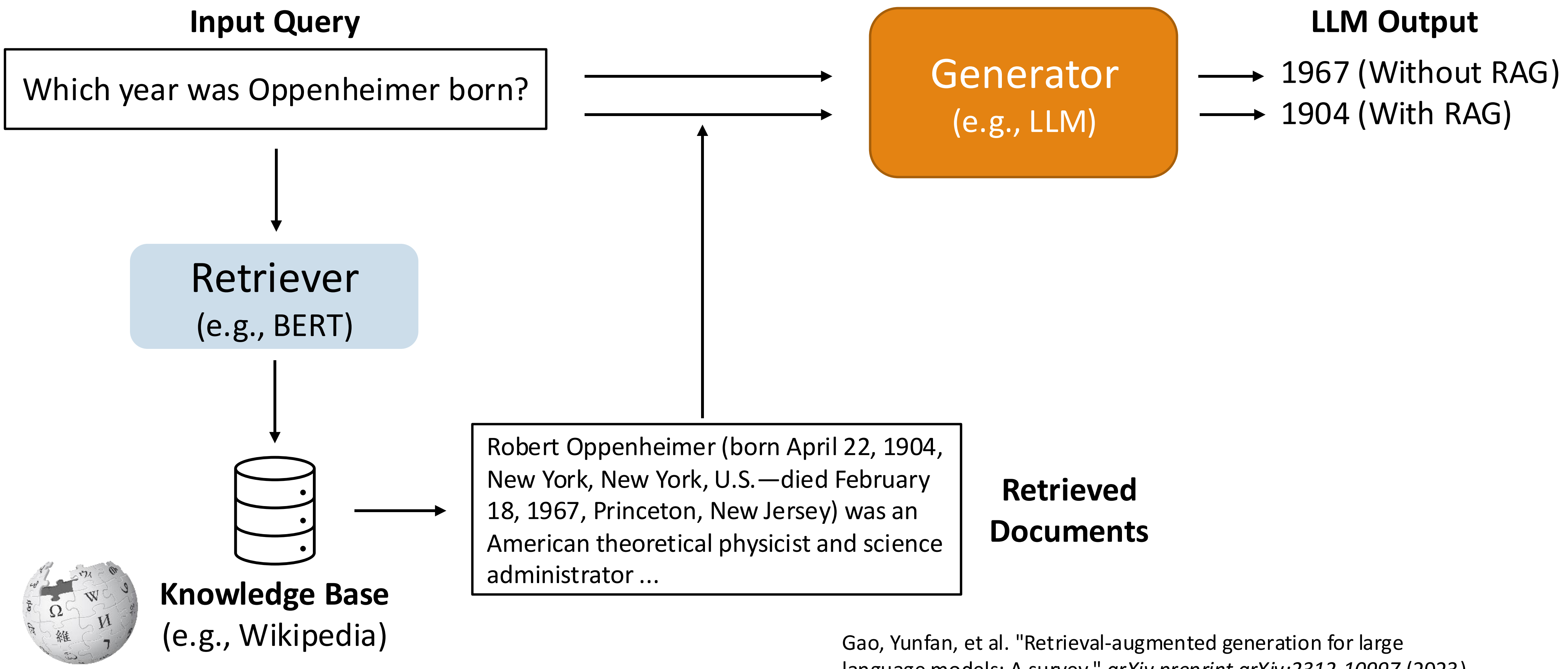
Natural Language Processing

RAG Tutorial 2 (on Colab) 2024/12/05

Outline

- RAG without using LangChain

Retrieval-Augmented Generation (RAG)



Gao, Yunfan, et al. "Retrieval-augmented generation for large language models: A survey." *arXiv preprint arXiv:2312.10997* (2023).

RAG without using LangChain

- In this chapter, we'll implement a RAG system without using LangChain (except the data preparing/preprocess)
- We'll see the detail of Database, Retriever and Model generation.

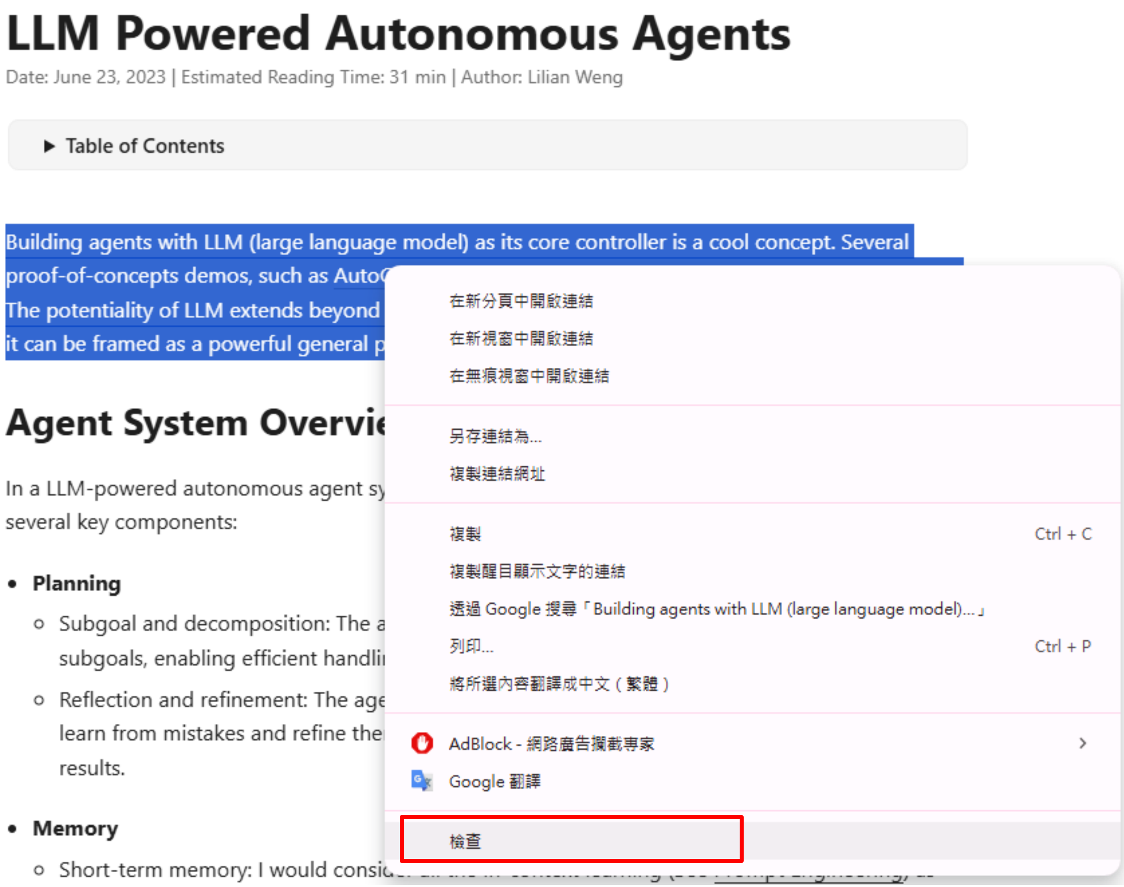
Loading data

- We use WebBaseLoader to crawl the website content by specific class name.
  - You can try any document types you want by checking this website.

```
# Load, chunk and index the contents of the blog.
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
docs = loader.load()
# print(docs[0].page_content)
# print(type(docs[0].page_content))
doc_content = docs[0].page_content
```

Loading data

- We use WebBaseLoader to crawl the website content by specific class name.



Loading data

- After crawling data from the website, we need to perform an initial preprocessing step to clean up the dirty data and transform it into a more organized and refined format.

```
def data_preprocessing(text):
    # Replace newline characters
    text = text.replace("\n", " ")
    # Remove excessive punctuation (e.g., "!!!" -> "!")
    text = re.sub(r'([.,!?])\1+', r'\1', text)

    # Replace multiple spaces with a single space
    text = re.sub(r'\s+', ' ', text)

    # Remove URL,HTML tags
    text = re.sub(r'https?://\S+|www\.\S+', '', text)
    text = re.sub(r'<.*?>', '', text)

    # Remove special characters (keep alphanumeric and basic punctuation)
    text = re.sub(r'[^a-zA-Z0-9\s.,!?\'"-]', '', text)

    text = text.strip()
    return text
```

```
doc_content = data_preprocessing(doc_content)
```

Loading data

- We need to preprocess the content into chunks so that we can fit the [top-k chunks] and [query] within the maximum length limit.
- Avoid retrieving overly lengthy paragraphs to reduce the irrelevant content in the generated results.
- The text splitter is the recommended one for generic text.
  - chunk_size, overlap, …

```
[ ]  text_splitter = TokenTextSplitter.from_huggingface_tokenizer(
             tokenizer,
             chunk_size=100,
             chunk_overlap=20,
     )
     textobj_db = text_splitter.create_documents([doc_content])
     print(len(textobj_db))
     print(textobj_db[:3])
```

Build up the database

- In real-world scenarios, we often deal with databases containing thousands or even millions of chunks/documents. Therefore, efficient saving and loading processes are essential.

```
text_db_path = "./text_db.json"
vector_db_path = "./vector_db.json"
bm25_db_path = "./bm25_tokenized_corpus.pkl"
```

Build up the database

```python
# Build up text database & vector database
text_db = []
for id, text_obj in enumerate(textobj_db):
    text_dict = {"id": id,"text": text_obj.page_content}
    text_db.append(text_dict)


emb_model = AutoModel.from_pretrained('jinaai/jina-embeddings-v2-base-en', trust_remote_code=True) # trust_remote_code is neede

vector_db = []
for text in tqdm(text_db):
    vector_dict = {"id": text["id"], "text": text["text"], "vector_obj": emb_model.encode(text["text"]).tolist()}
    vector_db.append(vector_dict)

# Save text_db & vector_db to reuse
with open(text_db_path, "w") as f:
    json.dump(text_db, f)
with open(vector_db_path, "w") as f:
    json.dump(vector_db, f)
```

Give each chunks id

Get chunks' embedding

Save text_db & vector_db

Build up the database

```python
def build_bm25_index(text_db):
    """
    Builds a BM25 index from a list of text documents.
    """
    # Extract texts and preprocess them
    texts = [doc["text"] for doc in text_db]
    tokenized_corpus = preprocess_texts(texts)

    # Build BM25 index
    bm25 = BM25Okapi(tokenized_corpus)
    return bm25, tokenized_corpus
```

Preprocess the texts into lowercase

Use the Rank-BM25 package to index the corpus effectively

```python
# Build the BM25 index
bm25, tokenized_corpus = build_bm25_index(text_db=text_db)

# Save the tokenized corpus
save_bm25_index(bm25=bm25, file_path=bm25_db_path)
```

Build and then save it

Dense retriever

- Using dense comparison, we leverage cosine_similarity to rank the chunks based on their relevance to the query.

```python
def dense_ranker(query, vector_db, model):
    """
    Ranks documents using cos_sim for a given query.
    """
    # Encode the query into a vector
    query_vector = model.encode(query)

    # Compute cosine similarity scores for each vector in the database
    scores = [
        {
            "id" : doc["id"],
            "text": doc.get("text", None),    # Optional: retrieve document content if available
            "score": cos_sim(query_vector, doc["vector"])
        }
        for doc in vector_db
    ]

    # Sort the documents by score in descending order
    ranked_docs = sorted(scores, key=lambda x: x["score"], reverse=True)

    return ranked_docs
```

Compute cos_sim score

Get the rank with chunks id

Sparse retriever

- Using sparse comparison, the rank_bm25 package allows us to compute BM25 scores for each chunk based on the query, enabling us to rank them effectively.

```python
def bm25_ranker(query, bm25, text_db, tokenized_corpus):
    """
    Ranks documents using BM25 for a given query.
    """
    # Tokenize query
    tokenized_query = word_tokenize(query.lower())

    # Get BM25 scores
    scores = bm25.get_scores(tokenized_query)

    # Rank documents by score
    ranked_docs = sorted(
        [{"id": doc["id"], "text": doc["text"], "score": scores[i]} for i, doc in enumerate(text_db)],
        key=lambda x: x["score"],
        reverse=True
    )

    # Return results
    return ranked_docs
```

Compute bm25 score

Map score with ID and then rank them

Hybrid retriever

- **Hybrid Retriever** combines **vector search** (semantic search) with **sparse search** (e.g., BM25) to leverage the strengths of both approaches.
- There are few methods to combine multiple ranking, here we use RRF (Reciprocal Rank Fusion) to combine two ranking results.

$$\text{RRF score}(d) = \frac{1}{k + r}$$

A constant that control the influence of rank positions

Rank position

Hybrid retriever

```python
def hybrid_ranker_rrf(dense_ranked_docs, sparse_ranked_docs, k=60):
    """
    Combines dense and BM25 ranking results using Reciprocal Rank Fusion (RRF).
    """
    # Create dictionaries for quick look-up of ranks
    dense_ranks = {doc["id"]: rank for rank, doc in enumerate(dense_ranked_docs, start = 1)}
    sparse_ranks = {doc["id"]: rank for rank, doc in enumerate(sparse_ranked_docs, start = 1)}

    # Collect all unique document IDs from both ranking results
    all_doc_ids = set(dense_ranks.keys()).union(sparse_ranks.keys())

    # Compute RRF scores
    rrf_scores = {}
    for doc_id in all_doc_ids:
        dense_rank = dense_ranks.get(doc_id, len(dense_ranks)+1)   # Use max_len of sorted + 1 for missing docs
        parse_rank = sparse_ranks.get(doc_id, len(sparse_ranks)+1)

        # RRF score formula: 1 / (k + rank)
        rrf_scores[doc_id] = (1 / (k + dense_rank)) + (1 / (k + parse_rank))

    # Combine results and sort by RRF score
    hybrid_ranked_docs = []
    for doc_id, rrf_score in sorted(rrf_scores.items(), key=lambda x: x[1], reverse=True):
        # Retrieve document content from either dense_ranked_docs or parse_ranked_docs
        doc_content = next(
            (doc["text"] for doc in dense_ranked_docs if doc["id"] == doc_id),
            next(doc["text"] for doc in sparse_ranked_docs if doc["id"] == doc_id)
        )
        hybrid_ranked_docs.append({"id": doc_id, "text": doc_content, "score": rrf_score})

    return hybrid_ranked_docs
```

Assigning ranks based on sorted results

Handle the missing docs errors

RRF algorithm

Get the final ranking by hybrid score

Retriever

- Consolidate the dense ranker, sparse ranker, and hybrid ranker into a single function that handles the complete ranking workflow.

```
def personal_retriever(query, text_db_path, vector_db_path, bm25_db_path, emb_model, topk=3):
    """
    Using hybrid method to retrieve the top_k docs from database.
    """
    # Load text_db
    text_db = load_text_db(text_db_path)
    # Load vector_db
    vector_db = load_vector_db(vector_db_path)
    # Load the tokenized corpus and rebuild the BM25 index
    bm25, tokenized_corpus = load_bm25_index(bm25_db_path)

    topk = 3
    dense_ranked_docs = dense_ranker(query=query, vector_db=vector_db, model=emb_model)
    sparse_ranked_docs = bm25_ranker(query=query, bm25=bm25, text_db=text_db, tokenized_corpus=tokenized_corpus)

    # Rank by using hybrid_ranker_rrk
    hybrid_ranked_docs = hybrid_ranker_rrf(dense_ranked_docs=dense_ranked_docs, sparse_ranked_docs=sparse_ranked_docs)
    return hybrid_ranked_docs[:topk]
```

Retriever

- Consolidate the dense ranker, sparse ranker, and hybrid ranker into a single function that handles the complete ranking workflow.

```
[ ] query = "What is the Self-Reflection?"
```

```
# Call personal_retriever
retrieved_docs = personal_retriever(query=query, text_db_path=text_db_path, vector_db_path=vector_db_path /
                                    , bm25_db_path=bm25_db_path, emb_model=emb_model, topk=3)
print(retrieved_docs)
```

Reader model loading

- You must have access to use the model. If not, you can use any other LLM model that does not require access.

```
# Initialize the Llama-3.2-1B model
model_id = "meta-llama/Llama-3.2-1B-Instruct"

# Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_id)
print("Tokenizer loaded successfully.")

# Load the model
model = AutoModelForCausalLM.from_pretrained(
        model_id,
        return_dict=True,
        torch_dtype=torch.float16,
        device_map="auto",
        trust_remote_code=True,
)
print("Model initialized successfully.")
```

Prompt Setting

- Set up the prompt and populate it with the query and retrieved chunks.
- Once completed, tokenize the prompt to prepare it for model input.

```python
def format_docs(docs):
        return "\n\n".join(doc["text"] for doc in docs)
```

```python
# Prompt setting refer to langchain "rlm/rag-prompt"
system_prompt = """
You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't
"""

user_prompt = """
Question: {question}
Context: {context}
Answer:
"""

input_prompt = (system_prompt + user_prompt).format(question=query, context=format_docs(retrieved_docs))
# print(input_prompt)
```

```python
# Tokenize the prompt to prepare it for model input
inputs = tokenizer(input_prompt, return_tensors="pt")
```

Reader Generation

- Move the input to the same device as the model, then use the general generation function provided by the Hugging Face library.
- Convert the generated tokens back into human-readable text by decoding them.

The inference time is directly proportional to the max_new_tokens value.

```
# Generate the output sequence from the model
outputs = model.generate(**inputs.to(device), pad_token_id=tokenizer.eos_token_id, max_new_tokens=300)

# Decode the generated tokens to convert them back to readable text
output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print("Output text:", output_text)
```

Retriever Evaluation

Required Packages

```
import json
from tqdm import tqdm
from transformers import AutoTokenizer
from transformers import AutoModel
from helper_functions import load_text_db,
build_bm25_index, save_bm25_index, personal_retriever
```

The cat fact database (in Assignment 4)

[1]
```
!wget https://huggingface.co/ngxson/demo_simple_rag_py/resolve/main/cat-facts.txt
```

0 秒 [2]
```
with open("cat-facts.txt", "r") as f:
  refs = f.read().splitlines()
```

0 秒 [3]
```
for ref in refs[:5]:
    print(ref)
```

```
On average, cats spend 2/3 of every day sleeping. That means a nine-year-old cat has been awake for only three years of its life.
Unlike dogs, cats do not have a sweet tooth. Scientists believe this is due to a mutation in a key taste receptor.
When a cat chases its prey, it keeps its head level. Dogs and humans bob their heads up and down.
The technical term for a cat’s hairball is a “bezoar.”
A group of cats is called a “clowder.”
```

Build up text database & vector database (1/2)

```
[5]
text_db_path = "./cats_text_db.json"
vector_db_path = "./cats_vector_db.json"
bm25_db_path = "./cats_bm25_tokenized_corpus.pkl"
```

```
[6]
emb_model_id = 'jinaai/jina-embeddings-v2-base-en'
emb_tokenizer = AutoTokenizer.from_pretrained(emb_model_id)
emb_model = AutoModel.from_pretrained(
    emb_model_id,
    trust_remote_code=True, # trust_remote_code is needed to use the encode method
)
```

Build up text database & vector database (2/2)

```
# Build up text database & vector database
text_db = []
for id, text in enumerate(refs):
  text_dict = {"id": id,"text": text}
  text_db.append(text_dict)

vector_db = []
for text in tqdm(text_db):
  vector_dict = {
      "id": text["id"],
      "text": text["text"],
      "vector": emb_model.encode(text["text"]).tolist()
  }
  vector_db.append(vector_dict)

# Save text_db & vector_db to reuse
with open(text_db_path, "w") as f:
  json.dump(text_db, f)
with open(vector_db_path, "w") as f:
  json.dump(vector_db, f)
```

```
100%|██████████| 150/150 [00:32<00:00,  4.60it/s]
```

Questions and Answer Sentences

```python
queries = [
    "How much of a day do cats spend sleeping on average?",
    "What is the technical term for a cat's hairball?",
    "What do scientists believe caused cats to lose their sweet tooth?",
    "What is the top speed a cat can travel over short distances?",
    "What is the name of the organ in a cat's mouth that helps it smell?",
    "Which wildcat is considered the ancestor of all domestic cats?",
    "What is the group term for cats?",
    "How many different sounds can cats make?",
    "What is the name of the first cat in space?",
    "How many toes does a cat have on its back paws?"
]
golden_chunks = [
    "On average, cats spend 2/3 of every day sleeping. That means a nine-year-old cat has been awake for only three years of its l
    "The technical term for a cat’s hairball is a “bezoar.””",
    "Unlike dogs, cats do not have a sweet tooth. Scientists believe this is due to a mutation in a key taste receptor.",
    "A cat can travel at a top speed of approximately 31 mph (49 km) over a short distance.",
    "Besides smelling with their nose, cats can smell with an additional organ called the Jacobson’s organ, located in the upper s
    "The ancestor of all domestic cats is the African Wild Cat which still exists today.",
    "A group of cats is called a “clowder.””",
    "Cats make about 100 different sounds. Dogs make only about 10.",
    "The first cat in space was a French cat named Felicette (a.k.a. “Astrocat”) In 1963, France blasted the cat into outer space.
    "Cats have five toes on each front paw, but only four toes on each back paw.",
]
```

Build BM25 index and save tokenized corpus

```
[ ]
# Build the BM25 index
text_db = load_text_db(text_db_path)
bm25, tokenized_corpus = build_bm25_index(text_db=text_db)

# Save the tokenized corpus
save_bm25_index(bm25=bm25, file_path=bm25_db_path)
```

Calculate Recall and Precision

```
[ ]
recall = 0
precision = 0

for i, query in enumerate(queries):
    retrieved_docs = personal_retriever(
        query=query,
        text_db_path=text_db_path,
        vector_db_path=vector_db_path,
        bm25_db_path=bm25_db_path,
        emb_model=emb_model,
        topk=3,
        k_bm=60,
    )
    for j, retrieved_doc in enumerate(retrieved_docs):
        if golden_chunks[i].lower() == retrieved_doc["text"].lower():
            recall += 1
            precision += 1/(j+1)
            break

recall_final = recall / len(queries)
precision_final = precision / len(queries)
```

Calculate Recall and Precision

Recall

$$\frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Top1	Top2	Top3
1/1	1/1	1/1

Precision

$$\frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Top1	Top2	Top3
1/1	1/2	1/3

https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)

Get the retrieval scores

```python
recall_final = recall / len(queries)
precision_final = precision / len(queries)
print(f"Recall@3: {recall_final}")
print(f"Precision@3: {precision_final}")
print(f"F1 score: {2*recall_final*precision_final/(recall_final+precision_final)}")
```

```
Recall@3: 1.0
Precision@3: 1.0
F1 score: 1.0
```