DDx Agent: Complete Project Documentation

Author: Nikita Kumari
Date: July 29, 2025
Version: 1.0.0

1. Project Overview

1.1. Problem Statement

The process of forming a medical differential diagnosis is a complex cognitive task that requires a clinician to recall and synthesize vast amounts of information under pressure. For medical students and junior clinicians, this can be a particularly challenging and error-prone process. This project aims to address this challenge by creating an AI-powered assistant that can help users generate a list of potential diagnoses based on a patient's clinical presentation.

1.2. Proposed Solution

The DDx Agent is a sophisticated tool that leverages a **Retrieval-Augmented Generation (RAG)** architecture to provide evidence-based diagnostic suggestions. Unlike a standard chatbot, which might "hallucinate" or invent information, this agent's responses are grounded in a curated knowledge base of reliable medical texts.

The system runs **100% locally**, ensuring complete data privacy and operating without any API costs. A user can input symptoms in natural language, and the agent will retrieve the most relevant medical information from its knowledge base before using a large language model (LLM) to generate a justified, ranked list of potential diseases.

1.3. Key Features

- **100% Local & Private:** No data ever leaves the user's machine.
- **Evidence-Based:** Responses are grounded in a curated knowledge base, not the LLM's general knowledge.
- **Natural Language Interface:** Users can describe symptoms conversationally.
- **Cost-Free Operation:** No API keys or paid services are required.
- **Interactive UI:** A clean and intuitive web interface built with Streamlit.

2. System Architecture & Pipeline

The agent's architecture is built around a two-phase RAG pipeline: an offline "knowledge base setup" phase and a real-time "live diagnosis" phase.

2.1. Pipeline Diagram

```
graph TD
  subgraph "Phase 1: Knowledge Base Setup (Offline)"
    A[1. Medical Text Files] -->|Documents| B(2. Document Loader);
    B -->|Chunks| C(3. Embedding Model);
    C -->|Vectors| D[4. ChromaDB Vector Store];
  end

  subgraph "Phase 2: Live Diagnosis (Real-time)"
    E[5. User Input] -->|Query| F{6. RAG Agent};
    F -->|"Symptoms query"| G(7. Embedding Model);
    G -->|Query Vector| H(8. ChromaDB Search);
    H -->|"Relevant Context"| F;
    F -->|Context + Query| I(9. Local LLM - Llama 3);
    I -->|Generated Text| J[10. Final Diagnosis];
  end

  style A fill:#f9f,stroke:#333,stroke-width:2px
  style D fill:#f9f,stroke:#333,stroke-width:2px
  style E fill:#bbf,stroke:#333,stroke-width:2px
  style J fill:#bbf,stroke:#333,stroke-width:2px
```

2.2. Pipeline Explanation

1. **Knowledge Base Setup (build_vector_store.py):**
   - **Medical Text Files:** The process begins with a hand-curated collection of .txt files in the data_sources/ folder. Each file contains reliable information about a specific disease.
   - **Document Loading:** LangChain's DirectoryLoader reads these files.
   - **Chunking:** The text is split into smaller, overlapping chunks of ~1000 characters using RecursiveCharacterTextSplitter. This ensures that the LLM receives context that is both focused and coherent.
   - **Embedding:** Each chunk is converted into a numerical vector by a local sentence-transformers model (all-MiniLM-L6-v2). This embedding captures the semantic meaning of the text.
   - **Vector Storage:** The resulting vectors and their corresponding text are stored in a local ChromaDB database, creating a searchable knowledge base.
2. **Live Diagnosis (app.py & agent.py):**
   - **User Input:** The user enters a clinical presentation into the Streamlit web interface.

- **Query Embedding:** The agent takes this input and uses the same embedding model to convert it into a query vector.
- **Semantic Search:** The agent performs a similarity search in the ChromaDB, retrieving the text chunks whose vectors are mathematically closest to the query vector. These are the most relevant pieces of information from the knowledge base.
- **Prompt Augmentation:** The retrieved context chunks are combined with the original user query into a detailed prompt. This prompt instructs the LLM on how to behave and provides it with the necessary evidence.
- **LLM Generation:** The augmented prompt is sent to the local LLM (Llama 3, served via Ollama). The LLM generates a structured differential diagnosis based *only* on the provided context.
- **Display Output:** The final, formatted response is displayed to the user in the Streamlit UI.

3. Data Sources & Exploratory Data Analysis (EDA)

3.1. Data Sources

The knowledge base was manually curated from the following public and reliable medical information sources:

- The Centers for Disease Control and Prevention (CDC)
- The World Health Organization (WHO)
- The Merck Manual (Online Public Version)
- The National Institutes of Health (NIH)

The initial corpus consists of information on 20 common diseases, with each disease's text stored in a separate file.

3.2. Exploratory Data Analysis

An EDA was performed in the EDA.ipynb notebook to understand the characteristics of the text corpus. Key findings include:

- **Document Length:** Most disease descriptions fall between 100 and 200 words, indicating a relatively consistent level of detail across the corpus.
- **N-gram Analysis:** Bigram (two-word phrase) analysis confirmed that key medical symptoms like "sore throat," "body aches," and "shortness of breath" were among the most frequent phrases, validating the quality of the source text.
- **TF-IDF Analysis:** This technique successfully identified the most discriminating keywords for each disease. For example, "erythema" and "bull's-eye" had high TF-IDF scores for Lyme Disease, confirming their diagnostic importance.

- **Embedding Space Visualization (t-SNE):** A t-SNE plot of the document embeddings showed clear clustering of related diseases. For instance, respiratory illnesses like Influenza, COVID-19, and Bronchitis were grouped closely together, visually confirming that the embedding model successfully captured the semantic relationships within the medical data.

4. Setup and Usage Guide

4.1. Technology Stack

- **AI Framework:** LangChain
- **LLM Server:** Ollama
- **Language Model:** llama3:8b
- **Embedding Model:** sentence-transformers/all-MiniLM-L6-v2
- **Vector Database:** ChromaDB
- **Web UI:** Streamlit
- **Dependencies:** pandas, nltk, scikit-learn, unstructured

4.2. Local Installation and Execution

1. **Clone the repository:**
   ```
   git clone https://github.com/your-username/ddx-agent.git
   cd ddx-agent
   ```

2. **Set up the Python environment:**
   ```
   python -m venv venv
   source venv/bin/activate  # On Windows: venv\Scripts\activate
   pip install -r requirements.txt
   ```

3. **Install and Run Ollama:**
   - Download and install Ollama.
   - Launch the Ollama application. It must be running in the background.
   - Pull the Llama 3 model from your terminal:
     ```
     ollama pull llama3:8b
     ```

4. **Build the Vector Store:**
   - Ensure your .txt files are in the data_sources/ directory.
   - Run the script to build the knowledge base:
     ```
     python build_vector_store.py
     ```

5. **Launch the Application:**
   ```
   streamlit run app.py
   ```

5. Evaluation

5.1. Methodology

A preliminary evaluation was conducted to assess the agent's performance. A small test suite of 10-15 clinical vignettes was created in evaluation_set.csv, with each case having a clear, expected diagnosis. The evaluate.py script automates the process of running each vignette through the agent and checking if the expected diagnosis is present in the generated output.

5.2. Results

The evaluation serves as a baseline for the agent's accuracy. The primary metric was **Top-1 Accuracy**: whether the correct diagnosis was mentioned in the agent's response. This simple metric provides a clear signal of the system's ability to retrieve relevant context and generate a plausible answer. Further work would involve more nuanced metrics like Mean Reciprocal Rank (MRR) to evaluate the ranking of the correct diagnosis.

6. Limitations & Future Work

6.1. Limitations

- **Limited Knowledge Base:** The agent's knowledge is strictly confined to the documents provided. It cannot diagnose any disease not included in the data_sources folder.
- **No Understanding of Negation/Context:** The current retrieval method is based on semantic similarity and may not fully grasp complex clinical context, such as negated symptoms ("patient has no fever") or family history.
- **Static Knowledge:** The knowledge base does not update automatically and can become outdated.

6.2. Future Work

- **Expand Knowledge Base:** Systematically add more diseases, including rarer conditions, to the corpus.
- **Advanced Retrieval Strategies:** Implement more sophisticated retrieval methods like HyDE (Hypothetical Document Embeddings) or a re-ranking model to improve the quality of retrieved context.
- **Incorporate Structured Data:** Modify the agent to accept and interpret structured lab results (e.g., from a CSV file) in addition to text symptoms.
- **Conversational Memory:** Add a memory component to allow for multi-turn,

follow-up questions, more closely mimicking a real diagnostic conversation.

7. Disclaimer

This is an academic and portfolio project designed to demonstrate RAG architecture with local models. It is **NOT** a medical device and must not be used for actual medical diagnosis, advice, or treatment.