CSE402: Default Final Project: Building GPT and Retrieval-Augmented Generation (RAG)

Due Date: Jun 7nd, Saturday, 11:59 PM KST.

1 Introduction

The Generative Pretrained Transformer (GPT), the precursor to OpenAI's ChatGPT, has progressively increased in parameter scale across its versions – GPT-1, GPT-2, and GPT-3. In particular, GPT-3 contains about 175 billion parameters and has demonstrated clear scaling laws relating model generalization to parameter size, revealing remarkable emergent abilities such as in-context learning. With its unprecedented scale and capabilities, GPT-3 ushered in a new era of large language models (LLMs).

Despite the success of LLMs, new knowledge and information about the world is continuously generated every minute, but LLMs – having been pretrained or fine-tuned up to a specific point in time – do not capture this newly updated content. To support users who wish to access newly generated information, an obvious approach is to continuously update an LLM's parameters to reflect this new information. However, this parameter-based updating approach is both *costly* – particularly in terms of GPU memory – and *risky*, as it may damage existing knowledge and critical capabilities due to issues such as catastrophic forgetting. To address the burden of updating LLMs, retrieval-augmented generation (RAG) has been widely adopted in commercial LLMs such as ChatGPT and Gemini, as it effectively and iteratively captures up-to-date world knowledge by accessing search engines, leveraging the models' powerful in-context learning and instruction-following capabilities.[10, 2, 4]; The simplest form of RAG consists of two steps: (1) *retrieval*, which identifies a set of relevant passages given a question, and (2) *generation*, which uses the retrieved passages to decode and produce a final answer.

2 Overview

In this default final project, you are required to implement GPT, namely **GPT-small** and RAG by filling up the missing codes of the provided python or Jupyter notebook files. Once the baseline RAG system is established, you will be required to further enhance it for open-domain QA tasks by proposing extension ideas – either from *existing* research papers or through your own *novel* approaches.

Overall, the default final project has two parts: 1) Building a GPT model and its finetuned models on summarization and question answering (QA) tasks, and 2) Building a RAG system (based on the huggingface's language models, as follows:

1. Part 1: **Building a GPT model**, which includes the implementations of the core modules of Transformer and **RoPE**[13]. Please check the main requirements for Part 1, summarized as follows:
   - Please fill in the missing blocks of code to complete the core components of the Transformer model, including the multi-head self-attention (MHSA) module with grouped query attention (GQA) [1], the feedforward network (FFN) layer, RMSNorm layer [16], and RoPE [13]. If necessary, refer to the Transformer paper [14] and the RoPE paper [13] for full implementation details.
   - Please run the pretraining notebook file to train the **GPT-small equipped with RoPE** from scratch using the complete code provided. If you don't have a GPU on your local machine, I recommend subscribing to Google Colab Pro, which should be sufficient for pretraining a small model required in the default final project.

- Please fill in the missing code blocks to finetune the pretrained `GPT-small` on two downstream tasks – summarization and QA tasks.
- Please prepare a detailed report on the performance of downstream tasks, comparing the results with and without pretraining. The report should also include a training progress curve showing metrics (e.g., loss) over the pretraining steps.

2. Part 2: **Building a RAG system**, which involves implementing a RAG architecture where the language model generates answers based on the top-retrieved passages for a given query.

    - Please fill in the missing code blocks to complete the RAG architecture. Note that the provided Python codes in this project are generalized to support both `GPT-small`, your own pretrained model, and Hugging Face language models.
    - Please finetune the RAG model with `GPT-small` on the NQ dataset – an open-domain QA dataset – and record the resulting QA performances.
    - Please deploy the basic prompting methods with few-shot examples for RAG with `Llama-3.2-1B-Instruct` and record the resulting QA performances. We refer to this prompt-based RAG with `Llama-3.2-1B-Instruct` as the **baseline RAG system** in this project.
    - Please improve the baseline RAG system by proposing extension ideas – either from *existing* research papers or through your own *novel* approaches.

Note on default project vs custom project (cited from the Stanford's NLP lecture): The effort/work/difficulty that goes into the default final project is not intended to be less compared to the custom project. It is just that the specific kind of difficulty around coming up with your own problem and evaluation methods was intended to be excluded, allowing students to focus an equivalent amount of effort on this provided problem.

3 Building a GPT model

In this section, you aim to reproduce and improve the architecture of the Transformer Decoder model, GPT, thereby constructing `GPT-small` and finetuning it on two downstream tasks – the summarization, and the classification tasks.

As in the assignments, before you begin, please install `PyTorch 2.6.0` under `Python 3.12` from `https://pytorch.org/` for this assignment. I recommend that you first edit the code in your local device, and then move to work on Google Colab after completing the implementation.

Installing **Java** is also required when working on your local device. If you're using a Conda environment, you can install it via Conda (`https://anaconda.org/conda-forge/openjdk`), and then update the `JAVA_HOME` path to point to the Conda-installed location (e.g., `~/miniconda3/envs/apex/lib/jvm/`).

Please read the related major papers.

- Transformer [14]
- Rotery Positional Embedding (**RoPE**) [13]
- Grouped Query Attention (**GQA**) [1]
- **RMSNorm** [16]

1. Pretraining `GPT-small`: We provide the model python file `model.py`, jupyter notebook file (`main.ipynb`, and python files in `dataset` and `utils` directories. In particular, the file `model.py` implements an improved version of GPT by adopting RoPE, RMSNorm, and GQA. You are first required to complete the missing codes from `model.py` file as follows:

   - **MHSA with GQA**: Complete the missing code in `MultiHeadAttention`
   - **RoPE**: Complete the missing code in the function `apply_rotary_emb` and the class `RotaryEmbedding`.
   - **RMSNorm**: Complete the missing code in the class `RMSNorm`
   - **FFN**: Complete the missing code in the class `FeedForwardNetwork`
   - **Transformer Layer**: Complete the missing code in the class `TransformerLayer`
   - **Attention mask**: Complete the missing code in the class `TransformerModel`
   - **Loss for Causal language model**: Complete the missing code in the class `TransformerForCausalLM`

   To test the pretraining component, please enable the `DO_PRETRAIN` flag variable in `main.ipynb`, and disable all other related flag variables as shown below:

   Listing 1: Code that declares flags indicating whether the corresponding parts or modules are available (in main.ipynb).

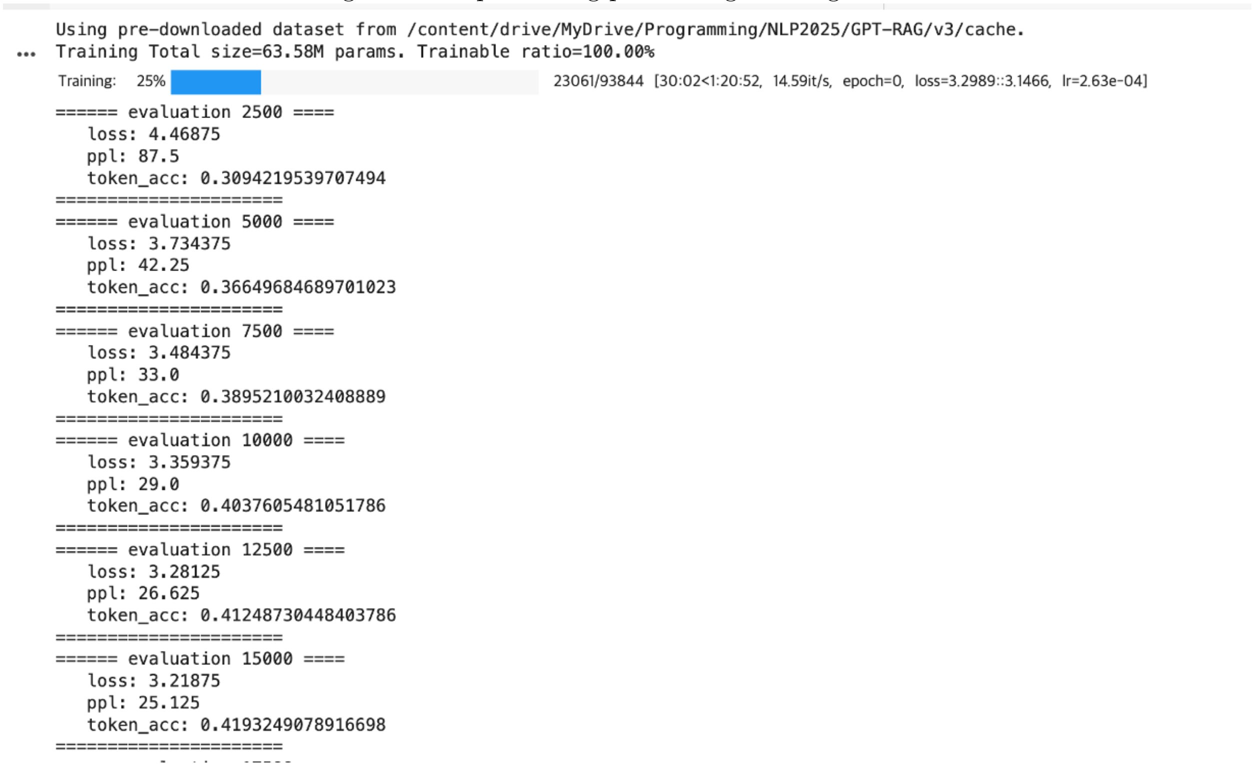
   ```
   DO_PRETRAIN = True
   DO_FINETUNE_SM = False
   DO_FINETUNE_CF = False
   DO_FINETUNE_RAG = False
   DO_ZEROSHOT_RAG = False
   DO_SUBMISSION = False
   ```

   Under the above flag settings, once you correctly edit the required missing codes, when you perform the notebook file `main_ipynb`, you should see the following results.

Listing 2: Outputs during pretraining after running main.ipynb in VS Code

```
    ...
Download and prerpocessing dataset from chengjunyan1/smollm-12.5-corpus on subset cosmopedia
    ↪-v2...

Resolving data files: 100% |==================================| 69/69 [00:00<00:00, 24.27it/s
    ↪]
Resolving data files: 100% |==================================| 69/69 [00:00<00:00, 6948.38it
    ↪/s]
Loading dataset shards: 100% |==================================| 69/69 [00:00<00:00, 1392.69
    ↪it/s]

Saving dataset to ./cache...

Saving the dataset (8/8 shards): 100% |==================================| 750743/750743
    ↪[00:02<00:00,
 286649.85 examples/s]
Saving the dataset (1/1 shards): 100% |==================================| 5846/5846
    ↪[00:00<00:00, 189281.48 examples/s]

Training Total size=63.58M params. Trainable ratio=100.00%

Training:17%.    |======----------------------------|16234/93844 [34:46<2:32:29, 8.48it/s,
    ↪epoch=0, loss=3.3368::3.1990, lr=2.83e-04]

====== evaluation 2500 ====
   loss: 4.46875
   ppl: 87.5
   token_acc: 0.3090330205479246
======================

====== evaluation 5000 ====
   loss: 3.734375
   ppl: 42.25
   token_acc: 0.36617981432191316
======================

====== evaluation 7500 ====
   loss: 3.484375
   ppl: 33.0
   token_acc: 0.38946515467544457
======================

====== evaluation 10000 ====
   loss: 3.359375
   ppl: 29.0
   token_acc: 0.40368312928541594
======================

====== evaluation 12500 ====
   loss: 3.28125
   ppl: 26.75
   token_acc: 0.4120873351171432
======================

====== evaluation 15000 ====
   loss: 3.21875
   ppl: 25.25
   token_acc: 0.4188291264648962
======================
....
```

Figure 1: Output during pretraining at Google Colab



```
Using pre-downloaded dataset from /content/drive/MyDrive/Programming/NLP2025/GPT-RAG/v3/cache.
Training Total size=63.58M params. Trainable ratio=100.00%
Training:  25%                     23061/93844 [30:02<1:20:52, 14.59it/s, epoch=0, loss=3.2989::3.1466, lr=2.63e-04]
====== evaluation 2500 ====
   loss: 4.46875
   ppl: 87.5
   token_acc: 0.3094219539707494
======================
====== evaluation 5000 ====
   loss: 3.734375
   ppl: 42.25
   token_acc: 0.36649684689701023
======================
====== evaluation 7500 ====
   loss: 3.484375
   ppl: 33.0
   token_acc: 0.3895210032408889
======================
====== evaluation 10000 ====
   loss: 3.359375
   ppl: 29.0
   token_acc: 0.4037605481051786
======================
====== evaluation 12500 ====
   loss: 3.28125
   ppl: 26.625
   token_acc: 0.41248730448403786
======================
====== evaluation 15000 ====
   loss: 3.21875
   ppl: 25.125
   token_acc: 0.4193249078916698
======================
```

2. Finetuning `GPT-small` on the summarization and text classifcation tasks: We provide the python files in the `dataset` directory for preparing the dataloader on task-specific datasets. You are first required to complete the missing codes from task-specific python files as follows:

   - **Summarization**: Complete the missing code in `_preprocess` in the `dataset/summary.py`
   - **Classification**: Complete the missing code in `collate_fn_for_classification` in the `dataset/classification.py`.

   To test the finetuning step, please enable the `DO_FINETUNE_SM` (or `DO_FINETUNE_CF`) flag variable in `main.ipynb`, and disable all other related flag variables as shown below:

   Listing 3: Code that enables the finetuning part for summarization task (in main.ipynb).

```
DO_PRETRAIN = False
DO_FINETUNE_SM = True
DO_FINETUNE_CF = False
DO_FINETUNE_RAG = False
DO_ZEROSHOT_RAG = False
DO_SUBMISSION = False
```

   Under the above flag settings, once you correctly edit the required missing codes, when you perform the notebook file `main_ipynb`, you should see the following results.

Listing 4: Outputs during finetuning on summarization task after running main.ipynb in VS Code

```
    ......
Training Total size=63.58M params. Trainable ratio=100.00%

Training:   5% |===                              | 2555/53835 [06:16<1:40:23, 8.51it/s,
    ↪epoch=0, loss=4.4644::4.3500, lr=5.00e-05]
```

```

====== evaluation 2500 ====
   loss: 4.290377505133978
======================

....
```

Similarly, you can enable DO_FINETUNE_CF for the classification task and observe the progress during finetuning.

4 Building a RAG model

RAG [9] provides explicit knowledge as input to the model, eliminating the need for the model to store all knowledge internally. While [9] uses a Transformer encoder-decoder model, we will perform RAG using a decoder-only model (as adopted by most modern LLMs), instead of an encoder-decoder architecture.

In this section, you are expected to implement the required modules for RAG, fine-tune the `GPT-small` model pretrained in Section 3, and explore prompt-based improvements using an instruct LLM. Furthermore, you are required to investigate and implement your own ideas and extensions to enhance the baseline RAG, using either the instruct LLM or `GPT-small`.

4.1 Finetuning RAG with GPT-small on NQ dataset

In this section, you are required to fine-tune `GPT-small`, the small model that you pretrained directly in Section 3, on *Natural Questions* (**NQ**) dataset.

4.1.1 NQ

NQ[8] is a dataset consisting of real questions submitted to the Google search engine, annotated by human annotators to indicate whether the retrieved search results contain long and short answers. Various filtered versions of `nq-open` have been created for open-domain question answering. A common approach is to use only samples with answers of five words or fewer. For RAG, we use the preprocessed dataset from DPR[7].

The preprocessed training and validation data, along with the Wikipedia corpus split into fixed-length passages, can be downloaded from the DPR GitHub repository[1]. As the retriever, we use the sparse retriever BM25 and rely on Pyserini [11][2], a Python toolkit for reproducible information retrieval research, which provides pre-built indexes. For usage details, please refer to the Pyserini GitHub page. Instructions for downloading the dataset and indexes are provided in each Jupyter notebook included in the project.

The number of samples in the `nq-open` dataset we will use is as follows:

- `train`: 58,880 examples
- `dev`: 6,515 examples
- `wiki`: 21,015,324 passages

Each data sample consists of **question**, **answers**, **positive_ctxs**, **negative_ctxs**, and **hard_negative_ctxs**. Note that the negative samples are prepared for training the DPR retriever and are not used by default.

4.1.2 Finetuning RAG with GPT-small

Unlike LLMs which perform reasonably well even in zero-shot settings, small language models like `GPT-small` seriously lack the zero-shot abilities, thereby requiring finetuning on the training set.

For finetuning `GPT-small` for RAG, you will use **question**, **answers**, and **positive_ctxs** from NQ, and evaluate using either **question**, **answers**, **positive_ctxs**, or **BM25 retrieval results**.

[1] https://github.com/facebookresearch/DPR
[2] https://github.com/castorini/pyserini

1. Completing the codes for RAG: We provide `model_rag.py` and the NQ-specific data processing code in `dataset/rag.py`. You are required to complete the missing codes for the baseline RAG in `model_rag.py` as follows:
   - **Retrieval based on BM25**: Complete the missing codes in `search` in `ModelRAG` class in the `model_rag.py` file
   - **Prompt formation based on retrieved passages**: Complete the missing codes in `make_augmented_inputs_for_generate` in `ModelRAG` class
   - **Inference based on RAG**: Complete the missing codes in `retrieval_augmented_generate` in `ModelRAG` class
2. Finetuning RAG with `GPT-small` on NQ dataset: Complete the `__getitem__()` method of `RAGDataset` class in `dataset/rag.py`. Training samples use positive contexts (`positive_ctxs`), and evaluation samples use only `question`, `answers`, and `uid` – the retrieval and tokenization will be handled in `ModelRAG` class.

To test the finetuning `GPT-small` on NQ datasaet, please enable the `DO_FINETUNE_RAG` flag variable in `main.ipynb`, and disable all other related flag variables as shown below:

Listing 5: Code that enables the finetuning part for RAG task (in main.ipynb).

```
DO_PRETRAIN = False
DO_FINETUNE_SM = False
DO_FINETUNE_CF = False
DO_FINETUNE_RAG = True
DO_ZEROSHOT_RAG = False
DO_SUBMISSION = False
```

Under the above flag settings, once you correctly edit the required missing codes, when you perform the notebook file `main_ipynb`, you should see the following results.

Listing 6: Outputs during finetuning on RAG task after running main.ipynb in VS Code

```
    ......
Downloading index at https://rgw.cs.uwaterloo.ca/pyserini/indexes/lucene-index.wikipedia-dpr-100w
    ↪.20210120.d1b9e6.tar.gz...
lucene-index.wikipedia-dpr-100w.20210120.d1b9e6.tar.gz: 8.55GB [10:27, 14.6MB/s]
Apr 24, 2025 6:18:22 AM org.apache.lucene.store.MemorySegmentIndexInputProvider <init>
INFO: Using MemorySegmentIndexInput with Java 21; to disable start with -Dorg.apache.lucene.store
    ↪.MMapDirectory.enableMemorySegments=false
Training Total size=63.58M params. Trainable ratio=100.00%
Training: 24% |===                              |  2663/11040 [07:12<15:32,
 8.98it/s, epoch=0, loss=1.6787::1.9892, lr=4.67e-05]

====== evaluation 2500 ====
   accuracy: 0.032847275518035304
   rouge1: 0.06593055660975829
   rouge2: 0.020944523626795324
   rougeL: 0.06568534220107497
   rougeLsum: 0.06589556687024052
======================
...
```

4.2 Prompting RAG with `Llama-3.2-1B-Instruct` on NQ dataset

With the increasing attention on the in-context following capabilities of LLMs, instruction tuning has emerged as a key approach. By training on diverse instruction-response datasets, these models improve their generalization to various input formats, leading to more effective user interaction beyond specific task templates.

In this section, you will perform `nq-open` **without fine-tuning**, using a large instruction-tuned model, `Llama-3.2-1B-Instruct`.

Please first test the zero-shot RAG using `Llama-3.2-1B-Instruct` on NQ datasaet, by enabling the `DO_ZEROSHOT_RAG` flag variable in `main.ipynb`, and disable all other related flag variables as shown below:

Listing 7: Code that enables the zero-shot RAG task (in main.ipynb).

```
DO_PRETRAIN = False
DO_FINETUNE_SM = False
DO_FINETUNE_CF = False
DO_FINETUNE_RAG = False
DO_ZEROSHOT_RAG = True
DO_SUBMISSION = False
```

When you perform the notebook file `main_ipynb`, you should see the following results.

Listing 8: Outputs of performing zero-shot RAG after running main.ipynb in VS Code

```
    ......
Dataset already exists at 'local_cache/rag/data/nq_open_dpr/nq_dev'
Dataset already exists at 'local_cache/rag/data/nq_open_dpr/nq_train'

config.json: 100% |=================================|  877/877 [00:00<00:00, 191kB/s]
model.safetensors: 100% |=================================|  2.47G/2.47G [00:21<00:00, 111MB/s]
generation_config.json: 100% |=================================| 189/189 [00:00<00:00, 22.6kB/s]
tokenizer_config.json: 100% |=================================| 54.5k/54.5k [00:00<00:00, 7.12MB/
    ↪s]
tokenizer.json: 100% |=================================|  9.09M/9.09M [00:00<00:00, 9.84MB/s]
special_tokens_map.json: 100% |=================================|  296/296 [00:00<00:00, 39.8kB/s
    ↪]

...

====== evaluation ====
   accuracy: 0.2528012279355334
   rouge1: 0.15259079583331261
   rouge2: 0.07130077136984242
   rougeL: 0.15065460462620794
   rougeLsum: 0.15128013880891772
======================
```

4.2.1 Evaluating Various Prompts for RAG

Here, you are required to design various prompting methods by creating a new subclass that **inherits** from `ModelRAG`, or to apply the preprocessing to extract precisely an answer part, as follows. You will create your own codes, and also modify and extend `main.ipynb` if necessary.

- **Input formatting**: Try prompting by referring to the Llama-3.2 User and Assistant conversation, referring to the following:
  `https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/text_prompt_format.md#user-and-assistant-conversation`.
- **Parsing generation results**: Instruction-tuned models like `Llama-3.2-1B-Instruct` tend to generate lengthy outputs with additional explanations. You will post-process the generated text to extract only the relevant parts for evaluation (For example, parse the output as JSON and extract the value

corresponding to the "Answer" key). In particular, please post-process **predictions** in the commented section of `eval_for_rag()` in the `main.ipynb` notebook.

- **Exploring various prompts or CoT prompts** Please design your own model class that inherits `ModelRAG` by developing other advanced prompting methods, such as using few-shot examples, chain-of-thought (CoT) prompting [15].

Please note that **the baseline RAG** refers to your improved version of RAG, tuned using prompting methods, as defined in the default final project. In contrast, the zero-shot RAG using a naive prompt—generated solely with the provided code—is referred to as the **naive RAG**.

4.3 Enhancing RAG with `Llama-3.2-1B-Instruct`

In Section 4.2, you implemented the baseline RAG. In this section, you will be required to propose your own extension and ideas for further improving the baseline RAG. We recommend referring to the list of papers below (not exhaustive).

1. Parallel Context Windows (PCW) [12] for Long RAG: When working with models that have a limited maximum input length, you may consider utilizing Parallel Context Windows (PCW) [12] to accommodate longer inputs. PCW divides a lengthy input into multiple segments, which are then processed independently and in parallel to produce key-value (KV) pairs. These pairs are subsequently stitched together for downstream use. Since each window is processed independently, all windows share identical position IDs. For example, to process an input sequence of 1.5k tokens using a model with a 1k-token input limit, the input can be split into two windows of 750 tokens each. Both windows would be assigned position IDs ranging from 1 to 750. After computing the KV pairs from these windows, the model receives both the precomputed KVs and the question as input. The question is assigned position IDs starting from 751, enabling the model to generate an answer based on the extended context.
2. Context compression [6]
3. Self-RAG [3] for reflective RAGs
4. RAFT [17] for domain specific RAGs

Other approaches include summarizing the long input into a shorter version before use, or selecting only the most relevant segments by measuring their relevance to the question, and reconstructing the input accordingly.

For additional issues related to RAG, you may refer to the survey paper [5] or the tutorial materials [2].

Submission Instructions

You shall submit the results of the final project on Blackboard as two submissions – one for a final report, another for codes and `GPT-small` model files, which are both pretrained and finetuned.

1. Upload your final report `defaultproject_report.pdf`. Please follow ACL* style format given by: `https://github.com/acl-org/acl-style-files?tab=readme-ov-file`. Your report should be 6-8 pages (excluding references and appendices). Basically, the report should include the basic results of `GPT-small` in the benchmarks, and the comparison results of the proposed RAG and baseline RAG systems. If there is limited space to include all content, I strongly recommend focusing primarily on the improvements and extensions made to enhance the baseline RAG system.

   Below, we describe the expected content of the final report. Sections marked as (Required) must be included. You may use a different structure for the other sections if you prefer. For more details, I strongly suggest you to check the instructions from stanford lecture `https://web.stanford.edu/class/cs224n/project/CS224N_Final_Project_Report_Instructions_2025.pdf`

   - **Title. (Required)**
   - **Abstract. (Required)**
   - **Introduction. (Required)**
   - **Related Work. (Required)**
   - **Approach. (Required)**:
   - **Experiments. (Required)** In the default project, if you are mainly focusing on NQ datasets, the detailed desccription of data and evaluation method are not necessary, as they are common. If you use other datasets such as TriviaQA, that are not described in this project guidance, you can refer to the data or present some details of Data and evaluation metric parts. Experimental details can be shortly presented, and too detailed configuration can be provided in Appendices.
   - **Results: (Required)** Report the quantitative results that you have found so far. Use a table or plot to compare results and compare against baselines.
   - **Analysis and Discussion**
   - **Conclusion. (Required)**
   - **Reference. (Required)**
   - **Appendix (optional).**

2. Revise (if necessary) and Run the `main.ipynb` by enabling `DO_SUBMISSION` flag to produce your `defaultproject_codes.zip` file, including `README` file, and Upload the `defaultproject_codes.zip` file. If necessary, please revise the submission section of the code in `main.ipynb` when adding additional code or other files.

   Please note that it is your responsibility to ensure that your code is runnable. If the code does not run, no points will be awarded.

3. Upload your models and datas `defaultproject_supplementaries.zip`.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[2] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. Acl 2023 tutorial: Retrieval-based language models and applications. *ACL 2023*, 2023.

[3] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*, 2024.

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-Augmented Generation for Large Language Models: A Survey, March 2024. arXiv:2312.10997 [cs].

[5] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[6] Haowen Hou, Fei Ma, Binwen Bai, Xinxin Zhu, and Fei Yu. Enhancing and accelerating large language models via instruction-aware contextual compression, 2024.

[7] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics.

[8] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[11] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, pages 2356–2362, 2021.

[12] Nir Ratner, Yoav Levine, Yonatan Belinkov, Ori Ram, Inbal Magar, Omri Abend, Ehud Karpas, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. Parallel context windows for large language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6383–6402, Toronto, Canada, July 2023. Association for Computational Linguistics.

[13] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[16] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[17] Tianjun Zhang, Shishir G Patil, Naman Jain, Sheng Shen, Matei Zaharia, Ion Stoica, and Joseph E. Gonzalez. RAFT: Adapting language model to domain specific RAG. In *First Conference on Language Modeling*, 2024.