

HW3: Retrieval-Augmented Generation

CSCI 662: Fall 2025

Copyright Jonathan May, Alexander Spangher, Katy Felkner. No part of this assignment including any source code, in either original or modified form, may be shared or republished.

out: Oct 27, 2025

due: Nov 21, 2025

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. Implement at least two different kinds of retriever in Python. BM25 must be one of them; the other(s) are your choice. You should implement your retrievers as subclasses of `RetrievalModel.py`. We have provided starter code for a from-scratch implementation of BM25 in `BM25.py`.
 - You are **not** required to implement your retrievers from scratch. Libraries such as `Retriv` <https://github.com/AmenRa/retriv> are allowed.
 - You will probably want to implement some kind of preprocessing before indexing, and you may want to experiment with different preprocessing settings to see how they impact RAG performance. `Retriv` provides a lot of preprocessing options to try!
2. Customize the indexing starter code in `index.py` for your experiments. 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 JSON format, with document consisting of and ID and text.
 - `-n <indexname>` to specify a file to save the index on disk

You can add other arguments (e.g. hyperparameters, a dev set so that held-out loss can be displayed), but please provide sensible defaults for them.

3. Implement retrieval-augmented generation using your retrievers and at least two different pretrained language models. These should be models from different families (e.g. Llama and Gemma), not just two sizes of the same model, but you are welcome to explore multiple model sizes as desired. The generator code must take in a set of questions, load the retriever, load a large language model and generate answers. You can use any LLM libraries you want.
 - We have provided starter code for Ollama and HuggingFace generator models in `ollama_generator.py` and `huggingface_generator.py`. These files have not been tuned or optimized at all – you are allowed and encouraged to make changes to these files to improve their performance!
 - If you want to use another library for generation, you should subclass `GeneratorModel.py` and implement specifics in a new file `somethingelse_generator.py`.
 - You should fill in the prompt in `GeneratorModel.py` and/or do something more sophisticated to build your prompt(s).
 - These files are only suggestions – you aren't required to use any or all of them. You are not required to implement both Ollama and HF generation.
4. Customize the generation starter code in `generator.py` for your experiments. It should display its invocation and a brief help when invoked as `python generator.py -h`. The program should take the following options:
 - `-r <retriever>` the type of retriever model to use [bm25, others].
 - `-n <indexname>` to specify the path to the index for the program to read (i.e. the output of `index.py`)
 - `-k <k>` the number of documents to return in each retrieval run.
 - `-p <platform>` platform to use for the generation. Defaults to "ollama"
 - `-m <modelname>` the type of generator model to use – can be an ollama model name, huggingface identifier, or something else depending on your implementation
 - `-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 generators on the evaluation set. We provide an additional file called `answers.dev.txt` that contains all possible answers for each query in the dev set. You will want to check if your generated answer is in the list of possible answers. We

provide an evaluation script in `evaluator.py`, which we will use in the autograder/leaderboard but you are encouraged to improve on metrics approaches in order to contextualize your results in other ways in your writeup or to gain better understanding of what your system is doing.

- For this assignment, we will **not** run your models on the Gradescope servers. You will run indexing and generation on your system, then submit files called `*.answers.txt` for scoring on Gradescope.
- In your report, discuss different retrievers and generators you tried and the different scores you got on the provided dev set and on the blind test set (on GradeScope).

Files to Submit

You must submit the following files:

1. `bm25.py`
2. another retriever (`.py` file, subclass of `RetrieverModel`)
3. `GeneratorModel.py` with prompt(s)
4. any other files related to your prompt(s)
5. `index.py`
6. `generator.py`
7. modified versions of `RetrieverModel.py`, `huggingface_generator.py`, `ollama_generator.py` if you modified these files
8. at least 4 answers files (minimally, 2 retrievers times 2 generators): `*.answers.txt`.
9. `README.md`
10. `hw3_report.pdf`

Getting Started

- Download the starter code.
- Implement retrieval first, then customize `index.py` and run it to build your indices.
- **After** creating your retrieval index, **then** write your generators and customize `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. In our (from-scratch, unoptimized) staff solution, indexing the full corpus with BM25 took about 2.5-3 minutes on CPU.
- If you have access to a GPU (including on your computer), you might want to use a semantic retriever like `sentence-transformers/all-MiniLM-L6-v2` (from Hugging Face). See the `Retriv` library for examples on dense retrievers.
- If you're implementing retrieval yourself, 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're using `Retriv`, you don't need FAISS.

- If you are GPU-constrained, we recommend that you use the Ollama package: <https://ollama.com/> to implement and run your generator locally.
- Regardless of how you implement generation, the long contexts necessary for RAG mean generation will take some time – plan accordingly. In our (probably suboptimal) staff solution, answering the 500 question dev set takes a little under 4 hours using Ollama and Gemma2-9B on an M1 Mac.
- More likely than not, your language model will produce a lot more text than just a simple answer. That’s OK. For evaluation, we will 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 $k = 1, 2, 3, \dots$!) 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.json**: A large set of retrieval texts in JSON format. Each document has an identifier and the full document text. Full text will contain whitespace including newlines and tabs. Text has been lightly normalized to remove nonstandard whitespace and control characters and clean up inconsistent quotes/apostrophes. If you’re using **Retriv**, you can use the following command to convert JSON to JSONlines as expected by **Retriv**: `jq -c '[]' datasets/retrieval_texts.json > datasets/retrieval_texts.jsonl`.
- **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. **answers.test.txt** is hidden but will be in the same format. Lines may be wrapped in `“”`, which will be stripped off before evaluation.

All datasets can be found in the startercode folder on Brightspace. The auto-grading scripts will automatically run our evaluation code 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 number of answers files and name them how you choose, but you should have at least four `*.answers.txt` files.

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).
- Explain and justify your choices about text preprocessing (for both the retrieval documents and the questions, if applicable).
- 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>

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, methods, 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.

- Implement some of your retrievers (e.g. BM25) from scratch.
- 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.

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 Gradescope, 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, 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!