Hanoi University of Science and Technology

School of Information and Communication Technology

Introduction to Business Analytics Report

Chatbot Using RAG and AutoCrawl

Supervisor

Prof.Tran Van Dang

Students

Nguyen Thanh Long - Student's ID : 20214912
Nguyen Minh Cuong - Student's ID : 20210140
Doan Ngoc Cuong - Student's ID : 20210141
Luu Anh Duc - Student's ID : 20204875

Ha Noi, 12/2024

Contents

Abstract

Our project integrates Retrieval-Augmented Generation (RAG) and automated web crawling to create an intelligent, real-time conversational agent. AutoCrawl collects and updates knowledge dynamically, embedding it into a vector database, while RAG retrieves, and synthesizes responses using a large language model (LLM). This architecture addresses the limitations of traditional chatbots by providing accurate, contextually relevant, and up-to-date answers. The solution offers a scalable approach to real-time question answering, combining dynamic data collection with advanced retrieval-based AI.

1 Introduction

1.1 Overview of the Project

This project focuses on developing a sophisticated chatbot using Retrieval-Augmented Generation (RAG) and AutoCrawl technology. The chatbot integrates data crawling and processing pipelines with a retrieval-based architecture to generate accurate and contextually relevant responses. The key architecture leverages an AutoCrawl mechanism to collect data from the web, embeds this data into a vector database, and employs RAG to fetch, rerank, and synthesize responses using a large language model (LLM).

By combining the strengths of modern natural language processing techniques and automated data collection, the chatbot serves as an intelligent conversational agent, providing users with real-time, precise answers to their queries.

1.2 Motivation

The rapid growth in information volume on the internet has created a demand for tools that can effectively retrieve and utilize knowledge in a meaningful way. Traditional chatbots often fail to provide accurate or up-to-date answers due to limited datasets or static knowledge bases. The motivation for this project stems from the need to bridge this gap by integrating real-time web crawling with RAG, which enhances the chatbot's ability to deliver reliable information dynamically.

Key objectives include:

- Developing a scalable solution for real-time question answering.
- Enhancing user interaction by ensuring up-to-date and relevant answers.

- Exploring the synergy between AutoCrawl and RAG to improve retrieval-based AI systems.

2 Background and Related Work

2.1 Retrieval-Augmented Generation (RAG)

RAG is an emerging paradigm in conversational AI that combines retrieval-based methods with generative models. Unlike traditional generative models that rely solely on pre-trained datasets, RAG retrieves relevant information from external databases or knowledge sources, ensuring that responses are accurate and contextually grounded. The architecture involves:

1. Retrieval: Fetching relevant documents or data chunks from a database.
2. Generation: Using an LLM to generate responses based on the retrieved content.

2.2 AutoCrawl Technology

The AutoCrawl system automates the process of web data collection and processing using Python libraries and AI tools. It is designed to ensure efficient, scalable, and adaptive data extraction from web pages while preprocessing and preparing the data for further use, such as embedding into vector databases. Below is the detailed description of the technology :

Key Features of AutoCrawl

1. **Automation**:
   - AutoCrawl uses Python-based automation for navigating and extracting content from web pages dynamically.
   - Incorporates BeautifulSoup for parsing HTML and API integrations (e.g., Generative AI) to process text.
2. **Scalability**:
   - Designed to handle recursive crawling across multiple levels of web pages, adhering to user-defined depth and link limits.
   - Can process multiple URLs in parallel, making it suitable for large-scale data collection tasks.

3. **Dynamic Adaptation**:
   - Automatically adapts to different web structures by validating and normalizing links.
   - Ensures robustness even if the layout or structure of a target website changes.
4. **Data Preprocessing**:
   - Cleans and refines the collected text using AI to remove redundant elements, punctuation, and irrelevant words.
   - Outputs structured and ready-to-use JSON data.

2.3 Related Work

Several advancements have influenced this project:

- **Conversational AI**: Projects like OpenAI's ChatGPT focus on generative dialogue but often lack integration with real-time data retrieval.
- **Web Crawling in NLP**: Previous works have utilized tools like Scrapy and BeautifulSoup for data collection, although these often lack seamless integration with retrieval-augmented frameworks.
- **Knowledge Bases**: Platforms like Google Knowledge Graph rely on static datasets, limiting their ability to answer time-sensitive or evolving queries.

By combining these advancements, the chatbot in this project addresses the limitations of both static knowledge bases and generative-only systems, leveraging a hybrid architecture for enhanced performance.

3 Methodology

3.1 System Architecture

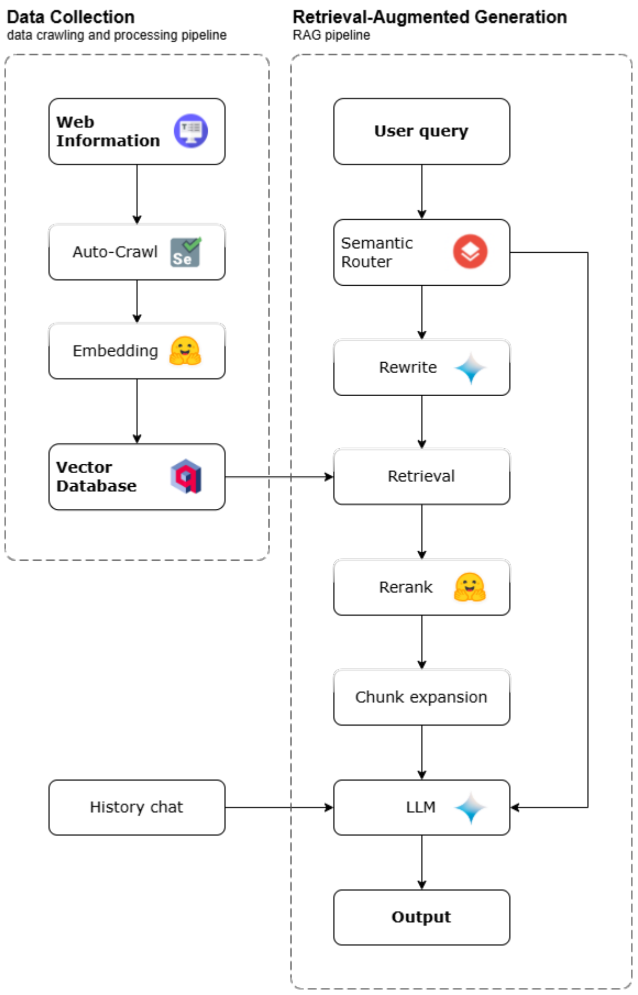

Figure 1: Architecture

3.2 Data Collection and Processing

The code provided is a Python-based solution designed for automated data collection and processing from web pages within the same domain. It systematically extracts and refines content to create a structured dataset. Below is a summary of the approach:

Data Collection

1. **URL Validation**: The program ensures that only URLs belonging to the same domain as the starting URL are processed. This validation is performed using the `is_valid_url` function.

2. **Content Extraction**:
    - Text content is retrieved from key HTML tags such as `h1`, `h2`, `p`, and `article` using the BeautifulSoup library.
    - Relevant links are extracted from the webpage for further crawling, adhering to a user-defined maximum number of links and depth.
3. **Crawling Mechanism**: The system recursively visits valid URLs, following links extracted from the content. This process continues until the specified depth or link limit is reached.
4. **Efficiency Measures**: A set is maintained to track visited URLs and prevent duplicate processing.

Data Processing

1. **Content Refinement**: The extracted text is sent to a Generative AI model (`gemini-1.5-flash`) via Google's API for cleaning. The model removes redundant punctuation and irrelevant words without summarizing the content.
2. **Output Formatting**:
    - Each processed URL and its corresponding refined content are stored in a JSON file.
    - The file structure ensures that data is easy to access and analyze.
3. **Error Handling**: The program implements exception handling to manage potential issues such as timeouts or invalid responses.

Final Dataset

- The resulting dataset includes:
    1. URLs of the crawled pages.
    2. Cleaned and structured content extracted from those pages.
- These results are stored in a JSON file for subsequent use in research or analysis.

Summary Metrics At the end of the process, the program reports:

1. Total number of links crawled.
2. Total characters of text content collected and refined.

This approach ensures the data collected is relevant, consistent, and formatted for further analysis while maintaining efficiency and scalability.

3.3 Indexing and Vector Search

After being saved in a JSON file, the data is further divided into smaller chunks, each containing approximately 150–350 tokens. This process utilizes a semantic chunking method to enhance the semantic coherence of the chunks. The method begins by splitting the text into sentences and embedding each sentence into dense vectors. A sliding window is then applied across these sentences. Chunk boundaries are determined based on specific conditions, such as when the semantic similarity between adjacent windows falls below a predefined threshold or when the maximum token length is reached. This ensures that the resulting chunks are semantically meaningful and consistent in length.

The generated chunks are saved along with references to their source documents for traceability. Subsequently, each chunk is embedded into a dense vector representation using a Sentence Transformer model, producing vectors of size 768. These embeddings serve as the foundation for semantic search and retrieval in downstream tasks.

3.4 RAG Pipeline

The architecture of the Retrieval-Augmented Generation (RAG) system follows a modular design to process user queries and generate contextually appropriate responses. It begins by deciding whether the user's query needs to be queried or not through the semantic router. If it doesn't need to be queried, the router will send the query directly to the LLM to generate the response. Then,the user's query is rewritten using a Large Language Model (LLM) to enhance clarity and effectiveness, facilitating a more precise retrieval of relevant information. Semantic search is employed to retrieve information from a vector database, with future integration plans for sparse search and knowledge graphs to enhance performance. The retrieved chunks are then reranked for relevance using the ViRanker model and expanded to include neighboring chunks to improve contextual understanding.

The expanded information is passed to an LLM (Google Gemini API), which synthesizes the data along with the conversation history to generate a coherent and context-aware response. The system's modular flow enables dynamic enhancements and scalability while maintaining high-quality outputs.

3.4.1 Semantic Router

The Semantic Router serves as a key component for determining whether a user query requires information retrieval or not. Its primary goal is to enhance response time and improve user satisfaction by efficiently handling queries. Two approaches were considered for this module.

The first approach involves using a Sentence Transformer to embed predefined common greetings (e.g., "hello," "my name is," etc.) into vector representations. When a user submits a query, its embedding is compared against these vectors by calculating the average distance. If the distance exceeds a predefined threshold, the query is classified as a common greeting; otherwise, it is flagged for retrieval.

The second approach leverages a prompt engine powered by the same Large Language Model (LLM) used for generating the final response (Google Gemini API). This engine uses the LLM's advanced reasoning capabilities to analyze the user query and decide whether retrieval is necessary.

Ultimately, the second approach was selected due to the diverse nature of greeting expressions, which can vary significantly across contexts and users. While the first method is suitable for systems with tightly controlled and well-curated data, the second approach offers greater flexibility and accuracy in handling dynamic user inputs.

3.4.2 Rewrite Query

The Rewrite Query module is employed to reformulate the original user query. The primary objective of this module is to improve the clarity, specificity, and overall effectiveness of the query, which can improve the quality of the retrieval process that follows.

To address this challenge, the approach taken involves the use of a prompt engine powered by the same Large Language Model (LLM) that is used for generating the final answer (google gemini api). The prompt engine leverages the LLM's capabilities to generate a refined version of the query, which is then passed to the retrieval module. However, the main drawback of this approach is that it requires an additional API call to the LLM, which can increase latency and computational costs.

3.4.3 Rerank Chunks

The Rerank Chunks module is responsible for improving the quality of the retrieved chunks by reordering them based on their relevance to the query.

In this project, the reranking process is performed using the namdp-ptit/ViRanker model, which has approximately 600 million parameters. The approach involves first expanding the set of retrieved chunks to twice the retrieval number if rerank is not use (i.e., from k to 2k chunks). Each of these chunks is then paired with the query, and the ViRanker model is run on each tuple (chunki,query). The model scores each pair of chunk-query based on relevance, and the top k chunks, according to these scores, are selected and reordered top k chunks for the next step in the pipeline.

However, this approach also comes with its drawbacks. The most notable is the increased latency and computational cost associated with reranking. Running the ViRanker model on each chunk-query pair, especially with the expanded set of chunks, requires additional processing time and computational resources.

3.4.4 Chunks Expansion

The Chunks Expansion module plays a pivotal role in enriching the context of each retrieved chunk by incorporating additional neighboring chunks. This process ensures that the selected chunks contain sufficient context to provide a comprehensive and coherent response in the subsequent steps.

In this project, each retrieved chunk is expanded by including up to four of its neighboring chunks — specifically, two chunks to the left and two chunks to the right. This expansion process allows the system to capture surrounding context that may not be present in the initially retrieved chunk alone. By aggregating adjacent chunks, the system can provide a more complete picture of the information, which is particularly useful for understanding the broader context in complex queries.

While this expansion increases the richness of the context, it also comes with potential downsides. The inclusion of additional chunks can increase the amount of data that needs to be processed in subsequent steps, which might lead to higher computational costs and longer processing times. However, these drawbacks are often outweighed by the benefits of improved contextual accuracy, particularly in cases where context is crucial to generating a meaningful response.

3.5 Implementation Details

The chatbot system is implemented with three primary components: **Backend**, **Frontend**, and **Deployment**, ensuring modularity, scalability, and seamless integration.

3.5.1 Backend Implementation

The backend is the core of the system, managing data collection, processing, querying, and generating intelligent responses. It is built using Flask and integrates various technologies for efficiency.

Technologies and Frameworks

- **Flask**: Web framework for building API endpoints.
- **Qdrant**: Vector database for storing and retrieving dense embeddings.
- **Google Gemini API**: For query rewriting and response generation using a large language model (LLM).
- **Beautiful Soup**: For web crawling and content extraction.
- **LangChain**: For orchestrating the Retrieval-Augmented Generation (RAG) pipeline.

Key Features

1. **Crawling and Content Processing**
   - **Crawling (`/crawl`)**: Accepts a URL, company name, and max link depth. It uses threading for asynchronous crawling to avoid blocking the main API.
   - **Crawl Status (`/crawl_status`)**: Provides real-time feedback on the progress of the crawling process by reading a status file.
2. **Database Management**
   - **Save Data to Database (`/save_db`)**: Reads raw crawled data, semantically chunks it, and inserts the chunks into a Qdrant collection.
   - **Get Collections (`/get_collections`)**: Retrieves a list of all available collections in the Qdrant database.
   - **Switch Collection (`/change_collection`)**: Dynamically switches the active Qdrant collection for queries.
3. **Query Handling**
   - **Send Message (`/send_message`)**: Processes user questions using the RAG pipeline. Retrieves relevant chunks, reranks them, and generates responses with the Google Gemini API.

4. **Health Check**

    - **Health (`/health`)**: Provides a simple endpoint to verify the health of the backend system.

3.5.2 Frontend Implementation

The frontend provides a user-friendly interface for interacting with the chatbot system. It is designed using HTML, CSS, and JavaScript, focusing on responsiveness and interactivity.

Technologies and Frameworks

- **JavaScript**: Handles user interaction and API integration.
- **React.js (Optional)**: Used for building dynamic single-page applications.
- **Bootstrap**: Ensures responsive and visually appealing design.

Key Features

1. **Chat Interface**

    - **User Interaction**: Accepts user input via a text field or ”Enter” key and displays bot responses.
    - **Message Display**: Formats bot responses using Markdown and embeds clickable links for related content.
    - **Close Button**: Allows users to minimize the chatbot interface.

2. **Dynamic Collection Management**

    - **Fetch Collections**: Populates a dropdown menu with available collections from the backend.
    - **Switch Collection**: Enables switching between active collections for dynamic context changes.

3. **Web Crawling**

    - **Start Crawl**: Triggers web crawling with a specified URL, company name, and link depth.
    - **Verify Crawl Status**: Displays real-time updates on the crawling process.

4. **Save Data**

- **Chunk and Save**: Sends a request to save processed crawled data into the database and displays feedback.

5. **Error Handling**
    - Displays detailed error messages for invalid inputs or backend issues in real time.

3.5.3 Deployment

The deployment ensures scalability, security, and high availability. Both backend and frontend are containerized for consistent environments across development and production.

Technologies and Tools

- **Docker**: Containerization for both backend and frontend components.
- **Docker Compose**: Orchestrates multi-container setups.
- **Cloud Platforms**: AWS, Google Cloud, or Azure for hosting the application.

Deployment Steps

1. **Backend Deployment**
    - Containerized using a `Dockerfile`.
    - Flask application exposed on port 3000 for API requests.
    - Environment variables (`.env`) configured for secure API keys and database URLs.
    - Orchestrated with a Qdrant container for the vector database.
2. **Frontend Deployment**
    - Static files hosted on platforms like Netlify or Vercel.
    - Configured for environment-specific backend URLs.
    - Served over HTTPS for secure user interactions.
3. **Networking and Security**
    - Enforced HTTPS for secure communication between frontend, backend, and external APIs.
    - API keys for Google Gemini and Qdrant securely handled through environment variables.

4. **Scalability**
    - Horizontal scaling for backend and Qdrant containers based on load.
    - Caching frequently accessed data to reduce redundant API calls.

4 Experimental

The system is evaluated based on the accuracy of its final answers. Specifically, the evaluation process involves accessing the target link (which is later used for automatic crawling) and identifying other related links within the same domain. Questions are generated based on the information from these links. To assess the correctness of the system's responses, an LLM is used to validate the answers, outputting a true/false result. Additionally, the crawl module is evaluated by checking whether the links generated from each question are present in the database.

The evaluation parameters mirror those used in real-world scenarios. For each link, a maximum of 50 related links (max_links) are retrieved and considered in the evaluation process.

Link	**Number of Questions**	**Correct Crawls**	**Correct Answers**
https://cmccloud.vn/	30	24	24
https://cmctelecom.vn/	20	12	10
Total	50	36	34

5 Conclusion and Future Work

5.1 Conclusion

The project successfully implemented a Retrieval-Augmented Generation (RAG) system integrated with an AutoCrawl pipeline to address dynamic information retrieval and response generation. Key accomplishments of the project include:

- **Efficient Data Collection:** The AutoCrawl pipeline automated the collection and embedding of relevant information into a vector database.
- **Effective Retrieval and Response Generation:** The combination of semantic search, multiple modules such as rewrite, rerank, and response synthesis using the Google Gemini API provided accurate and context-aware responses to user queries.

- **Scalability:** The modular architecture supports the integration of future enhancements, such as sparse search and knowledge graph-based retrieval.

5.2 Future Work

While the current RAG system demonstrates promising results, several areas have been identified for further development and improvement. These enhancements focus on refining existing modules and exploring additional functionalities to optimize the system's performance.

- **Enhancing Data Filtering in AutoCrawl:** Improve the data collection process by implementing advanced filtering techniques to exclude irrelevant or low-quality content, ensuring higher precision in retrieval results.
- **Rerank Model Fine-Tuning:** One observed challenge is that the rerank model, namdp-ptit/ViRanker, occasionally performs suboptimally, particularly in specific contexts or query types. To address this, a potential avenue for improvement is to fine-tune the rerank model using company-specific data.
- **Additional Modules - Query Decomposition:** Finally, expanding the system's capabilities by integrating additional modules such as Query Decomposition could offer further enhancements. Query Decomposition involves breaking down complex queries into simpler sub-queries, which can be processed individually before aggregating the results. This approach can improve the system's ability to handle complex, multi-faceted queries and provide more accurate and detailed responses.

These proposed enhancements aim to improve the system's robustness, usability, and scalability, making it more versatile for real-world applications.

References

[1] Aavache. Llm-based web crawler. https://github.com/Aavache/LLMWebCrawler/tree/main, 2023.

[2] Bangoc123. Vietnamese retrieval backend: Rag + mongodb + gemini 1.5 pro + semantic router + reflection. https://github.com/bangoc123/retrieval-backend-with-rag, 2024.

[3] Kai Zhang Shiwei Tong Qi Liu Hao Yu, Aoran Gan and Zhaofeng Liu. Evaluation of retrieval-augmented generation: Asurvey. https://arxiv.org/pdf/2405.07437, 2024. Accessed: 2024-12-24.

[4] Ngo Hieu. Halongembedding: A vietnamese text embedding, 2024.

[5] Nam Dang Phuong. Viranker: A cross-encoder model for vietnamese text ranking, 2024.

[6] Meng Wang Haofen Wang Yunfan Gao, Yun Xiong. Modular rag: Transforming rag systems into lego-like reconfigurable frameworks. https://arxiv.org/pdf/2407.21059, 2024.