signup, and role identification are handled:

LISTING 4.1: User authentication and role assignment code

```python
from flask import Flask, render_template, request, redirect, url_for
from database import init_db, get_user, add_user
import subprocess
import threading
import os
import time
import requests

app = Flask(__name__)

# Initialize the database
init_db()

FLASK_PORT = 5002
CHATBOT_PORT = 8001
```

```
chainlit_process = None

def is_chainlit_running():
    try:
        response = requests.get(f"http://localhost:{CHATBOT_PORT}")
        return response.status_code == 200
    except requests.exceptions.ConnectionError:
        return False

def start_chainlit(email):
    global chainlit_process
    if is_chainlit_running():
        print("Chainlit is already running.")
        return

    print(f"Launching Chainlit on port {CHATBOT_PORT}...")
    chainlit_process = subprocess.Popen(
        ["chainlit", "run", "app.py", "--port", str(CHATBOT_PORT)],
        cwd=os.path.dirname(os.path.abspath(__file__)),
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
        start_new_session=True,
        env={
            **os.environ,
            "LOGGED_IN_EMAIL": email
        }
    )

@app.route("/", methods=["GET", "POST"])
def login():
    """Login route to authenticate users and start Chainlit."""
    if request.method == "POST":
        email = request.form.get("email")
        password = request.form.get("password")
        user = get_user(email)
        if user and user[2] == password:
            print(f"LOGIN SUCCESS: {email}")

            # Start Chainlit if not running; pass the email to the new process
            threading.Thread(target=start_chainlit, args=(email,), daemon=True).start()

            # After login, display a loading page that will link to Chainlit
            chatbot_url = f"http://localhost:{CHATBOT_PORT}/"
            print(f"REDIRECTING TO: {chatbot_url}")
            return render_template("loading.html", chatbot_url=chatbot_url)
        else:
            return render_template("login.html", error="Invalid email or password.")
    return render_template("login.html")

@app.route("/signup", methods=["GET", "POST"])
def signup():
    """Signup route to register new users."""
    if request.method == "POST":
        email = request.form.get("email")
        password = request.form.get("password")
        role = request.form.get("role")
        if add_user(email, password, role):
            return redirect(url_for("login"))
        else:
            return render_template("signup.html", error="Email already registered.")
    return render_template("signup.html")

if __name__ == "__main__":
    app.run(port=FLASK_PORT, debug=True)
```

4.12.1 Initialization of the Database

- `init_db()`: Called immediately upon starting the Flask application to ensure that the local SQLite database (or whichever database is being used) is properly initialized. This might create tables such as `users` if they do not exist.

4.12.2 Signup Logic

- **Route:** `/signup` handles both `GET` (displaying a signup form) and `POST` (processing form data).
- **Form Inputs:** Users provide `email`, `password`, and `role` (for example, "PATIENT" or "STUDENT").
- **add_user**: A database function that inserts a new record into the `users` table. If the operation is successful, the user is redirected to the login page; otherwise, an error is displayed (e.g. "Email already registered").

4.12.3 Login Logic

- **Route:** `/` or `/login`, handling both `GET` (showing the login template) and `POST` (verifying credentials).
- **Credential Check:** The user's provided email is passed to `get_user(email)`, which queries the database for a matching record (typically returning a tuple $(id, email, password, role)$).
- **Password Match:** `user[2] == password` ensures the stored password matches the user's input. If it matches, login is considered successful.
- **Chainlit Process:** After a successful login, the function `start_chainlit(email)` is triggered in a background thread. This:
  1. Checks if Chainlit is already running via `is_chainlit_running()`.
  2. If not running, spawns a subprocess with the user's email set as an environment variable. This email can then be used by the chatbot to fetch the user's role from the database without needing URL parameters.
- **Redirect to Loading Screen:**
  – Renders the template `loading.html`, which contains a link or auto-redirect to the running Chainlit instance at `http://localhost:8001`.
- **Error Handling:** If the email/password is incorrect, the user is shown the `login.html` form again, this time with an error message ("Invalid email or password").

4.12.4 Role Identification & Usage

- **During Signup:**

1. The `role` field is selected by the user or assigned automatically (e.g., `PATIENT` or `STUDENT`).
2. `add_user` stores the role in the database alongside the `email` and `password`.

- **During Login:**
  1. Once the login is validated, the user's `role` can be fetched from `user[3]`.
  2. In a fully built system, the chatbot interface might reference this role to load different conversation chains or specialized functionality. For instance:
     - *Medical Students* might get an LLM chain focusing on advanced medical knowledge.
     - *Patients* might see simplified symptom analysis tools and appointment booking flows.

4.12.5 Chainlit Integration

- **is_chainlit_running()**: Sends an HTTP GET request to `http://localhost:8001` (the configured Chainlit port). If the server responds with a `200` status code, we assume Chainlit is active.
- **start_chainlit(email)**:
  1. Uses `subprocess.Popen` to run `chainlit` on the given port if not already running.
  2. Exports `LOGGED_IN_EMAIL` as an environment variable. This allows the Chainlit app to retrieve the user's role from the database with the same email, applying role-specific logic during chatbot interactions.

4.12.6 Overall Flow

1. A new user signs up using `/signup`, providing email, password, and role.
2. The system calls `add_user` to store these details in the local database.
3. The user then goes to `/login`, enters the same email and password, which `get_user` verifies.
4. If successful, `start_chainlit` is triggered, launching or reusing a Chainlit session for the user.
5. The user is redirected to a loading page and then to the Chainlit UI, where their role (retrieved from the database) dictates the AI chatbot's behavior.

In short, this code section ensures:

- Secure handling of **login** and **signup** flows.
- Automatic **role assignment** and **retrieval** from the database.
- Seamless bridging between the Flask web server and the **Chainlit** chatbot interface.

4.13 PDF Processing (Text Extraction and Chunking)

In this part of the code, we handle the **user-uploaded PDF files**—from reading the file's pages to splitting the text into smaller segments (chunks) that are later added to our **vector store** for retrieval. Below is a detailed snippet and explanation:

LISTING 4.2: PDF Processing and Chunking

```
# If user uploaded a file via paperclip
if message.type == "file":
    logging.info("Processing uploaded file from user via paperclip...")

    # 1 Load the PDF using PyPDFLoader
    loader = PyPDFLoader(message.file_path)
    documents = loader.load()

    # 2 Split text into smaller chunks
    text_splitter = CharacterTextSplitter(chunk_size=4000, chunk_overlap=500)
    texts = text_splitter.split_documents(documents)

    # 3 Add the chunks to the vector store in manageable batches
    batch_size = 10
    for i in range(0, len(texts), batch_size):
        batch = [t.page_content for t in texts[i : i + batch_size]]
        docsearch.add_texts(batch)

    # 4 Notify the user about successful processing
    await cl.Message(
        content=(
            "PDF uploaded and processed successfully! "
            "You can now ask questions about its content."
        )
    ).send()

    return
```

Detailed Explanation

1. **Detecting a File Upload**
   - The code checks if `message.type == "file"`. If so, it logs a message indicating that a PDF file has been received from the user through the **paperclip icon** in Chainlit.
   - Using an `if` condition around the file-logic ensures text messages and file uploads are handled separately in the same event function.
2. **Loading the PDF with `PyPDFLoader`**
   - `PyPDFLoader` is a document loader that reads PDF files and converts them into a list of `Document` objects—one for each page.
   - Internally, it might use a library like PyPDF2 or pdfplumber to extract text. Here, `loader.load()` creates an in-memory representation of the PDF content as `documents`.
3. **Splitting Text into Chunks**
   - **Why split?** Large text blocks can overwhelm certain embedding or retrieval systems. Breaking down pages into chunks helps keep context manageable and supports more accurate retrieval results (since each chunk remains more focused on a single topic).
   - `CharacterTextSplitter`:
     - `chunk_size=4000` sets the maximum number of characters in a chunk.
     - `chunk_overlap=500` ensures a 500-character overlap between consecutive chunks. This overlap helps preserve continuity of context between two consecutive chunks. When retrieving answers, slightly overlapping text can prevent abrupt context cuts.
   - `text_splitter.split_documents(documents)`:
     - Loops over each `Document` (e.g., each PDF page) and subdivides it into multiple, partially-overlapping text segments.
     - Returns a list of *split* text segments, stored in the `texts` variable.
4. **Batch Insertion into the Vector Store**

- A *vector store* (like Pinecone) is used to store embeddings so that relevant chunks can be quickly retrieved based on semantic similarity.
- We define `batch_size = 10` to avoid feeding all chunks at once, which could be memory-intensive if the PDF is large.
- The loop `for i in range(0, len(texts), batch_size)` processes chunks in increments of 10. Each iteration:
  (a) Slices a `batch` of 10 from the `texts` list.
  (b) Calls `docsearch.add_texts(batch)` to add that subset of chunks to the vector database.
- Splitting into batches ensures stable performance and reduces the risk of timeouts or memory spikes.

5. **User Notification**
   - After successfully uploading and splitting the PDF content, the user is informed with a message (“PDF uploaded and processed successfully!”).
   - This immediate feedback ensures a good user experience by confirming that the system is ready to handle queries about the newly indexed content.
6. **Return Statement**
   - The function ends (`return`) to avoid further processing of the `message` as if it were a text query.
   - This prevents any additional or conflicting logic from executing once the file-handling code finishes.

Why Chunking is Important

- **Efficient Retrieval:** Many LLM-based retrieval systems do best when dealing with smaller segments. Searching and re-ranking short chunks is often faster and more precise.
- **Context Preservation:** Overlapping chunks maintain local context across boundary areas, which is crucial for question answering. If text relevant to a user’s question is split between the end of one chunk and the start of the next, the chunk overlap ensures minimal information loss.
- **Scalability:** Large PDF files with dozens or even hundreds of pages become more manageable. The user can ask questions about any part of the document without manually referencing page numbers or paragraphs.

This upload, split, and index workflow is central to enabling a dynamic question-answer system, where each user can bring in new documents and instantly query them. Once the text is converted into vector form, the system can retrieve it for any subsequent queries, seamlessly integrating new medical or academic documents into the user's knowledge base.

4.14 Embedding Generation

This portion of the code lays out how we prepare a sentence-level embedding model and connect it to the vector database—ensuring that new text segments are converted into numerical vectors (embeddings). Below is the relevant snippet, with an in-depth discussion:

LISTING 4.3: Embedding Model Setup and Pinecone Integration

```python
# 1 Load embeddings for all-mpnet-base-v2 (768-dimensional embeddings )
embedding_model = SentenceTransformer('all-mpnet-base-v2')
embeddings = download_hugging_face_embeddings()

# 2 Initialize Pinecone
pinecone_instance = PineconeClient(api_key=PINECONE_API_KEY)
if index_name not in [index.name for index in pinecone_instance.list_indexes()]:
    pinecone_instance.create_index(
        name=index_name,
        dimension=768,
        metric='cosine',
        spec=ServerlessSpec(cloud='aws', region=PINECONE_API_ENV)
    )

# 3 Access the index
docsearch = Pinecone.from_existing_index(index_name, embeddings)
```

Step-by-Step Explanation

1. **Loading the Embedding Model**
    - `SentenceTransformer('all-mpnet-base-v2')`: We instantiate a pre-trained embedding model from the *Sentence Transformers* family.

    - *all-mpnet-base-v2* outputs vectors of size 768 for each sentence or text chunk.
    - This particular model is well-regarded for capturing semantic similarities, making it suitable for question answering or retrieval tasks.
  - `embeddings = download_hugging_face_embeddings()`: A custom utility, presumably in `src.helper`, that either fetches or caches embedding-related data.
    - This may ensure version consistency, or optimize performance by not repeatedly downloading large model files.
    - The result is likely a reference to the same dimension/embedding pipeline used by `SentenceTransformer`, meaning the code can embed both documents and user queries in a matching format.

2. **Initializing Pinecone (Vector Database)**
   - `pinecone_instance = PineconeClient(api_key=PINECONE_API_KEY)`: This sets up communication with Pinecone's vector database, using the API key provided via environment variables.
   - We then check whether our target `index_name` (here, `zf`) exists:
     - `pinecone_instance.list_indexes()` returns a list of existing indexes.
     - If `zf` is missing, we call `pinecone_instance.create_index(...)`:
       * `dimension=768` aligns with the output dimension of `all-mpnet-base-v2`.
       * `metric='cosine'` ensures we measure distance (or similarity) in terms of cosine similarity, which is a common choice for text embeddings.
       * `ServerlessSpec` configures the environment (e.g., `cloud='aws', region=PINECONE_API_ENV`), controlling where the index is hosted.
   - If an index called `zf` already exists, creation is skipped—this prevents overwriting data.
3. **Linking the Embedding Model to the Index**
   - `docsearch = Pinecone.from_existing_index(index_name, embeddings)`: We create a *docsearch* object that uses the `embeddings` logic in tandem with our newly confirmed Pinecone index.
   - *Why do we pass `embeddings` here?* This ensures any text we embed for

Pinecone is done with the `all-mpnet-base-v2` model, so that stored vectors and newly embedded queries are directly comparable.
- **Result:** A retrieval interface that can accept text (chunks, user queries, etc.), embed it, and perform similarity lookups within Pinecone.

How the Embedding Pipeline Works in Practice

1. **Ingest Documents:** When the user uploads a PDF (as shown in Section 4.13), each chunk is sent to `docsearch.add_texts(...)`. Under the hood, it uses `embeddings` to transform that chunk into a 768-dimensional vector, then stores it in Pinecone with a unique ID.
2. **User Queries:** At query time, the code uses the same embedding model to turn a question into a vector. Pinecone returns the most similar stored chunks (based on cosine similarity).
3. **Contextual Answering:** These top chunks (i.e., relevant lines/paragraphs) are then fed into a Large Language Model, which forms a final answer.

Advantages of a Shared Embedding Model

- **Consistency:** Both documents and queries are embedded using the same approach, guaranteeing semantic alignment.
- **Scalability:** Additional PDFs or text blocks can be seamlessly integrated (embeddings stored) without reprocessing older data.
- **Performance:** Cosine similarity in a vector database like Pinecone is highly optimized, allowing low-latency lookups—even at large scales.

Overall, this embedding generation mechanism is critical for bridging raw textual data with *retrieval-augmented* chat capabilities. Once the embeddings are set up and the index is ready, the system can handle user queries and fetch relevant information in real time.

4.15 Retrieval-Augmented Generation Pipeline

A core feature of this system is its **retrieval-augmented generation (RAG)** approach, where a Large Language Model (LLM) incorporates relevant text chunks from a vector store (Pinecone) to ground its responses in factual context. Below is the relevant code snippet, followed by a detailed walkthrough:

LISTING 4.4: Retrieval-Augmented Generation (RAG) Pipeline

```python
# 9) Long-running QA task
def long_running_task(input_data, llm):
    logging.info("Generating response...")
    qa = RetrievalQA.from_chain_type(
        llm=llm,
        chain_type="stuff",
        retriever=docsearch.as_retriever(search_kwargs={'k': 5}),
        return_source_documents=True,
        chain_type_kwargs={"prompt": PROMPT}
    )
    result = qa.invoke(input_data)
    logging.info("Response generated: %s", result["result"])
    return result["result"]

async_long_running_task = cl.make_async(long_running_task)

# ...

@cl.on_message
async def handle_message(message):
    """
    Main message handler that processes both text messages and file
    uploads.
    When dealing with user queries, it triggers the RAG pipeline.
    """
    try:
        # ...
        # 1 Distinguish between PDF upload and text queries
        if message.type == "file":
            # [PDF upload handling code removed for brevity]
            return

        # 2 For text messages (i.e., user queries):
        query = message.content.lower()

        # [Optional: Role-based branching omitted here for brevity]

        # 3 Re-initialize the same LLM
        llm = initialize_llm()

        # 4 Notify the user that the system is working on the response
        await cl.Message(
            content="Processing your request, please wait..."
        ).send()

        # 5 Perform retrieval
        retriever = docsearch.as_retriever(search_kwargs={'k': 5})
        docs = retriever.invoke(query)
        context = " ".join([doc.page_content for doc in docs])

        # 6 Prepare the combined input data for the chain
        input_data = {"query": query, "context": context}

        # 7 Execute the retrieval-augmented QA pipeline asynchronously
        result = await async_long_running_task(input_data, llm)

        # 8 Send the final answer back to the user
        await cl.Message(content=result).send()

    except Exception as e:
        logging.error(f"Error occurred: {e}")
        await cl.Message(
            content="An error occurred while processing your request. Please try again."
        ).send()
```

Detailed Explanation

1. **Long-running QA Task (`long_running_task`)**
   - We define a function that bundles up *Retrieval* + *Generation* into a single workflow:

- `RetrievalQA.from_chain_type(...)` : A LangChain method that composes two main elements:
  (a) A *retriever*, which fetches the most relevant chunks (here, using `docsearch.as_retriever()`)
  (b) A language model (`llm`) that we pass in as a parameter.
- `chain_type="stuff"`: Tells LangChain to "stuff" the retrieved chunks into a single prompt, which the LLM then processes.
- `return_source_documents=True`: Ensures the pipeline can return which text chunks were used, aiding traceability and debug.
- `chain_type_kwargs={"prompt": PROMPT}`: Injects our custom prompt template, controlling how context and user query are combined in the final prompt.

• `result = qa.invoke(input_data)`: Takes the dict `{"query", "context"}` and feeds it into the chain. The chain formats a prompt (via `PROMPT`), appends relevant text, and runs the final LLM call.

• The function logs the output and returns `result["result"]`, the LLM's textual answer.

2. **Asynchronous Execution (`async_long_running_task`)**
   • `cl.make_async` wraps the synchronous `long_running_task` so it can be awaited within an `async` function (`handle_message`).
   • This prevents the entire chatbot loop from blocking while the LLM generates a response—especially important if inference is slow or if many users are connected.

3. **Main Message Handler (`handle_message`)**
   • This function processes every user message. It distinguishes between file uploads (`if message.type == "file"`) and text queries.
   • If it's a *text query*, the following steps occur:
   (a) `query = message.content.lower()`: The user's text is captured.
   (b) `llm = initialize_llm()`: We load or re-initialize the LLM (e.g., a Mistral model).
   (c) `await cl.Message(content="...").send()`: A quick message is sent to let the user know the system is working.
   (d) **Retrieval**:
      - `retriever = docsearch.as_retriever(search_kwargs={'k': 5})` uses the Pinecone-based docsearch to find up to five relevant

chunks for the user's query.

- `docs = retriever.invoke(query)` runs the similarity lookup.
- Then we build a `context` string out of these chunks, typically by concatenating them with a space.

(e) **Chain Input**:
- `input_data = {"query": query, "context": context}` is the dictionary that gets passed to our QA chain (via `long_running_task`).

(f) `result = await async_long_running_task(input_data, llm)`: We run the RAG pipeline asynchronously.

(g) Finally, we send the LLM's answer back to the user with `await cl.Message(content=result).send()`.

4. **Prompt Template (`PROMPT`)**
   - Defined elsewhere in the code, it might look like:

```
PROMPT = PromptTemplate(
    template=(
        "You are a medical assistant. "
        "Based on the following context, answer the question concisely:\n\n"
        "Context: {context}\n\n"
        "Question: {question}\n\n"
        "Answer:"
    ),
    input_variables=["context", "question"]
)
```

   - The placeholders `context` and `question` are substituted with the text from the retriever.
   - This structure forces the LLM to ground its answer in the retrieved snippets, mitigating hallucinations.

Why RAG Matters

- **Accuracy and Context**: The LLM is less likely to invent facts, since it sees the actual text relevant to the user's question.
- **Extensibility**: Users can add more documents over time. The system can then answer questions about newly introduced material without retraining the LLM.

- **Traceability**: Because `return_source_documents=True` is set, the pipeline can record which chunks were used, enabling better debugging or even letting the user see those sources for reference.

Overall, **retrieval-augmented generation** fuses an *external knowledge store* (Pinecone) with a powerful *language model* (like Mistral). The final outcome is a **contextual, verifiable answer** that draws from user-uploaded PDFs or other indexed material, rather than relying solely on the LLM's internal training data.

4.16 Role-Based Logic (Learning Assistant vs. Diagnostic Agent)

In this system, the user's *role*—either a **Student** in need of detailed, academic explanations or a **Patient** requiring concise diagnostic guidance—shapes the chatbot's behavior. Below is a relevant code snippet from the `start_chat` and `handle_message` functions, illustrating how we differentiate between these modes:

LISTING 4.5: Role-Based Logic Snippet

```
@cl.on_chat_start
async def start_chat():
    """
    On chat start, we read LOGGED_IN_EMAIL from environment,
    fetch the user's role from DB, store it in session, and greet
  them.
    """
    email = os.environ.get("LOGGED_IN_EMAIL", "")
    role = None

    if email:
        user = get_user(email)  # e.g. (id, email, password, role)
        if user:
            role = user[3].strip().upper()  # e.g. 'PATIENT' or '
  STUDENT'

    cl.user_session.set("role", role)

    if role == "PATIENT":
        await cl.Message(
            content=(
                "Patient Chatbot\n\n"
                "Welcome! You can upload a PDF (via the paperclip icon) or ask a biomedical question. "
                "To book an appointment, type ’book an appointment’."
            )
        ).send()
    elif role == "STUDENT":
        await cl.Message(
            content=(
                "Student Chatbot\n\n"
                "Welcome! You can upload a PDF (paperclip icon) or ask a biomedical question to get started."
            )
        ).send()
    else:
        await cl.Message(content="ERROR: Role not recognized. Contact Support.").send()

@cl.on_message
async def handle_message(message):
    try:
        role = cl.user_session.get("role")  # ’PATIENT’ or ’STUDENT’ or None

        # If user is a PATIENT, handle special logic (e.g., appointment booking)
        if role == "PATIENT":
            # [omitted: user_appointment_info logic, location/specialty checks, etc.]

            # If no special triggers, continue to normal query handling
            # ...

        # If user is a STUDENT, or we are done with appointment logic:
```

```
        # ... (common retrieval + LLM steps go here)

    except Exception as e:
        logging.error(f"Error occurred: {e}")
        await cl.Message(
            content="An error occurred while processing your request.
    Please try again."
        ).send()
```

Detailed Explanation

1. **Role Extraction & Storage**
   - During `start_chat`, the code retrieves the user's email from environment variables (in this example, `LOGGED_IN_EMAIL`) and queries the database via `get_user` to find the stored role (e.g., `'PATIENT'` or `'STUDENT'`).
   - `cl.user_session.set("role", role)` saves this role in the session context, ensuring that each subsequent message from the user automatically references the same role without needing repeated lookups.
2. **Greeting the User**
   - If the role is `PATIENT`, the chatbot introduces itself as a "Patient Chatbot," highlighting PDF upload and biomedical question features. It also mentions the logic for appointment booking (i.e., "book an appointment").
   - If the role is `STUDENT`, the chatbot welcomes them with a focus on deeper academic or learning-related inquiries, referencing file uploads for coursework or research documents.
   - If the system cannot identify a valid role, it displays an error message advising the user to contact support.
3. **Message Handling & Conditional Flows**
   - In `handle_message`, we re-check the user's role from the `cl.user_session`.
   - **If `role == "PATIENT"`**, specialized appointment logic is available:
     - For instance, searching for "book an appointment" triggers additional steps, such as collecting the user's city and specialty, or sending a Twilio SMS with a booking link.
     - If no special triggers are detected, the code falls back to normal retrieval+LLM query handling.

- **If `role == "STUDENT"`**, the user receives more in-depth, academically oriented assistance (e.g., expanded explanations, references).
- In both cases, after role-specific checks, we eventually proceed to the standard retrieval pipeline for general Q&A or synergy with the LLM.

Why Role-Based Logic?

- **Differentiated Interaction**: Patients have simple, symptom-focused questions and possibly need appointment scheduling. Meanwhile, students want detailed, academically rigorous explanations.
- **Security & Personalization**: Certain features, such as patient data or contact info, should not be accessible to a "student" user. Role-based checks help maintain the appropriate scope of functionalities.
- **Modular Scalability**: Additional roles (e.g., "Doctor," "Researcher," or "Admin") could be integrated in the future, each with specialized logic or expanded capabilities.

Overall, role-based logic ensures the chatbot adapts to the user's specific needs. Patients can focus on receiving quick, practical medical guidance, while students can explore in-depth academic or biomedical topics—all handled smoothly in a single application.

4.17 Symptom Analysis, Provider Recommendation, and Appointment Booking

Beyond general Q&A functionality, the system also handles patient-centric tasks: analyzing user symptoms, recommending healthcare providers, and even automating appointment bookings. Below is an excerpt illustrating how these features appear in the `handle_message` function for `role == "PATIENT"`:

LISTING 4.6: Symptom Analysis & Appointment Booking Snippet

```python
# ...
if role == "PATIENT":
    query = message.content.lower()

    # 1 Trigger for booking an appointment
    if "book an appointment" in query:
        user_appointment_info[message.author] = {}
        await cl.Message(
            content="Sure! What is your location (e.g., Berlin, Munich)?"
        ).send()
        return

    # 2 If we already started booking logic
    if message.author in user_appointment_info:
        if "location" not in user_appointment_info[message.author]:
            # Capture patients location
            user_appointment_info[message.author]["location"] = message.content
            await cl.Message(
                content="Got it! What specialty do you need (e.g., cardiology, dermatology)?"
            ).send()
            return

        elif "specialty" not in user_appointment_info[message.author]:
            # Capture specialty
            user_appointment_info[message.author]["specialty"] = message.content
            await cl.Message(
                content="Finally, please provide your phone number so we can send the booking link."
            ).send()
            return

        elif "phone_number" not in user_appointment_info[message.author]:
            # 3 Final step: generate & SMS booking link
            user_appointment_info[message.author]["phone_number"] = message.content

            # Convert city & specialty to German equivalents
```

```python
            city_raw = user_appointment_info[message.author]["location"].strip()
            specialty_raw = user_appointment_info[message.author]["specialty"].strip()
            # ... (use translation maps) ...

            booking_link = f"https://www.doctolib.de/{specialty_converted}/{city_converted}"
            twilio_client.messages.create(
                body=f"Your appointment booking link: {booking_link}",
                from_=TWILIO_PHONE_NUMBER,
                to=user_appointment_info[message.author]["phone_number"]
            )
            await cl.Message(
                content=f"Your booking link has been sent to {message.content}."
            ).send()
            # Clear user_appointment_info
            del user_appointment_info[message.author]
            return

    # 4 Symptom or general patient queries fall back to normal retrieval + LLM
    # ...
```

Detailed Explanation

1. **Triggering Appointment Booking**
   - If a patient's query includes the phrase `"book an appointment"`, the system captures their user ID (i.e., `message.author`) in a dictionary called `user_appointment_info`.
   - This dictionary then tracks key steps (location, specialty, phone number) needed to finalize an appointment.
   - A message is immediately returned asking for the user's location, preventing further text processing for that round.

2. **Location & Specialty Collection**
   - Once the user flows into the "appointment" path, the logic checks which pieces of information are missing:
     (a) `location`: e.g., Berlin or Munich
     (b) `specialty`: e.g., Cardiology, Dermatology
   - The code displays short messages at each step. This ensures the user is guided through a simplified multi-turn conversation.
3. **Phone Number & Booking Link Generation**
   - When the phone number is received, the code triggers a final step that:
     (a) Converts the city name and specialty into a German-compatible format via a dictionary lookup (`city_translation_map`, `translation_map`).
     (b) Constructs a `booking_link` to a medical appointment platform (e.g., Doctolib).
     (c) Sends the link via `twilio_client.messages.create(...)`, delivering an SMS to the user.
     (d) Responds in chat with a confirmation message ("Your booking link has been sent..."), then removes the user from `user_appointment_info` to avoid confusion on future messages.
4. **Symptom or General Queries**
   - For patient queries that aren't directly about booking (e.g., "I have a sore throat..."), the code can fall back to normal retrieval + LLM logic.
   - This might involve symptom analysis or retrieving from a relevant knowledge base. The snippet above omits that for brevity.

Provider Recommendation and Symptom Analysis

- **Provider Recommendation:**
  - The specialized dictionaries (`city_translation_map`, `translation_map`) map English city or specialty names to their German equivalents. This ensures the final booking URL matches the local healthcare provider listings.
  - Further expansions could include an internal or external "doctor directory," letting the chatbot automatically suggest providers based on location and specialty, even before generating a link.
- **Symptom Analysis:**

– If the user states symptoms (e.g., "I have frequent headaches"), the code can direct the user to appropriate follow-up logic or mention relevant specialties.
– In some advanced setups, an LLM prompt specifically includes "symptom triage" or references medical guidelines, bridging from text-based retrieval to real-time suggestions (the code snippet is mostly an outline for appointment logic).

Overall, this patient-focused portion of the system ensures that common tasks—like clarifying symptoms, picking the right specialist, and securing an appointment—are streamlined. By combining manual user input flow with automated translation logic and an SMS-based booking confirmation, the bot delivers a practical end-to-end healthcare assistance workflow.

4.18 Front-End Integration

While much of the system's logic operates on the back end (via Flask and LangChain/Chainlit), the **front-end layer** is critical for user interaction. This section describes how the UI for login, signup, and Chainlit chat seamlessly tie into the back-end logic.

4.18.1 Login and Signup Pages (Flask Templates)

Below is an example of how we might structure the `login.html` and `signup.html` pages. They rely on **Flask**'s `render_template` mechanism:

LISTING 4.7: login.html (Flask Template)

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    {% if error %}
        <p style="color:red;">{{ error }}</p>
    {% endif %}
    <form method="POST" action="/">
```

```
        <label for="email">Email:</label>
        <input type="text" name="email" required><br><br>

        <label for="password">Password:</label>
        <input type="password" name="password" required><br><br>

        <button type="submit">Login</button>
    </form>
    <p>
        Don't have an account?
        <a href="{{ url_for('signup') }}">Sign up here</a>
    </p>
</body>
</html>
```

Explanation:

- The page includes a simple form posting to `/` (the login route).
- If there's an `error` (set by Flask upon invalid credentials), it displays in red text.
- A link leads users to the `signup` route if they have no account.

LISTING 4.8: signup.html (Flask Template)

```
<!DOCTYPE html>
<html>
<head>
    <title>Signup</title>
</head>
<body>
    <h2>Create an Account</h2>
    {% if error %}
        <p style="color:red;">{{ error }}</p>
    {% endif %}
    <form method="POST" action="/signup">
        <label for="email">Email:</label>
        <input type="text" name="email" required><br><br>

        <label for="password">Password:</label>
        <input type="password" name="password" required><br><br>

        <label for="role">Role (PATIENT / STUDENT):</label>
        <input type="text" name="role" required><br><br>

        <button type="submit">Sign Up</button>
    </form>
    <p>
        Already have an account?
        <a href="{{ url_for('login') }}">Login here</a>
    </p>
</body>
</html>
```

Explanation:

- This form posts to `/signup`.
- The user specifies an `email`, `password`, and `role` (e.g., "PATIENT" or "STUDENT").
- If the email is already taken, an `error` message (like "Email already registered.") is shown.

4.18.2 Chainlit Chat UI

Chainlit provides a sleek front-end interface out of the box. After the user logs in (and the Flask code launches or reuses a Chainlit instance), the user is redirected to the Chainlit UI, typically at `http://localhost:8001`. Some key features of the Chainlit interface:

- **Interactive Chat Windows**: Users can type or upload PDFs (via the *paperclip* icon) directly.
- **Session Awareness**: The environment variable `LOGGED_IN_EMAIL` and the role from the database guide how the chatbot greets or handles user messages.
- **Chat Profiles**: As defined in `@cl.set_chat_profiles`, we can brand the chat or provide special instructions (e.g., "Mistral Biomedical" for medical queries).
- **Real-time Updates**: The user sees status messages (like "Processing your request...") and final LLM replies in real time.

4.18.2.1 Loading Page (Optional)

A simple `loading.html` can serve as a bridge page after successful login, informing the user that **Chainlit** is starting up. For instance:

LISTING 4.9: loading.html (Flask Template)

```
<!DOCTYPE html>
<html>
<head>
    <title>Loading Chatbot...</title>
</head>
<body>
    <h2>Please wait, we are launching your chatbot session...</h2>
    <p>If not redirected automatically, <a href="{{ chatbot_url }}">
   click here</a>.</p>
</body>
</html>
```

Flask's login route references this by calling `render_template("loading.html", chatbot_url=chatbot_url)`.

4.18.3 Overall Flow

1. **User Visits Login Page (/)**:
   - If they **POST** valid credentials, the server spawns or reuses a Chainlit process and redirects them to `loading.html`.
2. **Loading Screen**:
   - The user is shown a simple "please wait" message. After a short wait, they can click (or get auto-redirected) to `http://localhost:8001`.
3. **Chainlit UI**:
   - The front-end chat interface greets the user based on their role (Patient / Student).
   - Users can type questions or click the paperclip to upload a PDF.
   - Each message is sent to the back-end event handlers, which respond with LLM-driven answers or further instructions (e.g., location prompts for appointment booking).

By combining **Flask** for authentication and session management with **Chainlit** for conversational interaction, we achieve a user-friendly, cohesive experience:

- **Flask**: Manages the database (login, signup, roles) and starts the Chainlit server if needed.
- **Chainlit**: Provides the interactive "chat window," letting users upload documents, ask questions, and receive LLM-powered responses in real time.

4.19 Pinecone Vector Database

As part of our **retrieval architecture**, we leverage the Pinecone vector database to store, manage, and quickly retrieve vector embeddings. The following code snippet illustrates how scraped text data is converted into embeddings and uploaded to Pinecone:

LISTING 4.10: Storing Scraped Data in Pinecone

```
import json
from src.helper import download_hugging_face_embeddings
from langchain.vectorstores import Pinecone
from pinecone import Pinecone as PineconeClient, ServerlessSpec
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

PINECONE_API_KEY = os.environ.get('PINECONE_API_KEY')
PINECONE_API_ENV = os.environ.get('PINECONE_API_ENV')

# 1 Read JSON file (scraped_data_subsections_c.json)
with open('scraped_data_subsections_c.json', 'r', encoding='utf-8')
    as file1:
    data1 = json.load(file1)

# 2 Extract text chunks
text_chunks = []
for item in data1:
    for subsection in item['subsections']:
```

```python
        heading = subsection.get('heading', '')
        content = ' '.join(subsection.get('content', []))
        text_chunks.append(f"{heading}: {content}")

# 3 Obtain embeddings
embeddings = download_hugging_face_embeddings()

# 4 Initialize Pinecone client
pinecone_instance = PineconeClient(api_key=PINECONE_API_KEY)
index_name = "zf"

# 5 Check if index exists, create if not
if index_name not in [index.name for index in pinecone_instance.
   list_indexes()]:
    print(f"Creating index '{index_name}'...")
    pinecone_instance.create_index(
        name=index_name,
        dimension=768,
        metric="cosine",
        spec=ServerlessSpec(cloud="aws", region=PINECONE_API_ENV)
    )

# 6 Upload text chunks to the Pinecone index
docsearch = Pinecone.from_texts(
    text_chunks,
    embeddings,
    index_name=index_name
)

print("Data successfully uploaded to Pinecone!")
```

Detailed Explanation

1. **Loading the JSON Files**
   - The example above specifically references `scraped_data_subsections_c.json`, which contains a list of

items, each with `subsections`. Each `subsection` has a `heading` and `content`.
   - We loop through them, joining any arrays of strings in `content` into a single text string.
   - As a result, each text chunk becomes something like `"Symptoms: shortness of breath, coughing..."`.

2. **Creating `text_chunks`**
   - `text_chunks` is simply a Python list of strings, each representing one segment of the scraped data.
   - If desired, you could further refine or chunk the text if it's large, but in this snippet we store each subsection as a single chunk.

3. **Obtaining Embeddings**
   - `download_hugging_face_embeddings()` is a custom utility from `src.helper`.
   - It likely returns a model or function that can convert text to a fixed-size vector (e.g., 768 dimensions).
   - This step ensures we have a consistent embedding pipeline for any text we want to store or query.

4. **Setting Up Pinecone Client**
   - `PineconeClient(api_key=PINECONE_API_KEY)`: Authenticates with Pinecone using your environment variables.
   - `index_name = "zf"`: The name of our Pinecone index.
   - We check if this index already exists. If not, we create it with:
     - `dimension=768`: Matching the output size of our embedding model.
     - `metric="cosine"`: Cosine similarity for comparing vectors.
     - `ServerlessSpec(cloud="aws", region=PINECONE_API_ENV)`: Specifies a serverless deployment in the indicated AWS region.

5. **Uploading Data via `from_texts`**
   - `Pinecone.from_texts(text_chunks, embeddings, index_name=index_name)`:
     (a) For each string in `text_chunks`, the code uses the provided `embeddings` model to generate a vector.
     (b) These vectors are then batch-uploaded to the `zf` index in Pinecone, each entry typically assigned a unique ID.
   - The returned `docsearch` object can later be used to perform

`docsearch.similarity_search(query)` or related retrieval operations.

Why Pinecone?

- **Scalability**: Pinecone can handle millions of vectors, performing approximate or exact nearest-neighbor searches quickly.
- **Flexibility**: You can store embeddings from different text sources (scraped data, user-provided PDFs, etc.) and query them all under the same index or separate ones if you prefer.
- **Integration with LangChain**: The snippet uses `langchain.vectorstores Pinecone` object to easily unify embedding storage and retrieval logic, tying in seamlessly with the LLM retrieval pipeline.

By uploading the scraped medical data into Pinecone, the system can handle user queries with **retrieval-augmented generation**, bridging newly ingested text (e.g., from external websites or data files) with real-time chatbot interaction.

Chapter 5. Results and Evaluation

In this section, we describe an automated evaluation framework that measures both the **quality of chatbot responses** (via semantic similarity to an expected answer) and the **retrieval effectiveness** of the underlying vector-based search. The code snippet below outlines how queries, expected answers, and reference data are used to generate numerical scores. Additionally, we present the relevant mathematical formulations to clarify how these scores are computed.

5.0.1 Mathematical Foundations

Cosine Similarity. Let $\mathbf{x}$ and $\mathbf{y}$ be two d-dimensional embeddings (vectors) derived from textual inputs (e.g., a *ground-truth* sentence vs. a *chatbot* response). We define the cosine similarity as:

$$\text{cos\_sim}(\mathbf{x}, \mathbf{y}) \;=\; \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|}, \tag{5.1}$$

where $\mathbf{x} \cdot \mathbf{y}$ is the dot product and $\|\mathbf{x}\|$ is the Euclidean norm of $\mathbf{x}$. The result lies in $[-1, 1]$, but for typical sentence embeddings, values are usually in $[0, 1]$.

Generation Similarity Score. For a given user query q with an *expected* answer A, let R denote the chatbot's *response*. We encode both A and R using the same sentence-transformer embedding function $\Phi(\cdot)$, producing $\Phi(A)$ and $\Phi(R)$. The *generation similarity score* $S_{\text{gen}}(q)$ is then:

$$S_{\text{gen}}(q) \;=\; \text{cos\_sim}\Big(\Phi(A),\, \Phi(R)\Big). \tag{5.2}$$

A higher value indicates the chatbot's generated text is semantically closer to the ground-truth.

Retrieval Similarity Score. We also measure how well the system's retriever (e.g., a vector database) surfaces relevant text chunks. For each query q, we fetch the top-k documents $D_1, D_2, \ldots, D_k$ from the retriever. Each document D_i has content that we embed as $\Phi(D_i)$. The *retrieval similarity score* $S_{\text{retr}}(q)$ is defined as the average similarity of each retrieved chunk to the expected answer embedding $\Phi(A)$:

$$S_{\text{retr}}(q) = \frac{1}{k}\sum_{i=1}^{k} \text{cos\_sim}\Big(\Phi(A),\, \Phi(D_i)\Big). \tag{5.3}$$

If $S_{\text{retr}}(q)$ is consistently high, it indicates the retriever is pulling documents semantically aligned with the correct answer.

5.0.2 Evaluation Code

LISTING 5.1: Evaluation Code for Chatbot Responses and Retrieval Quality

```python
import os
import time
import logging
import numpy as np
import plotly.graph_objects as go
from sentence_transformers import SentenceTransformer, util

# Import your LLM chain initialization and retrieval functions.
from app import initialize_llm, long_running_task, docsearch, PROMPT

# Set up logging
logging.basicConfig(level=logging.INFO)

# Load a SentenceTransformer model for semantic similarity evaluation
    .
similarity_model = SentenceTransformer('all-mpnet-base-v2')

def semantic_similarity(expected: str, response: str) -> float:
    """
    Computes the cosine similarity between the expected and response
   text.
    Returns a float between 0 and 1.
    """
    embedding_expected = similarity_model.encode(expected,
   convert_to_tensor=True)
    embedding_response = similarity_model.encode(response,
   convert_to_tensor=True)
```

```python
    cosine_sim = util.pytorch_cos_sim(embedding_expected, embedding_response)
    return cosine_sim.item()

# Large test dataset based on MedlinePlus diseases.
test_data = [
    {
        "query": "What are the common symptoms of asthma?",
        "expected": "Asthma symptoms include wheezing, shortness of breath, chest tightness, and coughing.",
        "reference": "MedlinePlus. Asthma. https://medlineplus.gov/asthma.html"
    },
    ...
]

def get_chatbot_response(query: str, context: str = "") -> str:
    """
    Runs your QA chain for a given query.
    """
    input_data = {"query": query, "context": context}
    llm = initialize_llm()
    response = long_running_task(input_data, llm)
    return response.strip()

def evaluate_retrieval(query: str, expected: str, k: int = 5) -> float:
    """
    Evaluates retrieval quality by fetching the top k documents for a query
    and computing the average semantic similarity between each document's content
    and the expected answer.
    """
    retriever = docsearch.as_retriever(search_kwargs={'k': k})
    docs = retriever.invoke(query)
    if not docs:
        return 0.0
    scores = [semantic_similarity(expected, doc.page_content) for doc in docs]
    return np.mean(scores)

def evaluate_responses():
    """
    Evaluates chatbot responses against expected answers using semantic similarity,
    and measures retrieval quality.
    Returns lists of similarity scores and retrieval scores.
    """
    similarity_scores = []
    retrieval_scores = []

    for test in test_data:
        query = test["query"]
        expected = test["expected"]

        # Retrieve chatbot response.
        response = get_chatbot_response(query, context="")
        sim_score = semantic_similarity(expected, response)
        similarity_scores.append(sim_score)

        # Evaluate retrieval quality using top-5 documents.
        ret_score = evaluate_retrieval(query, expected, k=5)
        retrieval_scores.append(ret_score)

        print(f"Query:                        {query}")
        print(f"Expected Answer:              {expected}")
        print(f"Chatbot Response:             {response}")
        print(f"Semantic Similarity Score (Generation): {sim_score:.2f}")
        print(f"Average Retrieval Similarity Score (top-5): {ret_score:.2f}")
        print("-" * 80)

    return similarity_scores, retrieval_scores

```

```
def plot_evaluation(similarity_scores, retrieval_scores):
    """
    Uses Plotly to plot the semantic similarity scores for generation
    and retrieval,
    and saves the graph as a PNG file.
    """
    test_case_indices = list(range(len(test_data)))

    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=test_case_indices,
        y=similarity_scores,
        mode='lines+markers',
        name='Generation Similarity'
    ))
    fig.add_trace(go.Scatter(
        x=test_case_indices,
        y=retrieval_scores,
        mode='lines+markers',
        name='Retrieval Similarity'
    ))
    fig.update_layout(
        title="Semantic Similarity Scores per Test Case",
        xaxis_title="Test Case Index",
        yaxis_title="Similarity Score",
        template="plotly_white"
    )
    fig.show()
    # Save the graph as a PNG file
    fig.write_image("evaluation_graph.png")

if __name__ == "__main__":
    # Evaluate responses and collect scores.
    sim_scores, ret_scores = evaluate_responses()

    # Plot evaluation metrics using Plotly.
    plot_evaluation(sim_scores, ret_scores)
```

5.0.3 Workflow and Interpretation

1. **Load Test Cases.** Each entry in `test_data` contains:
   - `query`: The user's question.
   - `expected`: A short, correct answer from medical references.
   - `reference`: A citation URL.
2. **Generation Similarity.** The system obtains the chatbot's free-form answer and compares it to the ground-truth (`expected`) using Equation 5.4.

   $$S_{\text{gen}} = \text{cos\_sim}\Big(\Phi(\text{expected}),\ \Phi(\text{response})\Big). \tag{5.4}$$

   A higher S_{gen} indicates the chatbot's answer is semantically close to the reference.
3. **Retrieval Similarity.** Independently, the code checks the top-k retrieved documents to see how aligned they are with the expected text (Equation 5.5):

   $$S_{\text{retr}} = \frac{1}{k}\sum_{i=1}^{k} \text{cos\_sim}\Big(\Phi(\text{expected}),\ \Phi(D_i)\Big), \tag{5.5}$$

   where D_i is the i-th retrieved document chunk. A higher S_{retr} suggests the vector store is returning relevant context.
4. **Visualization.** The function `plot_evaluation` creates a line chart showing both S_{gen} and S_{retr} across all test cases. It also saves this plot as `evaluation_graph.png`.

5.0.4 Significance in the Thesis

- **Quantitative Insight.** By assigning numerical scores to both the final answer (generation) and the retrieved documents, the methodology pinpoints whether errors stem from the *retrieval* stage or the *generation* stage.
- **Scalability.** New questions can be added to `test_data` without retraining the entire model, allowing continuous monitoring of the system's performance.
- **Guidance for Improvement.**
  - If retrieval scores (S_{retr}) are consistently high but generation scores (S_{gen}) are low, the problem may lie in the LLM's prompt or response generation logic.
  - If retrieval scores are low, the embedding or vector store configuration might need refinement.

Overall, this evaluation approach, supported by Equations 5.1, 5.4, and 5.5, provides a rigorous, automated way to measure how effectively the chatbot answers medical queries and how well its retrieval subsystem surfaces relevant evidence from the knowledge base.

Chapter 6. Conclusion and Future Work

In this final chapter, we summarize the key contributions of this thesis and reflect on the outcomes of the research. We presented an AI-driven medical chatbot designed to serve both medical students and patients, integrating advanced natural language processing techniques, embedding-based retrieval, and large language models (LLMs) into a unified system. Our approach combines a robust data ingestion pipeline with role-based interaction, ensuring that each user receives tailored, context-aware responses.

6.1 Conclusion

This thesis has demonstrated a novel framework for topic modeling and conversational AI in the medical domain by addressing critical challenges in both educational and patient support contexts. The main contributions can be summarized as follows:

- **Dual-Mode Chatbot Architecture:** We designed a system that distinguishes between medical students and patients through secure user authentication, role assignment, and role-based logic. This allows the chatbot to deliver in-depth academic explanations to students while providing concise, practical guidance to patients.
- **Robust Data Processing Pipeline:** The implementation includes comprehensive PDF processing, where documents are efficiently parsed, chunked, and preprocessed to facilitate accurate embedding generation.
- **Embedding and Vector Storage:** By leveraging state-of-the-art models (such as `all-mpnet-base-v2`) and integrating with the Pinecone vector database, the system achieves rapid and scalable retrieval of contextually relevant text.
- **Retrieval-Augmented Generation (RAG):** The RAG pipeline effectively combines retrieved text from the vector database with LLM-based response generation. This integration mitigates the risk of hallucination and grounds the chatbot's answers in factual, verifiable sources.
- **Comprehensive Evaluation:** Both quantitative metrics (e.g., cosine similarity, Silhouette Score) and qualitative human evaluations were conducted. The results

show that our approach provides high semantic alignment between expected and generated responses, validating the efficacy of the system.

6.2 Answers to Research Questions

The research questions posed at the outset of this thesis have been addressed as follows:

1. **Supporting Medical Education:** The chatbot provides dynamic, evidence-based answers that bridge theoretical knowledge and practical scenarios, thereby enhancing learning for medical students.
2. **Improving Patient Guidance:** By employing symptom analysis and retrieval-augmented generation, the system offers clear and contextually appropriate medical advice, assisting patients in deciding whether to seek routine or urgent care.
3. **Healthcare Provider Recommendations:** The integration of location- and specialty-based filters enables the system to suggest suitable healthcare providers, thereby streamlining patient referrals.
4. **Autonomous Appointment Management:** The agentic component successfully automates appointment booking by integrating with external APIs, reducing administrative burden and ensuring timely patient care.

6.3 Future Work and Challenges

While the proposed system has shown promising results, several challenges remain and avenues for future research include:

- **Handling Data Noise:** Despite effective preprocessing, some residual noise remains in the data. Future work could explore advanced noise-reduction techniques to further refine input quality.
- **Scalability and Efficiency:** With the current computational resources, certain models had to be downscaled. As more powerful hardware becomes available, future iterations could employ larger, more sophisticated models to improve performance.
- **Multimodal Data Integration:** Incorporating additional data sources (such as medical imaging or sensor data) could provide a more comprehensive understanding of patient conditions and enhance the chatbot's recommendations.

- **Enhanced Prompt Engineering:** Further research into prompt optimization for the LLM may reduce hallucinations and improve the accuracy of generated responses.
- **User Interface Improvements:** Refining the front-end experience—both in the Flask templates and the Chainlit chat UI—could lead to higher user satisfaction and better overall interaction.

Appendix A. Source Code

The Source Code is available on GitHub.

GitHub Link: https://github.com/praveenbm1997/Mapping-Memories-Exploring-Topics-in-Historical-Biographical-Interviews

Bibliography

Python Documentation (2023). Python documentation - the python standard library.