HW3: Retrieval-Augmented Generation

CSCI 662: Fall 2024

out: Oct 28, 2024
due: Nov 22, 2024

This assignment is designed to lead you through the process of creating a Retrieval-Augmented Generation (RAG) pipeline. RAG, as you may remember from class, is an approach to language modeling where, in order to answer user queries: (1) relevant documents are retrieved from an index, (2) these documents are placed into an LLM's prompt, (3) the language model outputs an answer to the question. The ability of modern language models to use RAG effectively is an exciting new capability for language models. Furthermore, it will test your ability to engineer a larger system that combines a *retriever* with a *generator*. While the key models should not be difficult to implement, expect to spend some time setting up the general framework and considerable time investigating different features and variations of this pipeline that you test.

Code To Write

1. Write code for a retriever that uses at least two different types of retrievers (`BM25` must be one of them) in Python. We have provided stub code for a trainer called `index.py`. It displays its invocation and brief help when invoked as `python index.py -h`. The program takes the following options:

   - `-m [bm25, tfidf, ...]` to specify which kind of model you are using to index the files.
   - `-i <inputfile>` to specify the file with documents to index. Training files will be in the form `<id>TAB<text>`, i.e. a unique identifier for text (that does not contain a tab), a tab, and a line of text.
   - `-n <indexname>` to specify a name for the index.

   It can take other options, too, such as specifying hyperparameters, a dev set so that held-out loss can be displayed, etc.

2. Write code for a generator that uses at least two different types of language models (`gemma2:2b` must be one of them, you'll see why) in Python. The generator code must take in a set of questions, load the retriever, load a large language model and generate answers. We have also provided stub code for a generator called `generator.py`. It should display its invocation and a brief help when invoked as `python generator.py -h`. The program should take the following options:

   - `-m <modelname>` the type of generator model to use ((`gemma2:2b` must be supported)
   - `-n <indexname>` to specify the name of the index for the program to read (i.e. the output of `index.py`)

- `-i <inputfile>` to specify a file of questions to be read. Question files will be in the form `<text>`, i.e. a line of text that does not contain a tab. For each line, an answer should be predicted.
- `-o <outfile>` to specify an output file to be written. The output file should contain one label for each line in the input file.

For more details about retrievers and generators, please refer to the class notes and textbook on RAG, plus any other RAG sources on the web.

Coding Requirements

- The retriever should be able to index a large corpus of documents using at least BM25 and one other method (minimally TF-IDF but you can use other retriever methods as well, and this would be considered extra mile exploration if going beyond basic term matching and/or beyond two methods). It does not need to be able to inject new documents into the index, but that can be an extra-mile effort if demonstrated/justified.
- The retriever should be able to quickly retrieve results from a large index. The retriever should be able to be stored on disk and loaded at will.
- The generator should be able to be run with and without the retriever (for ablation experiments).
- You should have some way of evaluating the output of your retriever on the evaluation set. We will provide an additional file called `answers.txt`, which we will describe later, that contains all possible answers for each query. You will want to check if your generated answer is in the list of possible answers.
- In your writeup, discuss different retrievers and generators you tried and the different scores you got on your own internal test set and on the blind validation set (on Vocareum).
- Submit `index.py`, `generator.py`, and any other code needed. Save results you want to score on Vocareum in a file called `*.answers.txt`. The submission code will look for all files with this suffix and compare against the answers. You need to have generated your output files *first* before submitting to Vocareum. We will *not* be running your model code, we will simply run our evaluation code on your output file.
- Make sure the Vocareum auto-scoring script runs and gives reasonable results.

Getting Started

- Download the starter code.
- Untar the `retrieval_texts.txt.gz` file and write `index.py` code to build your retriever.
- *After* creating your retrieval index, *then* write `generator.py` to test different RAG strategies.

Coding Recommendations

- For the retrieval models, if you are constrained to only use CPU, you might want to use `BM25` and `TF-IDF` as sparse retrievers. If you have access to a GPU (including on your computer), you might want to use `sentence-transformers/all-MiniLM-L6-v2` (from Hugging Face) as a dense retriever. All of these models create high-dimensional feature vectors; the indexing step is the same whether you use a sparse or a dense retriever.
- If stuck implementing this code by yourself, you might want to consider using a package. Here is an example package that we've implemented: https://github.com/alex2awesome/retriv/

- We recommend that you use the FAISS package: (https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/) to index your vectors and retrieve similar vectors quickly.
- If you are GPU-constrained, we recommend that you use the `Ollama` package: https://ollama.com/ to implement and run your generator locally.
- Calculate accuracy locally.
- More likely than not, your language model will produce a lot more text than just a simple answer. That's OK. Unless you are able to engineer your language model prompt to only answer with the question (unlikely, we tried), you might want to design your accuracy tester to simply loop through the possible answers and test if each exists within the LLM's generated text.
- Try different hyperparameters (e.g. first k retrievals where $k = 1, 2, 3...!$) Try with and without the retriever. Try a bigger language model. Try a smaller language model. How responsive are the responses for all of these variations?

Data

We have provided a dataset of retrieval texts, a set of questions for development and a set of questions for testing. For the development questions, you also have a set of development answers to test on.

- `retrieval_texts.txt`: A large set of retrieval texts, separated by newlines, that will be in the format `<id><tab><text>`, as mentioned above, where the `id` field does not contain a `tab`.
- `question.dev.txt`: A set of questions, separated by newlines, that will be in the format `<text>`.
- `question.test.txt`: A set of questions, separated by newlines, that will be in the format `<text>`.
- `answers.dev.txt`: A set of possible answers for each question in the `question.dev.txt` file, separated by newlines. The answers will be in the format `<text><tab><text><tab>....` There will be a variable number of answers per line.

All datasets can be found on Vocareum in `work/datasets` where the evaluation files are named `*.dev.txt` whereas testing/validation answer file is hidden from your view. As mentioned above, you are expected to output files `*.answers.txt` for any variations you wish to run. The auto-grading scripts will automatically run our evaluation script on every `*.answers.txt` file you provide for us. If this score differs greatly from your expectation, you may a) be overfitting, or b) have some design flaw in your code structure (e.g. hardcoded assumptions). You can submit any models you want and name them how you choose, but you should have at least one `*.answers.txt` model file for each data set.

Your Report

Your report should at a *minimum*:

- Explain the key differences in the methods and hyperparameters you used.
- Justify your choice of hyperparameters (include citations where relevant; you do not need to cite instructor notes or lectures) when optimizing for one task and/or when trying to generalize over many tasks.
- Where relevant, show retrieval scores and the effects of these scores on answer correctness (i.e. when you have a more highly-relevant document, does your likelihood of getting the answer correct increase?). Use graphs and tables *appropriately*, not superfluously. This means the graphs/tables should emphasize the message you are delivering, not simply be in place without thinking about why you are using that particular medium to convey an idea.

- Discuss: How do different properties of different questions affect performance? Apart from different categorizations you might come up with (e.g. question length), does anything else matter? How so? How do different retrievers and generators perform differently? Do you find that some generators do not need the retriever to perform well?

Use the ACL style files: https://github.com/acl-org/acl-style-files

There are many ways to write and compile LaTeX; I generally use Overleaf (www.overleaf.com) for minimal headaches, but I have colleagues who abhor Overleaf and greatly prefer to compile on their own machines. Do what works for you.

Your report should be at least two pages long, including references, and not more than four pages long, not including references (i.e. you can have up to four pages of text if you need to). Just like a conference paper or journal article, it should contain an abstract, introduction, experimental results, and conclusion sections (as well as other sections as deemed necessary). Unlike a conference paper/journal article, a complete related works section is not obligatory (but you may include it if it is relevant to what you do).

Grading

As discussed in class, grading will be roughly broken down as follows:

- about 50% – did you clearly communicate your description of what you implemented, how you implemented it, what your experiments were, and what conclusions you drew from them? This includes appropriate use of graphics and tables where warranted that clearly explain your point. This also includes well-written explanations that tell a compelling story. Grammar and syntax are a small part of this (maybe 5% of the grade, so 10% of this section) but much more important is the narrative you tell. Also, a part of this is that you clearly acknowledged your sources and influences with an appropriate bibliography and, where relevant, cited influencing prior work (you do not need to cite instructor notes or lectures).
- about 20% – is your code correct? Did you implement what was asked for, and did you do it correctly?
- about 20% – is your code well-written, documented, and robust? Will it run from a different directory than the one you ran it in? Does it rely on hard-codes? Is it commented and structured such that we can read it and understand what you are doing?
- about 10% – did you go the extra mile? Did you push beyond what was asked for in the assignment, trying new models, features, or approaches? Did you use motivation (and document appropriately) from another researcher trying the same problem or from an unrelated but transferrable paper?

'Extra Mile' ideas

This is not meant to be comprehensive, and you do not have to do any of the things here (nor should you do all of them). But an 'extra mile' component is 10% of your grade. Doing something more innovative (though well-reasoned) than what is listed here will be better than doing exactly what is written here, and doing a correct implementation with a thorough analysis will be better than an incorrect implementation or a trivial analysis.

- Test a stronger retriever. Test an instruction-based retriever (e.g. https://huggingface.co/Salesforce/SFR-Embedding-2_R).
- Test methods for filtering out retrieval results to get a more optimal set of results (e.g. maximal coverage, diversity).
- Add additional data to your retrieval index.

- A careful error analysis will go a long way – if you notice an unexpected drop in performance, analyze what is causing it: is it the retrieval quality? Is it the way the generator uses the retrieved results?
- Dig into the ACL archives and find ideas for RAG, try them out, and analyze performance.
- Implement BM25 with FAISS by yourself, for the challenge :)

Rules

- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code and writing you turn in.
- You may not look for solutions *to this homework* on the web, or use code you find online or anywhere else that is for this homework (you can use code or ideas for the various components, but you should document what you use).
- You may not download the data from any source other than the files provided on Vocareum, and you may not attempt to locate the test data on the web or anywhere else.
- You may use other *external* data that is not the data provided (e.g. a larger retrieval file), and you can make modifications to that data as needed, but you must document the data and modifications in your writeup.
- Generative language, code, and vision models (e.g. ChatGPT, Llama 2, Midjourney, Github Copilot, etc.; if you are unsure, ask and don't assume!!) can be used with the following caveats:
  - You must declare your use of the tools in your submitted artifact. If you don't declare the tool usage but you did use these tools, we will consider that as plagiarism.
  - For code and image generation, you must indicate the prompt used and the output generated.
  - For text generation you must provide either a link to the chat session you used to help write the content or an equivalent readout of the inputs you provided and outputs received from the system. You will lose credit if "the AI" is doing the work rather than you.
- Failure to follow the above rules is considered a violation of academic integrity and is grounds for failure of the assignment or, in serious cases, failure of the course.
- We use plagiarism detection software to identify similarities between student assignments, and between student assignments and known solutions on the web. Any attempt to fool plagiarism detection, for example the modification of code to reduce its similarity to the source, will result in an automatic failing grade for the course.
- If you have questions about what is and isn't allowed, post them to Slack!