

✓ CSE 256 FA24: NLP UCSD PA3:

Vedant Deshpande (A69032161)

✓ Retrieval-Augmented Generation (RAG) (40 points)

The goal of this assignment is to gain hands-on experience with aspects of **Retrieval-Augmented Generation (RAG)**, with a focus on retrieval. You will use **LangChain**, a framework that simplifies integrating external knowledge into generation tasks by:

- Implementing various vector databases for efficient neural retrieval. You will use a vector database for storing our memories.
- Allowing seamless integration of pretrained text encoders, which you will access via HuggingFace models. You will use a text encoder to get text embeddings for storing in the vector database.

Data

You will build a retrieval system using the [QMSum Dataset](#), a human-annotated benchmark designed for question answering on long meeting transcripts. The dataset includes over 230 meetings across multiple domains.

Release Date: November 6, 2024 | Due Date: November 18, 2024

IMPORTANT: After copying this notebook to your Google Drive along with the two data files, paste a link to your copy below. To create a publicly accessible link:

1. Click the *Share* button in the top-right corner.
2. Select "Get shareable link" and copy the link.

Link: paste your link here: <https://colab.research.google.com/drive/16apZFVotXFuHPxgAfaDCsTTbGfQGpQHG?usp=sharing>

Notes:

Make sure to save the notebook as you go along.

Submission instructions are located at the bottom of the notebook.

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

FOLDERNAME = 'UCSD/Fall_24/CSE256/PA3/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

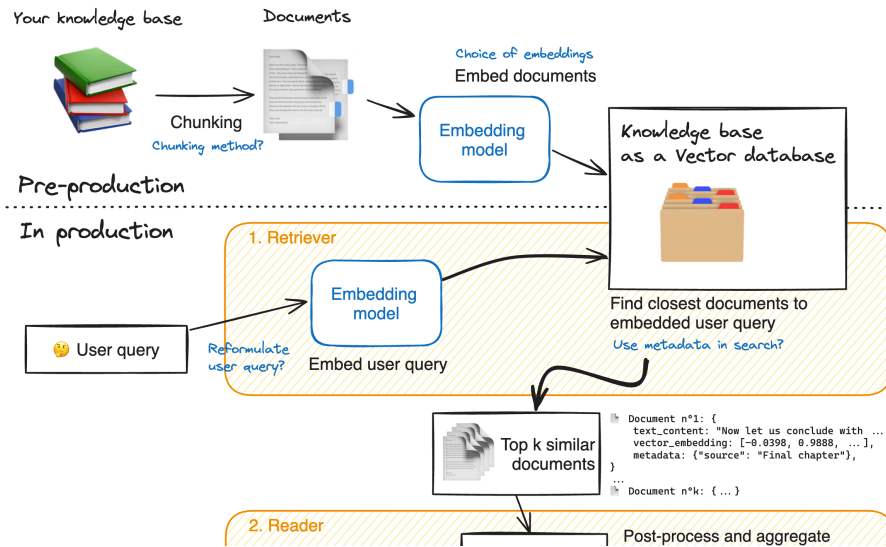
# This is later used to use the IMDB reviews
%cd /content/drive/My Drive/$FOLDERNAME/

📁 Mounted at /content/drive
/content/drive/My Drive/UCSD/Fall_24/CSE256/PA3
```

RAG Workflow

Retrieval-Augmented Generation (RAG) systems involve several interconnected components. Below is a RAG workflow diagram from Hugging Face. Areas highlighted in blue indicate opportunities for system improvement.

In this assignment, we will focus on the ***Retriever** so the PA does not cover any processes starting from "2. Reader" and below.



✓ First, install the required model dependencies.

```
pip install -q torch transformers langchain_chroma bitsandbytes langchain faiss-gpu langchain_huggingface langchain-community
```

```

Preparing metadata (setup.py) ... done
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
122.4/122.4 MB 7.3 MB/s eta 0:00:00
85.5/85.5 MB 8.5 MB/s eta 0:00:00
2.4/2.4 MB 65.5 MB/s eta 0:00:00
615.5/615.5 kB 40.3 MB/s eta 0:00:00
2.4/2.4 MB 67.4 MB/s eta 0:00:00
94.9/94.9 kB 7.9 MB/s eta 0:00:00
3.1/3.1 MB 74.4 MB/s eta 0:00:00
273.8/273.8 kB 23.3 MB/s eta 0:00:00
1.9/1.9 MB 68.6 MB/s eta 0:00:00
49.5/49.5 kB 4.3 MB/s eta 0:00:00
93.2/93.2 kB 7.7 MB/s eta 0:00:00
13.3/13.3 MB 75.8 MB/s eta 0:00:00
55.8/55.8 kB 5.1 MB/s eta 0:00:00
54.4/54.4 kB 4.7 MB/s eta 0:00:00
73.3/73.3 kB 6.5 MB/s eta 0:00:00
63.7/63.7 kB 5.1 MB/s eta 0:00:00
442.1/442.1 kB 32.7 MB/s eta 0:00:00
316.6/316.6 kB 23.0 MB/s eta 0:00:00
3.8/3.8 MB 52.6 MB/s eta 0:00:00
425.7/425.7 kB 25.8 MB/s eta 0:00:00
168.2/168.2 kB 12.1 MB/s eta 0:00:00
46.0/46.0 kB 4.1 MB/s eta 0:00:00
86.8/86.8 kB 6.7 MB/s eta 0:00:00
Building wheel for annoy (setup.py) ... done
Building wheel for pypika (pyproject.toml) ... done
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour
tensorflow 2.17.1 requires protobuf!=4.21.0,!4.21.1,!4.21.2,!4.21.3,!4.21.4,!4.21.5,<5.0.0dev,>=3.20.3, but you have
tensorflow-metadata 1.13.1 requires protobuf<5,>=3.20.3, but you have protobuf 5.28.3 which is incompatible.

```

```

from tqdm.notebook import tqdm
import pandas as pd
import os
import csv
import sys
import numpy as np
import time
import random
from typing import Optional, List, Tuple
import matplotlib.pyplot as plt
import textwrap
import torch
from numpy.linalg import norm

```

```
# ensures that the randomness is consistent across runs, which makes experiments and results reproducible
```

```

seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

# deterministic behavior in PyTorch's CUDA backend, which helps in achieving reproducibility by removing non-deterministic c
torch.backends.cudnn.deterministic = True

# disable CUDA's benchmarking mode, which could introduce variability in execution time and memory usage. This setting is al
torch.backends.cudnn.benchmark = False

# disable huggingface tokenizers parallelism
# reduce the number of parallel threads and avoid potential race conditions or excess resource usage when tokenizing text
os.environ["TOKENIZERS_PARALLELISM"] = "false"

```

Start coding or [generate](#) with AI.

✓ Load the meetings dataset

```

from langchain.docstore.document import Document

def load_documents(doc_file):
    """
    Loads the document contents from the first file.

    :param doc_file: Path to the document file (document ID <TAB> document contents).
    :return: A dictionary {document_id: document_contents}.
    """
    max_size = sys.maxsize
    csv.field_size_limit(max_size)


    documents = {}
    with open(doc_file, 'r', encoding='utf-8') as f:
        reader = csv.reader(f, delimiter='\t')
        for row in reader:
            if len(row)==0: continue
            doc_id, content = row
            documents[doc_id] = content
    return documents

docs = [] #
doc_file = '/content/drive/My Drive/{}meetings.tsv'.format(FOLDERNAME)
documents = load_documents(doc_file)

for doc_id in documents:
    doc = Document(page_content=documents[doc_id])
    metadata = {'source': doc_id}
    doc.metadata = metadata
    docs.append(doc)

print(f"Total meetings (docs): {len(documents)}")

```

 Total meetings (docs): 230

✓ Retriever - Building the retriever

The **retriever functions like a search engine**: given a user query, it returns relevant documents from the knowledge base.

These documents are then used by the Reader model to generate an answer. In this assignment, however, we are only focusing on the retriever, not the Reader model.

Our goal: Given a user question, find the most relevant documents from the knowledge base.

Key parameters:

- **top_k**: The number of documents to retrieve. Increasing **top_k** can improve the chances of retrieving relevant content.
- **chunk size**: The length of each document. While this can vary, avoid overly long documents, as too many tokens can overwhelm most reader models.

Langchain **offers a huge variety of options for vector databases and allows us to keep document metadata throughout the processing.**

✓ 1. Specify an Embedding Model and Visualize Document Lengths

```
# Setting the name of the embedding model to be used, specifically a pre-trained model from the Hugging Face Hub.
EMBEDDING_MODEL_NAME = "thenlper/gte-small"

# Importing the SentenceTransformer class from the sentence_transformers library, which allows us to load and work with embeddings
from sentence_transformers import SentenceTransformer

# Loading the embedding model specified by EMBEDDING_MODEL_NAME and printing its maximum sequence length.
# The maximum sequence length is the number of tokens the model can handle in a single input.
print(
    f"Model's maximum sequence length: {SentenceTransformer(EMBEDDING_MODEL_NAME).max_seq_length}"
)

# Importing AutoTokenizer from Hugging Face's transformers library to handle tokenization.
from transformers import AutoTokenizer

# Loading the tokenizer for the specified model, which converts text into tokens compatible with the model.
tokenizer = AutoTokenizer.from_pretrained(EMBEDDING_MODEL_NAME)

# Calculating the length of each document in the knowledge base by counting the number of tokens.
# Using list comprehension to create a list of token counts for each document in 'docs'.
lengths = [len(tokenizer.encode(doc.page_content)) for doc in tqdm(docs)]

# Plotting the distribution of document lengths in terms of token counts to understand the length variation in the knowledge base.
# pd.Series(lengths).hist() creates a histogram of the token counts.
fig = pd.Series(lengths).hist()

# Adding a title to the histogram for better understanding of what the plot represents.
plt.title("Distribution of document lengths in the knowledge base (in count of tokens)")

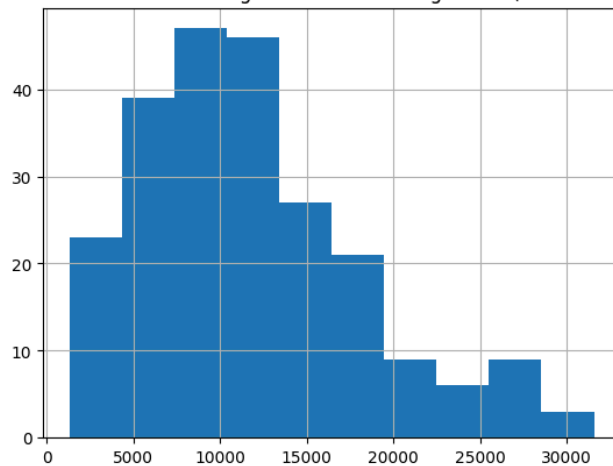
# Displaying the plot, which shows how many documents fall into different ranges of token counts.
plt.show()
```

```

/usr/local/lib/python3.10/dist-packages/sentence_transformers/cross_encoder/CrossEncoder.py:13: TqdmExperimentalWarning:
  from tqdm.autonotebook import tqdm, trange
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
  The secret `HF_TOKEN` does not exist in your Colab secrets.
  To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens),
  You will be able to reuse this secret in all of your notebooks.
  Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
modules.json: 100% 385/385 [00:00<00:00, 27.6kB/s]
README.md: 100% 68.1k/68.1k [00:00<00:00, 1.09MB/s]
sentence_bert_config.json: 100% 57.0/57.0 [00:00<00:00, 3.38kB/s]
config.json: 100% 583/583 [00:00<00:00, 31.9kB/s]
model.safetensors: 100% 66.7M/66.7M [00:00<00:00, 169MB/s]
tokenizer_config.json: 100% 394/394 [00:00<00:00, 22.6kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 3.58MB/s]
tokenizer.json: 100% 712k/712k [00:00<00:00, 5.38MB/s]
special_tokens_map.json: 100% 125/125 [00:00<00:00, 5.45kB/s]
1_Pooling/config.json: 100% 190/190 [00:00<00:00, 6.28kB/s]
Model's maximum sequence length: 512
100% 230/230 [00:17<00:00, 13.15it/s]

```

Distribution of document lengths in the knowledge base (in count of tokens)



2. Split the Documents into Chunks

The documents (meeting transcripts) are very long—some up to 30,000 tokens! To make retrieval effective, we'll **split each document into smaller, semantically meaningful chunks**. These chunks will serve as the snippets the retriever compares to the query, returning the `top_k` most relevant ones.

Objective: Create Semantically Relevant Snippets

Chunks should be long enough to capture complete ideas but not so lengthy that they lose focus.

We will use Langchain's implementation of recursive chunking with `RecursiveCharacterTextSplitter`.

- Parameter `chunk_size` controls the length of individual chunks: this length is counted by default as the number of characters in the chunk.
- Parameter `chunk_overlap` lets adjacent chunks get a bit of overlap on each other. This reduces the probability that an idea could be cut in half by the split between two adjacent chunks.

From the produced plot below, you can see that now the chunk length distribution looks better!

```

# Importing RecursiveCharacterTextSplitter from LangChain, which helps in splitting large texts into manageable chunks.
from langchain.text_splitter import RecursiveCharacterTextSplitter

```

```

# Initializing the text splitter with specified chunk size and overlap.
# chunk_size defines the maximum length of each chunk in characters.
# chunk_overlap specifies the number of overlapping characters between consecutive chunks.
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=768,

```

```

    chunk_overlap=128,
)

# Splitting each document in `docs` into smaller snippets based on the specified chunk size and overlap.
# This results in a list of document snippets, where each snippet is within the model's maximum sequence length constraints.
doc_snippets = text_splitter.split_documents(docs)

# Printing the total number of snippets created to verify the amount of data to be stored in the vector store.
print(f"Total {len(doc_snippets)} snippets to be stored in our vector store.")

# Calculating the length (in tokens) of each document snippet by encoding each snippet's content using the tokenizer.
# The lengths list stores the token counts of all snippets, which helps in understanding the distribution of snippet lengths
lengths = [len(tokenizer.encode(doc.page_content)) for doc in tqdm(doc_snippets)]

# Plotting the distribution of document snippet lengths, showing the count of tokens per snippet.
# This histogram visualizes how many snippets fall within different token length ranges.
fig = pd.Series(lengths).hist()

# Adding a title to the histogram for clarity.
plt.title("Distribution of document lengths in the knowledge base (in count of tokens)")

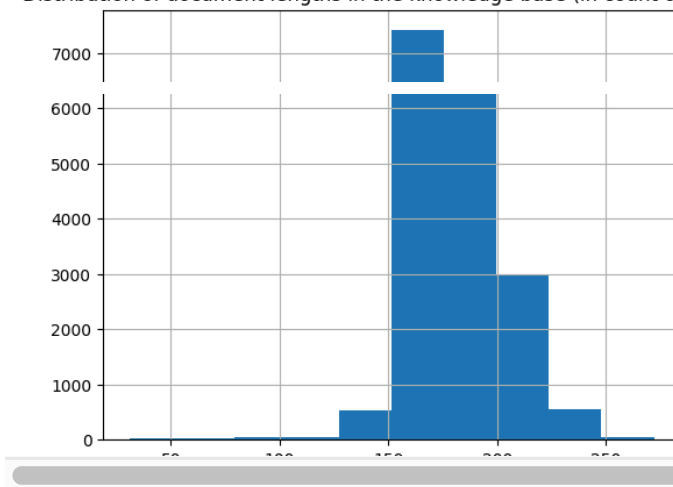
# Displaying the plot to help analyze the distribution of snippet lengths.
plt.show()

```

Total 18070 snippets to be stored in our vector store.

100% 18070/18070 [00:14<00:00, 1766.14it/s]

Distribution of document lengths in the knowledge base (in count of tokens)



3. Build the Vector Database

To enable retrieval, we need to compute embeddings for all chunks in our knowledge base. These embeddings will then be stored in a vector database.

How Retrieval Works

A query is embedded using an embedding model and a similarity search finds the closest matching chunks in the vector database.

The following cell builds the vector database consisting of all chunks in our knowledge base.

```

# Importing HuggingFaceEmbeddings, which wraps a Hugging Face model for creating document embeddings.
from langchain_huggingface import HuggingFaceEmbeddings

# Importing FAISS, a vector store from LangChain that enables efficient similarity search using FAISS (Facebook AI Similarity Search).
from langchain.vectorstores import FAISS

# Importing DistanceStrategy to define which distance metric (e.g., cosine similarity) to use in the FAISS vector store.
from langchain_community.vectorstores.utils import DistanceStrategy

# Automatically setting the device to 'cuda' if available, otherwise defaulting to 'cpu' for embedding computations.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Found device: {device}")

# Initializing the embedding model with specified parameters:
# - `model_name`: Uses the pre-trained model defined by EMBEDDING_MODEL_NAME.
# - `multi_process=True`: Enables multiprocessing to speed up embedding generation.
# - `model_kwargs={"device": device}`: Sets the device to either 'cuda' or 'cpu' based on availability.
# - `encode_kwargs={"normalize_embeddings": True}`: Normalizes embeddings to unit length, suitable for cosine similarity.

```

```

embedding_model = HuggingFaceEmbeddings(
    model_name=EMBEDDING_MODEL_NAME,
    multi_process=True,
    model_kwargs={"device": device},
    encode_kwargs={"normalize_embeddings": True}, # Set `True` for cosine similarity
)

# Recording the start time to calculate the total time taken to build the vector database.
start_time = time.time()

# Building the FAISS vector store from the document snippets using the embedding model:
# - `doc_snippets`: List of document snippets created by the text splitter, each needing an embedding.
# - `embedding_model`: The Hugging Face model used to generate embeddings for each document snippet.
# - `distance_strategy=DistanceStrategy.COSINE`: Specifies cosine similarity as the distance metric for retrieving similar d
KNOWLEDGE_VECTOR_DATABASE = FAISS.from_documents(
    doc_snippets, embedding_model, distance_strategy=DistanceStrategy.COSINE
)

# Recording the end time after the vector database has been created.
end_time = time.time()

# Calculating the elapsed time in minutes and printing it for reference.
elapsed_time = (end_time - start_time) / 60
print(f"Time taken: {elapsed_time} minutes")

↗ Found device: cuda
Time taken: 1.362663165728251 minutes

```

✓ 4. Querying the Vector Database

Using LangChain's vector database, the function `vector_database.similarity_search(query)` implements a Bi-Encoder (covered in class), independently encoding the query and each document into a single-vector representation, allowing document embeddings to be precomputed.

Let's define the Bi-Encoder ranking function and then use it on a sample query from the QMSum dataset.

```

# Define the function for ranking documents given a user query.
def rank_documents_biencoder(user_query, top_k=5):
    """
    Function for document ranking based on the query.

    :param user_query: The user query to retrieve documents for.
    :param top_k: The number of top documents to retrieve, default is 5.
    :return: A list of document IDs ranked based on the query.
    """

    # Use the FAISS vector database to perform similarity search.
    # `query=user_query` is the input query, and `k=top_k` limits results to the top-k most similar documents.
    retrieved_docs = KNOWLEDGE_VECTOR_DATABASE.similarity_search(query=user_query, k=top_k)

    # Initialize an empty list to store the ranked document IDs.
    ranked_list = []

    # Iterate over the retrieved documents to extract their IDs.
    for i, doc in enumerate(retrieved_docs):
        # Extract the 'source' metadata (document ID) for each retrieved document and append it to ranked_list.
        ranked_list.append(retrieved_docs[i].metadata['source'])

    # Return the list of ranked document IDs based on similarity to the query.
    return ranked_list # ranked document IDs.

# Define a sample query.
user_query = "what did kirsty williams am say about her plan for quality assurance ?"

# Retrieve the top-5 ranked documents for the given query.
retrieved_docs = rank_documents_biencoder(user_query)

# Print out the retrieved document IDs.
print("\n=====Top-5 documents=====")
print("\nRetrieved documents:", retrieved_docs)
print("\n=====")

↗ =====
Retrieved documents: ['doc_211', 'doc_2', 'doc_43', 'doc_160', 'doc_43']
=====

```

✓ 5. TODO: Implementation of ColBERT as a Reranker for a Bi-Encoder (35 points)

The Bi-Encoder's ranking for the sample query is not optimal: the ground truth document is not ranked at position 1, instead the document ID, **doc_211** is ranked at position 1. To determine the correct document ID for this query, refer to the `questions_answers.tsv` file.

In this task, you will implement the [ColBERT](#) approach by Khattab and Zaharia. We'll use a simplified version of ColBERT, focusing on the following key steps:

1. Retrieve the top ($K = 15$) documents for query (q) using the Bi-Encoder.
2. Re-rank these top ($K = 15$) documents using ColBERT's fine-grained interaction scoring. This will involve:
 - Using frozen BERT embeddings from a HuggingFace BERT model (no training is required, thus our version is not expected to work as well as full-fledged ColBERT).
 - Calculating scores based on fine-grained token-level interactions between the query and each document.
3. Implement the method `rank_documents_finegrained_interactions()` to perform this re-ranking.
 - Test your method on the same query as in the cell from #4 above.
 - Print out the entire re-ranked document list of 5 document IDs, as done in #4 above (the code below does it for you)
4. Ensure that your ColBERT implementation ranks the correct document at position 1 for the sample query.

Note: Since the same document is divided into multiple chunks that retain the original document ID, you may see the same document ID appear multiple times in your top_k results. However, each instance refers to a different chunk of the document's content.

Note2: For this PA we are not focused on query latency, just the late interactions part in the ColBERT approach. Thus, we don't have to pre-compute document matrix representations for ColBERT.

Note3: Both the bi-encoder and the ColBERT used in this PA are not trained for retrieval and their performance is therefore not SOTA. What we are doing is **zero-shot transfer for retrieval from BERT**. For SOTA retrieval performance both have to be trained on data of the form (q, doc+, docs-). For this PA, it is best to leave the setup as is for grading purposes. But you can certainly explore for your own purposes.

```
import torch
import torch.nn.functional as F
from transformers import AutoTokenizer, AutoModel

# Load tokenizer and model BERT from HuggingFace
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")

def rank_documents_finegrained_interactions(user_query, shortlist=15, top_k=5):
    """
    Rerank the top-K=15 retrieved documents from Bi-encoder using fine-grained token-level interactions
    and return the top_k=5 most similar documents.

    Args:
    - user_query (str): The user query string.
    - shortlist (list): Number of documents in the longer short list
    - top_k (int): Number of top reranked documents to return.

    Returns:
    - ranked_list of document IDs.
    """

    # Get initial retrieved documents
    retrieved_docs_fine = KNOWLEDGE_VECTOR_DATABASE.similarity_search(query=user_query, k=shortlist)

    # Tokenize the user query
    query_inputs = tokenizer(user_query, return_tensors='pt', truncation=True,
                             max_length=512, padding=True)

    # Get query token embeddings from BERT
    with torch.no_grad():
        query_embeddings = model(**query_inputs).last_hidden_state # Shape: (1, seq_len_query, hidden_dim)

    ranked_scores = [] # Store (doc, score) pairs

    # Process each document
    with torch.no_grad():
        for doc in retrieved_docs_fine:
            # Tokenize and get embeddings for document
            doc_inputs = tokenizer(doc.page_content, return_tensors='pt',
                                   truncation=True, max_length=512, padding=True)
            doc_embeddings = model(**doc_inputs).last_hidden_state # Shape: (1, seq_len_doc, hidden_dim)

            # Compute similarity matrix between query and document tokens
            # Shape: (1, seq_len_query, seq_len_doc)
            similarity_matrix = torch.bmm(query_embeddings, doc_embeddings.transpose(-2, -1))
```



```

# MaxSim operation: get maximum similarity for each query token
max_similarities = similarity_matrix.max(dim=-1)[0]

# Sum over query tokens for final document score
doc_score = max_similarities.sum().item()

ranked_scores.append((doc, doc_score))

# Sort documents by score in descending order
ranked_docs = sorted(ranked_scores, key=lambda x: x[1], reverse=True)

# Get top_k document IDs
ranked_list = [doc[0].metadata['source'] for doc in ranked_docs[:top_k]]

return ranked_list

# Example usage
user_query = "what did kirsty williams am say about her plan for quality assurance ?"
retrieved_docs_finegrained = rank_documents_finegrained_interactions(user_query=user_query, top_k=15)

print("\n=====Top-5 documents=====")
print("\n\nRetrieved documents:", retrieved_docs)
print("\n\nFinegrained Retrieved documents:", retrieved_docs_finegrained[:5])
print("\n=====")

```

```

tokenizers_config.json: 100% 48.0/48.0 [00:00<00:00, 3.28kB/s]
config.json: 100% 570/570 [00:00<00:00, 41.9kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.82MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 24.6MB/s]
model.safetensors: 100% 440M/440M [00:01<00:00, 230MB/s]

```

```
=====Top-5 documents=====
```

```
Retrieved documents: ['doc_211', 'doc_2', 'doc_43', 'doc_160', 'doc_43']
```

```
Finegrained Retrieved documents: ['doc_2', 'doc_102', 'doc_2', 'doc_160', 'doc_160']
```

```
=====
```

6. TODO: ColBERT Max vs. Mean Pooling for Relevance Scoring of Documents: (5 points)

ColBERT uses a form of **max pooling**, where each query term's contribution to the relevance score of a document is determined by its maximum similarity to any document term. One alternative approach is **mean pooling**, where each query term's contribution is calculated as the average similarity across all document terms.

Discuss the merits and potential limitations of using mean pooling versus max pooling in ColBERT. In your answer, consider how each approach might affect retrieval accuracy, sensitivity to specific token matches, interpretability, and anything you deem relevant. You are welcome to use an analysis of ColBERT's performance on the provided sample query in your discussion, but this is not required.

Explanation

-> Max pooling in ColBERT focuses on the most relevant document term for each query term, highlighting highly specific matches that are often key to determining relevance. This makes it suitable for capturing sharp, precise alignments but might overlook broader contextual relationships.

-> On the other hand, mean pooling considers the average similarity across all document terms, offering a more balanced view of the document's overall relevance. While this can better reflect the document's general alignment with the query, it risks diluting the importance of critical matches.

-> Ultimately, max pooling tends to prioritize precision for specific token matches, while mean pooling favors a more holistic but potentially less precise approach, impacting retrieval accuracy and interpretability.

Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Runtime -> Run All. This will run all the cells in order, and will take several minutes.

3. Once you've rerun everything, save a PDF version of your notebook. Make sure all your code and answers are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
4. Look at the PDF file and make sure all your code and answers are there, displayed correctly. The PDF is the only thing your graders will see!
5. Submit your PDF on Gradescope.

✓ 7. (Optional) Full evaluation pipeline for your own exploration.

For this assignment, we only ask you to explore one sample query. Running on many queries is super slow without the right compute. If you have compute/and/or time to wait, below is a more complete evaluation setup that works with all the queries in QMSum dataset, and reports the precision@k=5 metric.

Note: you need to remove the comment markers from the code below.

```
# def load_questions_answers(qa_file):
#     """
#     Loads the questions and corresponding ground truth document IDs.

#     :param qa_file: Path to the question-answer file (document ID <TAB> question <TAB> answer).
#     :return: A list of tuples [(document_id, question, answer)].
#     """
#     qa_pairs = []
#     with open(qa_file, 'r', encoding='utf-8') as f:
#         reader = csv.reader(f, delimiter='\t')
#         for row in reader:
#             doc_id, question, answer = row
#             qa_pairs.append((doc_id, question, answer))

#     random.shuffle(qa_pairs)

#     return qa_pairs

# def precision_at_k(ground_truth, retrieved_docs, k):
#     """
#     Computes Precision at k for a single query.

#     :param ground_truth: The name of the ground truth document.
#     :param retrieved_docs: The list of document names returned by the model in ranked order.
#     :param k: The cutoff for computing Precision.
#     :return: Precision at k.
#     """
#     return 1 if ground_truth in retrieved_docs[:k] else 0

# def evaluate(doc_file, qa_file, ranking_fuction = None, k= 5):
#     """
#     Evaluate the retrieval system based on the documents and question-answer pairs.

#     :param doc_file: Path to the document file.
#     :param qa_file: Path to the question-answer file.
#     :param k: The cutoff for Precision@k.
#     """
#     # Load the QA pairs
#     qa_pairs = load_questions_answers(qa_file)
```