The ChrysisRagDeepThought Project: Design, Implementation, and Evaluation of an Advanced RAG System for University of Cyprus Information Retrieval

Author: Chrysis

Abstract: This report details the "ChrysisRagDeepThought_final" project, which focused on the design, implementation, and rigorous evaluation of an advanced Retrieval Augmented Generation (RAG) system. The system is tailored for information retrieval from a corpus of documents pertaining to the University of Cyprus. We discuss the multi-stage data preprocessing pipeline, the core RAG architecture including data ingestion, chunking, embedding, vector indexing with FAISS, and LLM-based answer generation. A significant component of this project was the development of a synthetic Question-Answer (QA) dataset using k-means clustering for representative chunk selection, which served as the basis for our evaluation. The report provides a thorough justification for methodological choices, including model selection and evaluation metrics. Finally, we present and analyze the evaluation results from two distinct test sets: one with automatically generated questions and another with manually curated "good questions," demonstrating the impact of question quality on RAG system performance.

1. Introduction

The "ChrysisRagDeepThought" project was initiated with the primary objective of developing a robust and efficient Retrieval Augmented Generation (RAG) system specialized for querying information about the University of Cyprus (UCY). In an academic environment, timely access to accurate and relevant information is crucial for students, faculty, and administrative staff. Traditional search methods often fall short in providing concise, context-aware answers, necessitating a more advanced approach.

The motivation behind this project stems from the increasing volume and complexity of information available on university websites and related documents. Navigating this vast dataset to find specific answers can be a time-consuming and often frustrating experience. Our RAG system aims to alleviate this by leveraging the power of large language models (LLMs) combined with a targeted retrieval mechanism to provide direct answers based on the UCY-specific corpus.

A narrative element, the "DeepThought" persona, was woven into the project, drawing inspiration from Douglas Adams' "The Hitchhiker's Guide to the Galaxy." In the novel, DeepThought is the supercomputer famed for providing the answer "42" to the "Ultimate Question of Life, The Universe, and Everything," though the actual question remains humorously unknown. This persona serves as an allegory for our RAG system's aspiration: to provide definitive answers regarding the University of Cyprus. Our evaluation process playfully incorporates this theme. The generation of a test set of 42-questions directly references DeepThought's iconic answer. This not only provides a concrete benchmark for assessing system performance but also captures the spirit of the persona, which might declare, "I am DeepThought... I am ready for more!" and encourages users to "keep asking questions until you find the ULTIMATE QUESTION!"

This report aims to provide a comprehensive technical overview of the ChrysisRagDeepThought project. We will meticulously detail the system's architecture, the data preprocessing pipeline, the core RAG implementation with a special focus on the synthetic QA dataset generation process using k-means clustering, and the rationale behind our design choices. Furthermore, we will present a detailed analysis of the evaluation results derived from different test sets, interpreting their significance. The report concludes by summarizing our findings and suggesting potential avenues for future development.

2. System Architecture

The ChrysisRagDeepThought system is an end-to-end information retrieval pipeline designed to process, understand, and query documents related to the University of Cyprus. Its architecture can be broadly categorized into three major interacting components: Data Preprocessing, the Core RAG Engine & QA Dataset Generation, and an Interactive Querying and Evaluation Framework.

High-Level Overview: The workflow begins with the **Data Preprocessing** stage (`chrysisRagPreprocess` directory), where raw data (links to UCY web pages) is collected, converted into a usable format (Markdown), cleaned, and filtered to retain substantial documents. This curated dataset then feeds into the **Core RAG Engine & QA Dataset Generation** stage (`chrysisRagDeepThought_pre-run` directory). Here, documents are ingested, chunked, and their embeddings are generated. These embeddings are used to build a FAISS vector index for efficient retrieval. This stage also encompasses a crucial sub-process: the generation of a synthetic Question-Answer (QA) dataset. This dataset is created by first selecting representative document chunks using k-means clustering, then employing an LLM to generate QA pairs from these chunks, and finally augmenting these pairs with relevant context. The generated QA dataset is pivotal for the system's evaluation. The **Interactive Querying and Evaluation Framework** (primarily realized through Python scripts in `chrysisRagDeepThought_pre-run` and orchestrated by the Colab notebooks) allows for real-time interaction with the RAG system and provides a systematic means to assess its retrieval and generation performance using the synthetic QA dataset.

Major Components:

1. **Data Preprocessing (`chrysisRagPreprocess`):**
   - **Objective:** To transform raw web data into a clean, structured, and relevant corpus of Markdown documents.
   - **Modules:** Scripts for fetching HTML from URLs, converting HTML to Markdown, cleaning the Markdown (removing headers, footers, irrelevant sections), and filtering for sufficiently long documents.
2. **Core RAG Engine & QA Dataset Generation (`chrysisRagDeepThought_pre-run`):**
   - **Objective:** To implement the RAG pipeline for answering queries and to create a high-quality synthetic dataset for evaluation.
   - **Modules:**
     - `rag_core.py` : Encapsulates the fundamental RAG functionalities – initializing the embedding model and LLM client, building the vector index, retrieving relevant document chunks, and generating answers.
     - `1_dataset_pipeline.py` : Orchestrates the creation of the synthetic QA evaluation dataset. This involves document loading, chunking, embedding, k-means clustering for diverse chunk selection, LLM-based QA pair generation, and ranking of relevant chunks for each QA pair.
     - `utils.py` : Provides utility functions for text preprocessing, chunking, and document loading, shared across different modules.
     - `config.py` : A central configuration file storing parameters like model names, file paths, and operational settings (e.g., chunk size, retrieval K).
3. **Interactive Querying and Evaluation Framework:**
   - **Objective:** To enable live interaction with the RAG system and to systematically evaluate its performance.
   - **Modules:**
     - `2_interactive_agent.py` : Provides a command-line interface for users to ask questions and receive answers from the RAG system.
     - `3_evaluate_retrieval.py` : Evaluates the retrieval component of the RAG system using the generated QA dataset, calculating metrics like Hit Rate, MRR, Precision, and Kendall's Tau.
     - `4_evaluate_generation.py` : Evaluates the answer generation component by comparing system-generated answers against ground-truth answers from the QA dataset, primarily using semantic similarity.
     - `5_visualize_evaluation.py` : Generates plots from the evaluation results for better understanding and presentation.
   - **Orchestration:** The Colab notebooks (`chrysisRagDeepThought_colab_...ipynb`) serve as the primary environment for running these scripts in sequence and observing their outputs.

Workflow: The process starts with `chrysisRagPreprocess_colab.ipynb` executing scripts in `chrysisRagPreprocess/` to produce cleaned Markdown files in `long_files/` . These files become the input for the `chrysisRagDeepThought_colab...ipynb` notebooks, which run the scripts in `chrysisRagDeepThought_pre-run/` . `1_dataset_pipeline.py` first initializes the RAG core (using `rag_core.py` and `config.py`) to build an index and then uses this setup, along with an LLM, to create `generated_qa_dataset_ranked.xlsx` . Subsequently, `3_evaluate_retrieval.py` and `4_evaluate_generation.py` use this QA dataset and the initialized RAG agent to produce evaluation metrics, which are then visualized by `5_visualize_evaluation.py` . The `2_interactive_agent.py` script can be run independently to query the initialized RAG agent.

3. Data Preprocessing (`chrysisRagPreprocess`)

The initial stage of our project, encapsulated within the `chrysisRagPreprocess` directory and orchestrated by `chrysisRagPreprocess_colab.ipynb` , was dedicated to transforming raw data from the University of Cyprus web domain into a clean, structured, and relevant corpus suitable for ingestion by our RAG system. The primary objective of this preprocessing phase was to ensure that the data fed into the RAG pipeline was of high quality, free from irrelevant noise (like navigation bars, footers, and advertisements), and formatted consistently.

Pipeline: The preprocessing pipeline involved a sequence of operations, executed by Python scripts:

1. **`01_linksToHtml.py` (using `01_links.csv`):**
   - **Input:** A CSV file (`01_links.csv`) containing a list of URLs pointing to various pages on the University of Cyprus website and related domains (e.g., `datascience.cy` , `globed.eu`).
   - **Functionality:** This script was designed to iterate through the URLs in the CSV, fetch the HTML content from each link, and save it as an individual HTML file. To handle potential IP blocking or rate limiting, this script was configured to use Bright Data proxies (credentials managed via environment variables). It also included timeout settings and error handling for robustness. The `sanitize_filename` function within this script converts URLs into valid, unique filenames for the saved HTML files, which were stored in an `01_html_files/` directory.
   - **Output:** A collection of HTML files, each corresponding to a URL from the input CSV.
2. **`02_htmlToMD.py` :**
   - **Input:** The directory containing HTML files generated by `01_linksToHtml.py` (`01_html_files/`).
   - **Functionality:** This script utilizes the `markdownify` library to convert the HTML content of each file into Markdown format. Markdown was chosen for its simplicity, readability, and ease of further text processing. The `heading_style="ATX"` option

ensures that headers are formatted with `#` symbols.
- **Output:** A corresponding set of Markdown files, typically stored in a `02_md_files/` directory.

3. **`03_clean_md.py`:**
   - **Input:** The directory of Markdown files generated by `02_htmlToMD.py` (e.g., `02_md_files/`).
   - **Functionality:** This script performs a crucial cleaning step. As observed in the script's logic, the primary cleaning strategy involves identifying the first H1 header (`# `) in each Markdown file and considering content from that point onwards. It then attempts to identify common footer elements (e.g., "Contact Us," copyright notices, "Privacy Overview") using a predefined list of regex patterns (`contact_header_pattern`, `copyright_pattern`, etc.). Content appearing after the first detected footer pattern is discarded. This aggressive approach aims to remove boilerplate headers, navigation menus that often precede the main content, and common footers, thereby isolating the core informational content of each page. The script also normalizes whitespace by reducing multiple consecutive blank lines to a single one.
   - **Output:** Cleaned Markdown files, stored in a `03_cleaned_md_files/` directory. Files where no H1 header was found or that became empty after cleaning would result in empty output files.
4. **`04_long_files.py`:**
   - **Input:** The directory of cleaned Markdown files from `03_clean_md.py` (e.g., `03_cleaned_md_files/`).
   - **Functionality:** This script filters the cleaned Markdown files based on their content length. It counts the number of words in each file and copies only those files that meet a minimum word count threshold (defaulting to 100 words as specified in the script) to a new directory. This step ensures that only substantially informative documents are retained for the RAG system, excluding very short or empty pages that might have resulted from the cleaning process or were inherently sparse.
   - **Output:** A curated collection of Markdown files containing substantial content, saved into the `long_files/` directory. This directory serves as the primary data source (`DOCS_DIRECTORY` in `config.py`) for the subsequent RAG pipeline.

This multi-step preprocessing pipeline was essential for preparing a high-quality, focused corpus from diverse web sources, significantly impacting the subsequent effectiveness of document chunking, embedding, and retrieval within the ChrysisRagDeepThought system.

4. Core RAG Implementation & QA Dataset Generation (`chrysisRagDeepThought_pre-run`)

The `chrysisRagDeepThought_pre-run` directory houses the heart of our project: the core RAG engine and the pipeline for generating a synthetic Question-Answer (QA) dataset used for evaluation. These components are orchestrated by a series of Python scripts, configured through `config.py`, and leverage functionalities from `rag_core.py` and `utils.py`.

4.1. `1_dataset_pipeline.py`: Synthetic Evaluation Dataset Creation

This script is pivotal for creating a robust evaluation framework for our RAG system. It automates the generation of a QA dataset derived directly from our UCY document corpus. The process ensures that evaluation questions are relevant to the indexed content and that ground-truth answers are based on specific document chunks.

- **Document Ingestion & Preparation:**
  - The pipeline begins by loading documents from the `DOCS_DIRECTORY` specified in `config.py` (which is `./long_files/`, the output of our preprocessing stage). This is handled by the `load_documents` function from `utils.py`. This function iterates through markdown files, reads their content, and performs initial text preprocessing using `preprocess_text` from `utils.py` to normalize whitespace.
  - The loaded text is then chunked. We employed the `SentenceSplitter` from `llama_index.core.node_parser` (as used in `utils.py`'s `split_text_into_chunks` function). The `CHUNK_SIZE` parameter, set to 200 characters in `config.py`, and a default `chunk_overlap` of 20 characters (within `split_text_into_chunks`) dictate how documents are segmented. This strategy aims to create manageable, semantically coherent chunks suitable for embedding and retrieval. The Colab logs indicate that this process yielded 1002 chunks from 72 markdown files.
- **Embedding and Indexing (Leveraging `rag_core.py`):**
  - Before QA generation, the script initializes the RAG agent core components via `rag_core.initialize_agent`. This function loads the specified embedding model (`EMBEDDING_MODEL = 'intfloat/multilingual-e5-large-instruct'` from `config.py`) and the `Together` client for LLM access.
  - It then either loads an existing FAISS index if `INDEX_FILE` (`indexE5.faiss` from `config.py`) is found or builds a new one. The `build_index` function in `rag_core.py` is responsible for this. It takes the document chunks, encodes them using the loaded `SentenceTransformer` model, normalizes the resulting embeddings (to ensure that inner product similarity is equivalent to cosine similarity), and constructs a `faiss.IndexFlatIP` index. `IndexFlatIP` was chosen for its efficiency with dense vectors and its direct support for inner product, which, with normalized vectors, measures cosine similarity. The logs confirm the index was built with 1002 vectors and saved to `indexE5.faiss`.
- **K-Means Clustering for Representative Chunk Selection:**
  - **In-depth Rationale:** To generate a diverse and representative set of QA pairs for evaluation, we needed a method to select chunks covering a wide array of topics within the corpus, rather than randomly sampling or oversampling from dense topic

areas. K-means clustering of the document chunk embeddings provides an effective solution. By grouping semantically similar chunks, we can then sample from each cluster, ensuring that different thematic areas are represented in our evaluation dataset. This helps create a more balanced and comprehensive QA set.

- **Mechanism:**
  1. The embeddings of all 1002 document chunks are generated using the `intfloat/multilingual-e5-large-instruct` model.
  2. K-means clustering is performed on these embeddings. The number of clusters, `k`, was dynamically determined as `min(max(2, int(np.sqrt(num_chunks))), num_chunks)`. For 1002 chunks, this resulted in `k=31` clusters, as shown in the Colab logs.
  3. The script calculates the number of samples to draw from each cluster. The initial target value for `TOTAL_QA_PAIRS` (set to 39 in `1_dataset_pipeline.py`) was determined through experimentation. This specific target, when combined with the methodology of proportional sampling based on cluster size and a rule ensuring at least `max(1, ...)` sample per non-empty cluster, consistently resulted in a total selection of 42 samples. This outcome of 42 questions was a deliberate choice, playfully referencing the project's "DeepThought" theme, inspired by "The Hitchhiker's Guide to the Galaxy" where 42 is the iconic answer.
- **Centroid-based and Diversity Sampling:**
  1. For each cluster, the chunk whose embedding is closest (highest cosine similarity) to the cluster's centroid is selected first.
  2. If more samples are needed from that cluster to meet its quota, subsequent chunks are selected using a diversity sampling approach. This involves iteratively picking the chunk that is least similar (most diverse) to the set of chunks already selected from that cluster, ensuring varied content even within a single cluster.
  3. A t-SNE (t-distributed Stochastic Neighbor Embedding) visualization of the clusters, with highlighted selected chunks and centroids, is generated and saved as `cluster_plots/kmeans_tsne_clusters_k31_highlighted.png` (Figure 1). This plot helps visually confirm the spread and representativeness of the selected chunks.

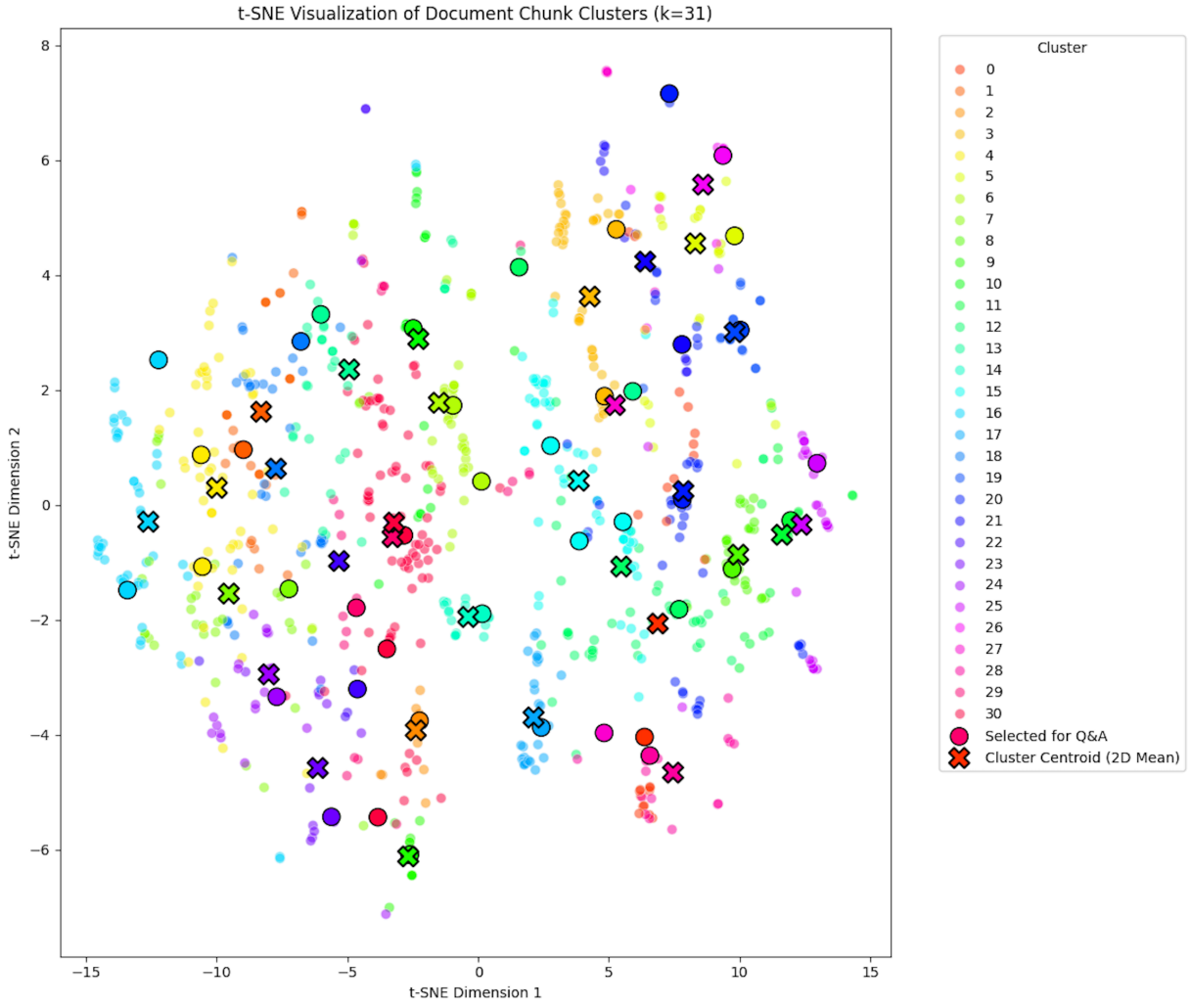

Figure 1: t-SNE visualization of document chunk clusters (k=31). The plot displays document chunks as points, colored by cluster. Circled points indicate chunks selected for Q&A generation (as per the legend "Selected for Q&A"), and 'X' marks denote cluster centroids. This visualization visually confirms the spread and representativeness of the selected chunks, as mentioned in the report.

- **LLM-based QA Pair Generation:**

- The selected 42 document chunks serve as contexts for QA pair generation.
- The `generate_qa_pair` helper function within `1_dataset_pipeline.py` uses the `Together` client to interact with the LLM specified by `QA_GENERATION_MODEL` in `config.py` (`meta-llama/Llama-3.3-70B-Instruct-Turbo-Free`).
- A specific prompt instructs the LLM to generate a relevant question and its corresponding answer *based solely* on the provided chunk text, and to format the output as "Question: [...] Answer: [...]". Retries are implemented in case of generation or parsing failures.
- The generated QA pairs are saved to an intermediate Excel file, `generated_qa_dataset.xlsx`.

- **QA Dataset Augmentation:**
  - The initial QA pairs are then augmented to facilitate retrieval evaluation. For each generated QA pair (specifically, its `Source_Chunk`), the script finds the top-K (where K is `RETRIEVAL_K = 5` from `config.py`) most semantically similar chunks from the entire corpus. This is achieved by calculating the cosine similarity between the `Source_Chunk`'s embedding and all other chunk embeddings.
  - These top-K similar chunks are added as `Relevant_Chunk_Rank_1` to `Relevant_Chunk_Rank_5` columns in the dataset. `Relevant_Chunk_Rank_1` is always the `Source_Chunk` itself.
  - The final, augmented dataset is saved as `generated_qa_dataset_ranked.xlsx`. This file is the primary input for the evaluation scripts.

4.2. `rag_core.py`: The RAG Engine's Heart

This module contains the core logic for our RAG system, providing functions that are used by the dataset pipeline, interactive agent, and evaluation scripts.

- **`initialize_agent(docs_dir, index_file, embedding_model_name, chunk_size)`:**
  - This function is the entry point for setting up the RAG system.
  - It loads the `SentenceTransformer` embedding model specified by `embedding_model_name` (e.g., `intfloat/multilingual-e5-large-instruct` from `config.py`).
  - It initializes the `Together` client using the `TOGETHER_API_KEY` from `config.py` (sourced from environment variables).
  - It calls `utils.load_documents` to load and chunk documents from `docs_dir` with the given `chunk_size`.
  - It attempts to load an existing FAISS index from `index_file`. If the file doesn't exist or loading fails, it calls `build_index` to create a new one.
  - Returns the `index`, `documents_data` (list of chunk dicts), the `model` (embedding model instance), and the `client` (Together client instance).
- **`build_index(documents_data, model, index_file)`:**
  - Takes the list of document chunks and the embedding model.
  - Encodes all chunk texts into embeddings.
  - Normalizes these embeddings to unit length. This is crucial because `faiss.IndexFlatIP` calculates the inner product. For normalized vectors, the inner product is equivalent to cosine similarity, which is our desired semantic similarity measure.
  - Creates a `faiss.IndexFlatIP` index with the dimensionality of the embeddings.
  - Adds the normalized embeddings to the index.
  - Saves the built index to `index_file` (e.g., `indexE5.faiss`).
- **`retrieve_relevant_documents(index, documents_data, model, query, k)`:**
  - Takes the FAISS `index`, the list of `documents_data` (containing original chunk texts), the embedding `model`, the user `query`, and the number of documents `k` to retrieve (from `config.RETRIEVAL_K`).
  - Encodes the input `query` using the `model`.
  - Normalizes the query embedding, consistent with how document embeddings were indexed.
  - Performs a search on the `index` using `index.search(normalized_query_embedding, k)`. This returns the similarity scores and indices of the top-`k` most similar document chunks.
  - Maps these indices back to their original text from `documents_data`.
  - Returns a list of tuples, each containing a retrieved chunk's text and its similarity score, sorted by score in descending order.
- **`generate_answer(question, context_docs, client, model_name, ...)`:**
  - Takes the `question`, a list of `context_docs` (retrieved chunks), the `Together client`, and the `model_name` for generation (e.g., `meta-llama/Llama-3.3-70B-Instruct-Turbo-Free` from `config.QA_GENERATION_MODEL`).
  - Constructs a prompt that includes the system role ("You are an extraction engine... Respond *only* with the extracted answer..."), the combined `context_docs`, and the `question`. The prompt design aims to elicit concise, fact-based answers directly from the provided context.
  - Uses the `client.chat.completions.create` method to send the request to the Together AI API.
  - Extracts and returns the generated answer text. Includes error handling for API issues.

4.3. `utils.py` : Utility Functions

This script provides essential helper functions:

- **`preprocess_text(text)` :** Cleans text by removing extra whitespace.
- **`split_text_into_chunks(text, chunk_size, chunk_overlap)` :** Implements the chunking logic using `llama_index.core.node_parser.SentenceSplitter` , taking `chunk_size` (e.g., 200 from `config.py`) and `chunk_overlap` (default 20) as parameters.
- **`load_documents(directory, chunk_size)` :** Reads all `.md` files from the specified `directory` , preprocesses their content, and then splits them into chunks using `split_text_into_chunks` . It returns a list of dictionaries, where each dictionary contains the `text` of a chunk and its `source_file` name.

4.4. `config.py` : Configuration Hub

This file centralizes all key parameters for the project, making it easy to modify system behavior without altering code in multiple places. Important parameters include:

- `DOCS_DIRECTORY` : Path to the processed Markdown files (`./long_files`).
- `CHUNK_SIZE` : Character size for document chunks (200).
- `EMBEDDING_MODEL` : Name of the SentenceTransformer model (`intfloat/multilingual-e5-large-instruct`).
- `INDEX_FILE` : Filename for the FAISS index (`indexE5.faiss`).
- `QA_GENERATION_MODEL` : LLM used for QA generation and answer synthesis (`meta-llama/Llama-3.3-70B-Instruct-Turbo-Free`).
- `TOGETHER_API_KEY` : API key for Together AI (sourced from environment).
- `RETRIEVAL_K` : Number of documents to retrieve for context (5).

These configurations directly influence data processing, model behavior, retrieval scope, and generation quality.

4.5. `2_interactive_agent.py` : Interactive Querying

This script provides a command-line interface for users to interact directly with the ChrysisRagDeepThought system. It initializes the RAG agent using `rag_core.initialize_agent` . When a user inputs a query:

1. It calls `rag_core.retrieve_relevant_documents` to fetch the top-K relevant context chunks.
2. It then calls a *local* `generate_answer` function (slightly modified from `rag_core.py` 's version to be more conversational and allow general knowledge if context is insufficient) with the query and retrieved contexts to produce an answer using the configured LLM. This script is invaluable for qualitative testing and demonstration purposes.

4.6. Evaluation Scripts (`3_evaluate_retrieval.py` , `4_evaluate_generation.py` , `5_visualize_evaluation.py`)

These scripts form the automated evaluation framework:

- **`3_evaluate_retrieval.py` :**
  - Loads the QA dataset (e.g., `generated_qa_dataset_ranked.xlsx`).
  - Initializes the RAG agent.
  - For each question in the dataset, it retrieves the top-K documents.
  - It compares these retrieved documents against the ground-truth relevant chunks (stored in `Relevant_Chunk_Rank_1` to `Relevant_Chunk_Rank_5` columns).
  - Calculates retrieval metrics: Hit Rate, Mean Reciprocal Rank (MRR), Precision@K, and Kendall's Tau-b, for various values of K (1, 3, 5 as defined in `EVAL_K_VALUES`).
  - Outputs a summary of these metrics to `retrieval_evaluation_summary.json` .
- **`4_evaluate_generation.py` :**
  - Also loads the QA dataset and initializes the RAG agent.
  - For each question:
    - Retrieves relevant documents.
    - Generates an answer using these documents and the LLM.
    - Compares the generated answer to the ground-truth answer from the dataset using semantic similarity. The `intfloat/multilingual-e5-large-instruct` model is used to embed both answers, and their cosine similarity is computed.
  - Outputs detailed results, including questions, generated answers, ground-truth answers, contexts, and semantic similarity scores, to `generation_evaluation_results.csv` . It also reports the Mean Semantic Similarity.
- **`5_visualize_evaluation.py` :**

- Reads the `generation_evaluation_results.csv` and `retrieval_evaluation_summary.json`.
- Generates and saves plots:
  - A histogram/KDE plot of the semantic similarity score distribution (`semantic_similarity_distribution.png`).
  - Bar charts comparing retrieval metrics (Hit Rate, MRR, Precision, Kendall's Tau) across different K values (`retrieval_metrics_comparison.png`).
- These visualizations are saved in the `evaluation_plots/` directory.

This structured approach within `chrysisRagDeepThought_pre-run` allows for reproducible dataset generation, robust RAG operations, and systematic evaluation of the system's capabilities.

5. Methodology Justification

The design and implementation of the ChrysisRagDeepThought system involved several key methodological choices. Each decision was made to optimize performance, relevance, and evaluability for the specific task of information retrieval concerning the University of Cyprus.

- **Choice of RAG Architecture:** We adopted the Retrieval Augmented Generation (RAG) architecture because it effectively combines the strengths of pre-trained LLMs (for fluent and coherent text generation) with information retrieval from a specific, up-to-date knowledge base. For a domain-specific application like UCY information, RAG allows us to ground LLM responses in factual documents, mitigating hallucinations and ensuring answers are relevant to the university's context. This is superior to a general-purpose LLM alone, which might lack specific or current UCY details.
- **Embedding Model (`intfloat/multilingual-e5-large-instruct`):**
  - The `intfloat/multilingual-e5-large-instruct` model was selected for its strong performance in semantic retrieval tasks, particularly its multilingual capabilities. Given that UCY documents can be in both Greek and English, a multilingual model is essential for consistent embedding quality across languages. The "-instruct" variant is fine-tuned to better follow instructions and understand query intent, which is beneficial for embedding queries effectively for similarity search. Its 1024-dimensional embeddings offer a good balance between representational power and computational efficiency for indexing and search.
- **LLM for Generation (`meta-llama/Llama-3.3-70B-Instruct-Turbo-Free` via Together AI):**
  - For both generating QA pairs in `1_dataset_pipeline.py` and synthesizing final answers in `rag_core.py` and `2_interactive_agent.py`, we chose `meta-llama/Llama-3.3-70B-Instruct-Turbo-Free`. This is a powerful instruction-tuned model known for its strong reasoning, instruction-following, and generation capabilities. The "70B" parameter size indicates a large capacity for understanding context and generating nuanced text. Accessing it via the Together AI API provides a convenient and scalable way to leverage this model without needing to host it ourselves. This model is free through Together AI.
- **Vector Database/Indexing (FAISS `IndexFlatIP`):**
  - FAISS (Facebook AI Similarity Search) was chosen for its efficiency and scalability in handling large numbers of vector embeddings. Specifically, `IndexFlatIP` (Inner Product) was used. When combined with L2-normalized embeddings (as implemented in `rag_core.build_index` and `rag_core.retrieve_relevant_documents`), the inner product search becomes equivalent to cosine similarity search, which is a standard and effective measure of semantic similarity. `IndexFlatIP` performs an exhaustive search, ensuring accuracy, which is suitable for our corpus size (1002 chunks). For much larger corpora, approximate nearest neighbor (ANN) indexes within FAISS could be considered.
- **Chunking Strategy (`SentenceSplitter`, Size 200, Overlap 20):**
  - Documents were chunked using `SentenceSplitter` from `llama_index.core.node_parser` with a `CHUNK_SIZE` of 200 characters and an overlap of 20 characters (as per `config.py` and `utils.py`). This relatively small chunk size ensures that retrieved contexts are concise and focused, reducing the amount of irrelevant information passed to the LLM for answer generation. The overlap helps maintain semantic continuity between chunks, reducing the chance of losing context at chunk boundaries. This strategy is a trade-off: smaller chunks can improve precision but might sometimes fragment information that spans across larger segments.
- **Synthetic QA Dataset Generation (with K-Means):**
  - Evaluating RAG systems requires a relevant QA dataset. Manually creating one is time-consuming. We opted for a synthetic approach using `1_dataset_pipeline.py`. The use of k-means clustering on document chunk embeddings to select representative chunks for QA generation was a key innovation. This ensures that our evaluation questions cover diverse topics from the corpus, rather than being skewed towards areas with more text. This leads to a more comprehensive and fair assessment of the RAG system's ability to handle varied queries. The LLM (`meta-llama/Llama-3.3-70B-Instruct-Turbo-Free`) was then used to generate question-answer pairs from these selected diverse chunks.
- **Evaluation Metrics:**
  - **Retrieval:** We used Hit Rate, Mean Reciprocal Rank (MRR), Precision@K, and Kendall's Tau-b.
    - *Hit Rate @K* indicates if any correct document is retrieved within the top K results. A value of 1 means a relevant document was found, 0 otherwise.

- *Mean Reciprocal Rank (MRR) @K*: This metric evaluates how high up the *first* correct document appears in the list of top K results. For each search, if the first correct document is at rank `r` (e.g., 1st, 2nd, 3rd), we calculate its 'reciprocal rank' as 1/`r`. If no correct document is found in the top K, the reciprocal rank for that search is 0. The MRR@K is then the average of these reciprocal ranks across many searches. A higher MRR (closer to 1) indicates that, on average, the system quickly presents a relevant document near the top of the list.
- *Precision@K* measures the proportion of retrieved documents (up to K) that are relevant. It is calculated as the number of relevant documents retrieved in the top K divided by K.
- *Kendall's Tau-b @K* measures the rank correlation between the system's retrieved list (up to K items) and a ground-truth ranked list of relevant documents. It assesses how well the system's ranking order agrees with the ideal ranking order for items present in both lists.

- **Generation:** We primarily used Mean Semantic Similarity (cosine similarity between embeddings of generated and ground-truth answers, using `intfloat/multilingual-e5-large-instruct`). This provides an objective measure of how semantically close the generated answer is to the expected answer.

These choices reflect a pragmatic approach, balancing performance, resource availability (e.g., using an API for the large LLM), and the need for robust, domain-specific evaluation.

6. Evaluation and Results Analysis

The performance of the ChrysisRagDeepThought system was rigorously evaluated using the synthetic QA dataset generated by `1_dataset_pipeline.py`. We conducted two main evaluation runs, orchestrated by the Colab notebooks (`chrysisRagDeepThought_colab_with outputs for automatically generated questions.ipynb` and `chrysisRagDeepThought_colab_with outputs for good questions.ipynb`), to assess the impact of question quality on both retrieval and generation.

Evaluation Setup:

The evaluation process leveraged the scripts `3_evaluate_retrieval.py` and `4_evaluate_generation.py`.

1. **"Automatically Generated Questions" Run:**
   - This run used the `generated_qa_dataset_ranked.xlsx` file as produced directly by `1_dataset_pipeline.py`. This dataset contained 42 question-answer pairs with their corresponding ranked relevant source chunks. The outputs are stored in the `output_for automatically generate questions/` directory.
2. **"Good Questions" Run:**
   - For this run, we used a manually curated version of the QA dataset named `generated_qa_dataset_ranked_qoodQuestions.xlsx`. As noted in the `chrysisRagDeepThought_colab_with outputs for good questions.ipynb`, some initially generated questions were found to be non-specific or poorly formed, potentially leading to suboptimal retrieval. In this "good questions" run, these problematic questions were removed. The `generated_qa_dataset_ranked_qoodQuestions.xlsx` file (containing 33 high-quality questions) was renamed to `generated_qa_dataset_ranked.xlsx` before executing the evaluation scripts. The outputs for this run are in the `output_for good questions/` directory.

All evaluations were performed with `RETRIEVAL_K = 5`, meaning the top 5 documents were retrieved for context.

Retrieval Performance (`3_evaluate_retrieval.py` outputs):

The retrieval metrics are summarized in `retrieval_evaluation_summary.json` within each output directory. We compared performance for K values of 1, 3, and 5. For brevity, we focus on K=5, as it represents the full context window provided to the generator.

- **Automatically Generated Questions (42 Qs):**
  - Mean Hit Rate @5: 0.7381
  - MRR @5: 0.5877
  - Mean Precision @5: 0.3810
  - Mean Kendall Tau-b @5: 0.2460

These retrieval metrics for the automatically generated questions are visualized in Figure 2.

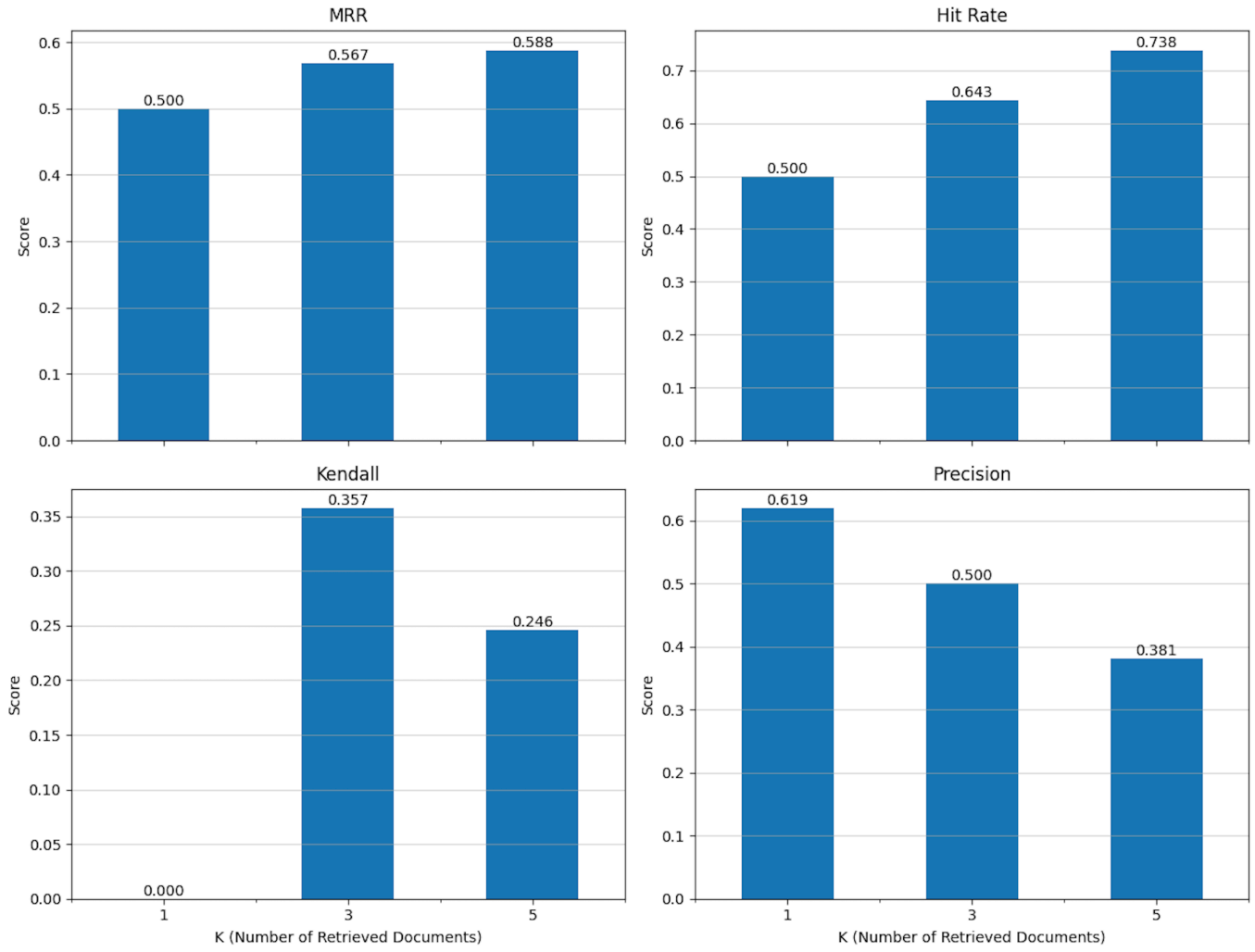

Figure 2: Retrieval performance metrics (MRR, Hit Rate, Kendall's Tau, Precision) for the automatically generated questions dataset across different K values (1, 3, 5).

- **Good Questions (33 Qs):**
  - Mean Hit Rate @5: 0.8485
  - MRR @5: 0.6490
  - Mean Precision @5: 0.3879
  - Mean Kendall Tau-b @5: 0.3091

Figure 3 illustrates these improved retrieval metrics for the 'good questions' dataset.

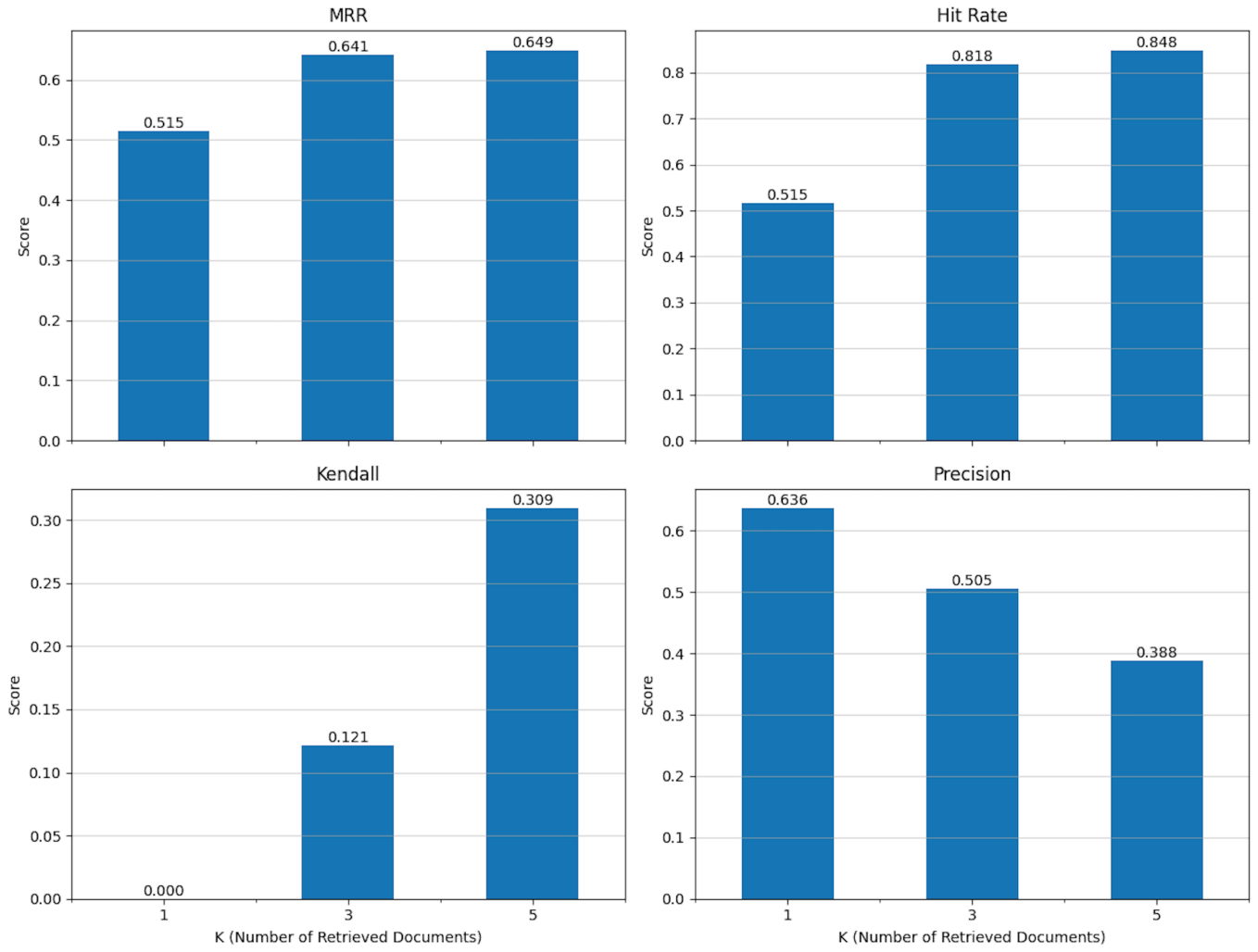

Figure 3: Retrieval performance metrics (MRR, Hit Rate, Kendall's Tau, Precision) for the 'good questions' dataset across different K values (1, 3, 5), showing improvement over the automatically generated set.

Discussion of Retrieval Improvements with "Good Questions": The use of the "good questions" dataset led to notable improvements across most retrieval metrics.

- **Mean Hit Rate @5** increased from 0.7381 to 0.8485. This signifies that for well-posed questions, the system was more likely to include at least one correct source chunk within the top 5 retrieved documents.
- **MRR @5** improved from 0.5877 to 0.6490. This indicates that for "good questions," the correct source chunk was, on average, ranked higher within the top 5 retrieved results.
- **Mean Precision @5** remained relatively stable (0.3810 vs. 0.3879). Precision measures the proportion of retrieved documents that are relevant. This suggests that while the system got better at finding *at least one* correct document and ranking it higher (Hit Rate and MRR), the overall density of relevant documents within the top 5 didn't change drastically. This could be due to the inherent nature of the data or the K value; with K=5 and typically one primary source chunk, precision is sensitive to other retrieved similar-but-not-target chunks.
- **Mean Kendall Tau-b @5** improved from 0.2460 to 0.3091. This suggests a better correlation between the ranking of retrieved documents and the ground-truth ranking of relevant documents for the "good questions."

The primary reason for these improvements lies in the quality of the questions themselves. The Colab notebook highlights that "some of the generated questions are non-specific, leading to poor retrieval." Well-posed, specific questions provide clearer signals to the embedding model, allowing it to generate query embeddings that are more distinct and better aligned with the embeddings of the truly relevant document chunks. Ambiguous or overly broad questions can lead the retriever to fetch semantically related but ultimately incorrect or less optimal contexts. By curating the dataset to include only "good questions," we reduced this ambiguity, enabling the retriever to perform more accurately.

Generation Performance (`4_evaluate_generation.py` outputs):

The generation performance was primarily assessed using Mean Semantic Similarity between the system-generated answers and the ground-truth answers from the QA dataset. These results are in `generation_evaluation_results.csv` in each output directory.

- **Automatically Generated Questions (42 Qs):**

- Mean Semantic Similarity: 0.9429

The distribution of these semantic similarity scores for the automatically generated questions is shown in Figure 4.

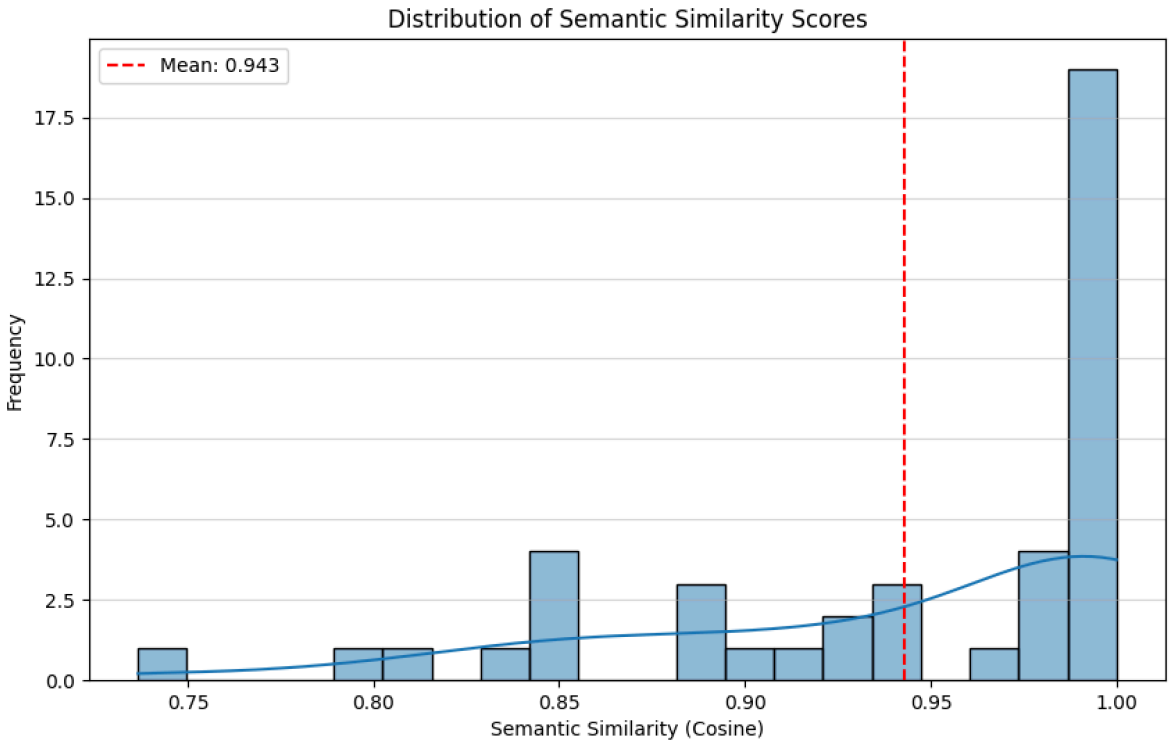

Figure 4: Distribution of semantic similarity scores between generated answers and ground-truth answers for the automatically generated questions dataset.

- **Good Questions (33 Qs):**
  - Mean Semantic Similarity: 0.9631

The distribution of semantic similarity scores for the 'good questions' dataset, depicted in Figure 5, also reflects this improvement.

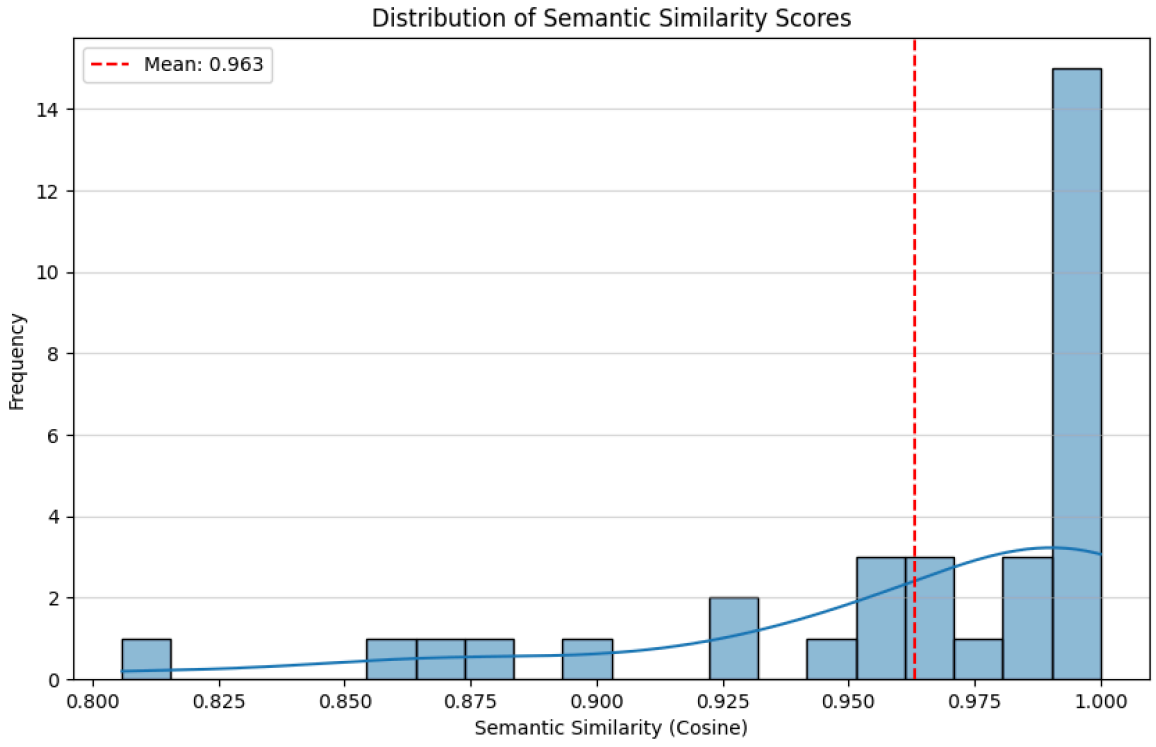

Figure 5: Distribution of semantic similarity scores between generated answers and ground-truth answers for the 'good questions' dataset, indicating higher overall similarity.

Discussion of Generation Improvements with "Good Questions": The Mean Semantic Similarity also improved when using the "good questions" dataset, rising from 0.9429 to 0.9631. This indicates that the answers generated by the LLM were semantically closer to the ground-truth answers when the input questions were of higher quality. This is a direct consequence of improved retrieval. When the retriever provides more accurate and relevant context documents to the LLM (as seen with the "good questions"), the LLM has better source material from which to synthesize an answer. Clearer questions not only aid retrieval but also likely help the LLM focus on the most pertinent information within the provided context to formulate a more accurate response.

Visualizations (`5_visualize_evaluation.py` outputs):

The `5_visualize_evaluation.py` script generates plots stored in the `evaluation_plots/` directory, which aid in understanding these performance trends:

- `semantic_similarity_distribution.png` : This histogram/KDE plot shows the distribution of semantic similarity scores. For the "good questions" run, the distribution is more skewed towards higher similarity values (closer to 1.0), reflecting more consistently accurate answers.
- `retrieval_metrics_comparison.png` : This set of bar charts visualizes the retrieval metrics (Hit Rate, MRR, Precision, Kendall's Tau) across different K values (1, 3, 5). These plots clearly illustrate the improvements achieved with the "good questions" dataset, particularly for Hit Rate and MRR.

In summary, the evaluation clearly demonstrates that the quality of input questions significantly impacts both the retrieval and generation performance of the ChrysisRagDeepThought RAG system. Using a curated set of "good questions" leads to more accurate document retrieval and more semantically correct answer generation.

7. Conclusion and Future Work

The ChrysisRagDeepThought project successfully designed, implemented, and evaluated an advanced RAG system tailored for information retrieval from University of Cyprus documents. We developed a comprehensive pipeline, from data preprocessing and ingestion to sophisticated QA dataset generation using k-means clustering for diverse chunk selection, and finally, a robust evaluation framework. The system leverages powerful embedding models (`intfloat/multilingual-e5-large-instruct`) and LLMs (`meta-llama/Llama-3.3-70B-Instruct-Turbo-Free`) to provide contextually relevant answers.

Our key findings from the evaluation underscore the critical importance of question quality. When evaluated against a curated set of "good questions," the system demonstrated significant improvements in retrieval metrics such as Mean Hit Rate @5 (from 0.7381 to 0.8485) and MRR @5 (from 0.5877 to 0.6490), as well as in generation quality, with Mean Semantic Similarity increasing from 0.9429 to 0.9631. This highlights that well-posed, specific questions enable more accurate document retrieval, which in turn provides better context for the LLM to generate high-quality answers. The k-means clustering approach for selecting chunks for QA generation proved effective in creating a diverse initial set of questions.

Despite these achievements, the system has limitations. The performance is still dependent on the quality of the underlying document corpus and the precision of the chunking strategy. Very nuanced or ambiguously phrased user queries might still pose challenges. Furthermore, the evaluation, while extensive, relied on synthetically generated QA pairs and semantic similarity, which may not perfectly capture all aspects of answer quality like factual accuracy under all conditions or user satisfaction.

Several avenues for future work could enhance the ChrysisRagDeepThought system:

1. **Advanced Retrieval Strategies:** Explore hybrid search (combining dense and sparse retrieval like BM25) or re-ranking models to further improve the relevance of retrieved contexts.
2. **Refined Chunking and Preprocessing:** Experiment with different chunking strategies (e.g., semantic chunking) and more sophisticated NLP techniques for cleaning and structuring the source documents.
3. **Iterative QA Dataset Improvement:** Continuously refine the QA dataset, perhaps incorporating human feedback or more advanced LLM-based question generation and filtering techniques to improve its quality and coverage.
4. **Expanded Evaluation Metrics:** Incorporate human evaluation for aspects like factual accuracy, completeness, and helpfulness. Explore other automated metrics like RAGAs or faithfulness scores.
5. **Model Exploration:** As new and improved embedding models and LLMs become available, evaluate their potential to further boost system performance.
6. **User Interface and Feedback Loop:** Develop a more user-friendly interface and incorporate a feedback mechanism where users can rate answers, providing valuable data for ongoing system improvement.

In conclusion, the ChrysisRagDeepThought project lays a strong foundation for an effective information retrieval system for the University of Cyprus. The insights gained from its development and evaluation provide valuable lessons for future enhancements and for the broader field of domain-specific RAG systems.