Task 2 *(optional)*: Search Improvement

Codebase link: https://github.com/iamcalledayush/task-2

How to run the final working RAG code:

1. *Clone/Download the above repo*
2. *Create a virtual env (optional)*
3. *Install dependencies using pip install -r requirements_new.txt*
4. *Create a .env file by running touch .env command in terminal in root directory. Once the .env is created, go in it and place my OpenAI API key there. So, inside the .env file, put the following line:*
   OPENAI_API_KEY=sk-proj-38RbDlgza0jdeEJaxP2uGTJhxUfP_ughAK9DloNykPsSU4uNJJYuPVUj53QQ52a8bZS7LMuSjST3BlbkFJODIK-K7wp2kKuSG4PDFat7EUcjePxAxLyN8gVm_P_MKEnSou8c2hxOa0gsOEvWoV31E2Iw1wA
5. *To run the RAG pipeline on top of gradio interface, simply run the file: python3 rag_test.py*

Technique Used

***Summary*: I created an end to end RAG pipeline using a new embedding model (e5-large-v2), Step-Back prompting to generate enhanced user queries, GPT4o API, FAISS for vector index database, and Gradio for UI. Please read below for all the steps.**

Step 1: Using the new e5-large-v2 model

I used a different local embedding model: "**intfloat/e5-large-v2**". It is Instruction-tuned, and resulted as top-tier for semantic search as compared to default 8 available models.

The previous best model was: ***stella-base-en-v2.*** ***The metrics for this model were as follows:***

- Top-1 Similarity: 0.8100
- Weighted Top-5 Similarity: 0.7873
- Coverage: 100%
- Avg. Results: 5.00
- Response Time: 0.0213 sec

The metrics for this new model **intfloat/e5-large-v2** **are as follows:**

- Top-1 Similarity: 0.8588
- Weighted Top-5 Similarity: 0.8457
- Query Coverage (%): 100.00
- Average Number of Results: 5.00
- Average Response Time (s): 0.0173

This new model beats the previous best model in every aspect. The embeddings quality are clearly better here for this model.
Metric results are present in the ***evaluation_metrics_e5.csv*** file. For code, you can just run for this single model using the ***task2_e5.py*** file. Also, *I have modified the* ***model_registry.py*** *file to use this model by default along with other 8 models as well.* So running evaluate_models.py will result in metrics comparison/evaluation of all 9 models.

Step 2: Query Enhancement using Step-Back Prompting

Step-back prompting is a technique in prompt engineering where a user's specific query is rephrased into a more general or abstract version. This approach enables large language models (LLMs) to derive high-level concepts and first principles, facilitating more accurate and insightful responses. By focusing on broader aspects of the query, LLMs can better navigate complex reasoning tasks and provide more relevant information.

The ***stepback_method.py*** file contains the creation of Step-Back queries from original queries, using GPT4o API (This part is commented). All the corresponding step-back queries of original queries are stored in the ***stepback_enhanced_queries_sampled.csv*** file with one column for original query and second column for enhanced queries.

The final query embedding was calculated using the following method:

1. Compute the embedding of original query using the new **e5-large-v2** **model.**

2. Compute the embedding of Step-Back version of original query using the new **e5-large-v2 model.**
3. Average both the embeddings to get the final embedding.

Using this new step-back method on the new E5 model, the metrics for the new model improved even further! Following are the new metric results:

- Top-1 Similarity: 0. 8709 (previously 0.8588)
- Weighted Top-5 Similarity: 0. 8613 (previously 0.7873)
- Query Coverage (%): 100.00
- Average Number of Results: 5.00
- Average Response Time (s): 0.07

These are present in the ***evaluation_metrics_e5_avg_embeddings.csv*** file. The code for evaluation is present in the ***stepback_method.py*** file.

Step 3: Embedding Generation using the new E5 model

The new **intfloat/e5-large-v2** embedding model was used to create the embeddings of items in the item_db.csv file. I encoded both the item_name and description from the item DB using E5. Then combined the two embeddings using a weighted average (0.5 for each) to form final item embeddings.

Step 4: FAISS Index Creation

Now, a FAISS indexing database was created. I created a FAISS index using the combined item embeddings. Then, saved the index to disk as ***faiss_index.idx*** and metadata as ***faiss_metadata.pkl***.

Step 5: Similarity Search with FAISS

Retrieved top 3 most similar items from FAISS using the final averaged embedding as described in the end of step 2. Then, I applied a similarity threshold of 0.4 (it can be set to much higher to get more specific results). Took the top 3 (or even less than 3) items which were above the threshold.

Step 6: Generative Response with GPT-4o

Passed the original query and the top retrieved item results to GPT-4o. Asked it (using basic rule based prompting) to generate a user-friendly summary of the matched items.

Step 7: Gradio UI

Created a script for on-the-fly query testing along with Gradio integration for visualization and user UI.

Steps:

- User enters a query.
- Step-back query is generated.
- Averaged embedding ({original + step-back} / 2) is computed.
- Similar items are retrieved and summarized by GPT-4o, and displayed to user on Gradio UI.

Hence, user can input a query in the UI and receive a natural language response from GPT-4o based on RAG retrieval.

The complete RAG pipeline along with Gradio integration is present in the ***rag_test.py*** file. Just run the file after installing all requirements to view the final end product.

For testing purposes, I also ran the RAG pipeline on first 10 queries and the final results were stored in the rag_results_sample.csv file.