NLP Systems Final Project

**Leora Baumgarten, Brynna Kilcline,
Gabby Masini, and Annika Sparrell**

May 7, 2024

1 Introduction

We built a simple retrieval augmented generation (RAG) system to answer questions about books. Given the user's query, we retrieve the most relevant books from the database and pick the best one. Then, we combine the query and chosen book into a prompt, feed it to a large language model, and present both the response and the book's data. Although the subject we chose is fairly mundane, it was of mutual interest to our group and formed a nice milieu to explore the pipelines required around retrieval augmented generation, currently a very hot domain within the industry.

2 Dataset

We used the CMU book summary dataset [Bamman, 2018] from Kaggle.[1] This dataset was compiled as a part of work being done on book summarization [Bamman and Smith, 2013]. It contains information extracted from Wikipedia about 16,559 books. This includes useful information such as a book's title, author, genres, and summary. It also includes other metadata such as a book's Wikipedia ID. Note that this dataset is from 2013, so any books published after that year cannot be retrieved by our system.

3 Database

We use an SQLite database, and we interact with it using SQLAlchemy. Each row in this database represents a book. Each book row contains information such as the book's title, author, publication date, genres, and summary. Note that, at the very minimum, a title is required, and when certain other information in a row is not in the database, `None` is placed there instead. Each

[1] This can be found at `https://www.kaggle.com/datasets/ymaricar/cmu-book-summary-dataset`.

row also contains an embedding vector that represents it. The embedding vectors were generated using the sentence transformer all-MiniLM-L6-v2. The model was fed the book information in a template of the form "[Title] was written{ by [Author]}{ in [Publish Date]}.{ It is a work of [Genres].} Here is a summary of [Title]: [Summary]." Information in the curly braces is optional, and when the information did not exist within the dataset, the corresponding part was removed. Each embedding vector is a `numpy` array of length 384.

4 RAG Pipeline

Our RAG pipeline contains two main components. First, we have an information retrieval step that outputs book information based on the user's query. Second, using this retrieved information in conjunction with the query, we generate a response to it with an LLM. Both steps are described in greater detail below.

4.1 Retrieval

For the information retrieval step, we take the query and create a vector using all-MiniLM-L6-v2. For quick and easy computations, we read in our database to a `pandas` DataFrame at the program's start. We then take the query vector and compute the cosine similarity with each stored book embedding vector. After computing all of the similarity scores, these scores are ranked and we take the top three entries with the highest cosine similarity scores. The information from these entries is then handed off to the next step in our pipeline.

4.2 Generation

For the generation step, we used Mistral-7B [Jiang et al., 2023], a state-of-the-art pretrained generative text model with 7 billion parameters. We used the Mistral AI API for greater efficiency.

After retrieval, a prompt was constructed based on five fields of data about the given book, in the format described in Section 3. The top three books by cosine similarity as retrieved from the database, are fed into this template as context, and the model is asked to respond which book (first,

second, or third) most accurately represents the book described in the user's query. The chosen book is then provided to the model as context, and the model is instructed to concisely answer the user's question based on this context. We chose to use this two-stage approach in order to mitigate errors from the retrieval stage in which the most relevant book was chosen as the second or third ranked option in terms of query similarity.

5 Frontend

The frontend was implemented with Flask due to its compromise of flexible customizability and ease of backend integration. The Flask script queries the SQLAlchemy database with the user's query from the frontend input, then feeds the relevant information into the LLM model to obtain the full results, which are then displayed to the user. The site has a simple, modern interface which naturally prompts the user to submit a query as the main interaction. The results are then displayed below the query field, first giving the extracted answer from the LLM and then key information regarding the relevant book from the database. The user may then requery the model from the same page or return to the default page at any time by clicking the site title. An interactive button that links to the GitHub repository of the project is also provided for convenience. General UX conventions were followed to provide as intuitive an experience to the user as possible, including indicating when the search bar is selected and displaying the active query on the results page.

6 Evaluation

6.1 Test Set

We wrote 45 test questions about books that were answerable from the dataset. We annotated each question with the row from the database that contains the relevant context, and also wrote ground truth answers for each of these questions.

We also wrote a script to automatically generate some test questions based on available information in the database. These include questions about the authors and publication dates of books. The formats of these questions were "When was [Title] written?" and "Who is the author of [Title]?" respectively. These questions, along with the answers and the relevant

database row information, were then stored as dictionaries in files for later use in evaluation. Because not every book entry in the original dataset contained author or publication date information, we ended up with 14177 questions about authors and 10949 questions about publication dates, for a total of 25126 automatically generated test questions.

6.2 Metrics

6.2.1 Retrieval

In order to evaluate the retrieval step of the pipeline, we devised a metric that compared the predicted book context with the ground truth book context. The score for each context is made up of three components:

- a **title score** equal to 1 if the predicted title is identical to the correct title, and 0 otherwise
- an **author score** equal to 1 if the predicted author is identical to the correct author, and 0 otherwise
- a **summary score** equal to the ROUGE-2 score of the predicted summary and correct summary

To get the final score for a single context, each of the three components is scaled—the **title score** by 0.45, the **author score** by 0.25, and the **summary score** by 0.3—for a maximum total of 1. We gave the most weight to the **title score** because users tend to ask questions that include specific book titles, but we also wanted predicted contexts that were close to correct—for example, books that were written by the same author, or that contain the same character names in their summaries—to score higher than those that were completely incorrect, because they would likely be more helpful in the generation step. We then summed the scores for each context and divided by the number of test questions to get an overall average score.

We first used this metric to evaluate retrieval performance on the hand-written test set, and then evaluated on the entire test set, including the automatically generated questions. Because Elasticsearch (see Appendix A) runs much faster than SQLAlchemy, and because they produce identical results, we used Elasticsearch to predict the contexts on the much larger complete test set. The retrieval results are summarized in Table 1. Unsurprisingly, the performance improved on the complete test set. The heuristics we used

	Test Set	Score
Retrieval	Handwritten	0.771
	Complete	0.835
Generation	Handwritten	0.656

Table 1: Evaluation results

to generate test questions meant that each of the automatically generated questions included the full book title, making retrieval easier; in contrast, when handwriting questions we purposefully chose to pose several questions that did not include the full book title, so as to better assess the our system's capabilities, as well as to better mimic user behavior.

6.2.2 Generation

In order to evaluate the generation step of the pipeline, we used the falcon_evaluate library[2] to calculate the semantic similarity between each predicted answer and ground truth answer. We then summed the scores for each answer and divided by the number of test questions to get an overall average score.

We evaluated generation performance on the handwritten set. The results are summarized in Table 1. Because the retrieval scores for the handwritten and automatically generated set weren't drastically different, and because the handwritten questions are more representative of the types of questions real users would ask, we determined that it wasn't worth the time or money it would take to evaluate the generation step on the automatically generated test set.

7 Testing

We wrote unit tests to ensure correct behavior for all of the deterministic functions, including the LLM prompt strings, database entry both for SQLAlchemy and Elasticsearch, and the evaluation scripts.

[2]This can be found at `https://github.com/Praveengovianalytics/falcon-evaluate`.

8 Containerization

The Docker image is built with Python 3.10, slim, as the base. Building the image installs the required packages, and running a container starts the application.

There is a second dockerfile that installs and connects to Elasticsearch. Because it uses Elasticsearch rather than Python as the base image, Python commands cannot be run in the container.

9 Future Work

A future direction for this work would be to integrate Elasticsearch into the application. This would involve composing interdependent Docker containers, with the Flask container depending on the Elasticsearch service for its connection to the database and generation of the password. It would be nice to write a Bash script to parse the credentials outputted by running the Elasticsearch container and automatically store or pass the password to the Python scripts, bypassing the need to manually save the password. Building the index could also be added to the automatic pipeline. Elasticsearch offers very simple functionality for changing the method used to score the relevance of a document to the query, which would be interesting to experiment with. We could implement dot product, L1 and L2 norm scoring functions on our own, but using Elasticsearch would streamline the experimentation.

Additionally, we could experiment with other LLMs and compare results to find the best performing configuration.

Other functionality we considered for a more fleshed-out product would be to allow the user to search the database directly, such as finding all books with a given author, rather than the model intuiting a single best response. We also considered allowing the user to enter new data into the database, either full new entries or to enter missing information for existing entries.

Finally, a longer term goal for this project would be to host it at a live domain. We explored hosting it via GitHub Pages, but found that it is a very involved process if not impossible to host a website with a complex backend on GH Pages.

10 Work Distribution

- Frontend and integration: Annika
- SQLAlchemy and data processing: Gabby
- Mistral and evaluation: Leora
- Elasticsearch and Docker containerization: Brynna
- Unit tests and documentation: Collaborative

References

[Bamman, 2018] Bamman, D. (2018). CMU Book Summary Dataset.

[Bamman and Smith, 2013] Bamman, D. and Smith, N. A. (2013). New alignment methods for discriminative book summarization. *CoRR*, abs/1305.1319.

[Jiang et al., 2023] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. (2023). Mistral 7b.

A Elasticsearch

While Elasticsearch was not integrated into the RAG pipeline due to redundancy with the SQLAlchemy functionality and additional complications such as password access, we did experiment with it. It was set up to mirror the SQLAlchemy functionality so that it could be easily incorporated into the application. The index, which is a database optimized for information retrieval, is created from the relational database with the same fields and embeddings. First, we load the documents from the database. Then, we establish a connection to the Elasticsearch server. We use that to create the index and populate it with the loaded documents.

Once the index is ready, we can process user input. We embed it with SBERT and create an Elasticsearch Query with cosine similarity as the scoring function. There is an option to use a different scoring function. Finally, the top scoring k documents are retrieved.