Q & A System using RAG

Objective

To create a Q & A system using RAG technique.

Basic Idea of implementation

A query will be obtained from the user and introduced into the retriever system. The retriever system will then query our dataset (reference text) to obtain more accurate results for the given query. This result, combined with the knowledge already possessed by the LLM (generator system) from its training, will generate an answer that will be provided to the user.

The question + the reference text + knowledge of the pre-trained LLM = generated response for the user.

About BERT (developed by Google AI)

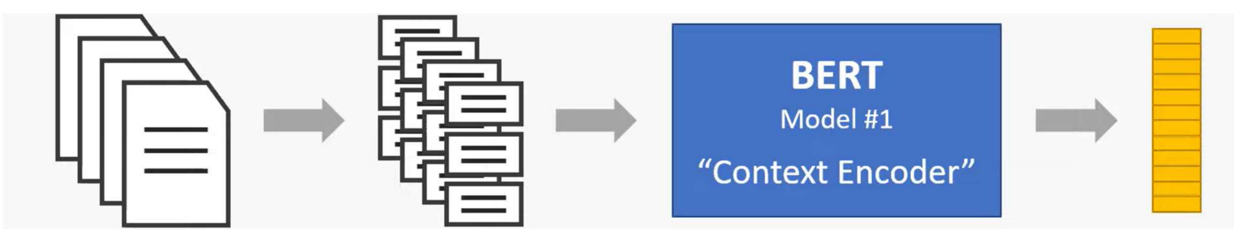
BERT reads the entire sequence of words at once, allowing it to understand the context from both directions. This bidirectional approach enables BERT to capture richer contextual information. The Transformer uses self-attention mechanisms to weigh the importance of different words in a sentence, allowing the model to focus on relevant parts of the text.

About BART (developed by Facebook AI)

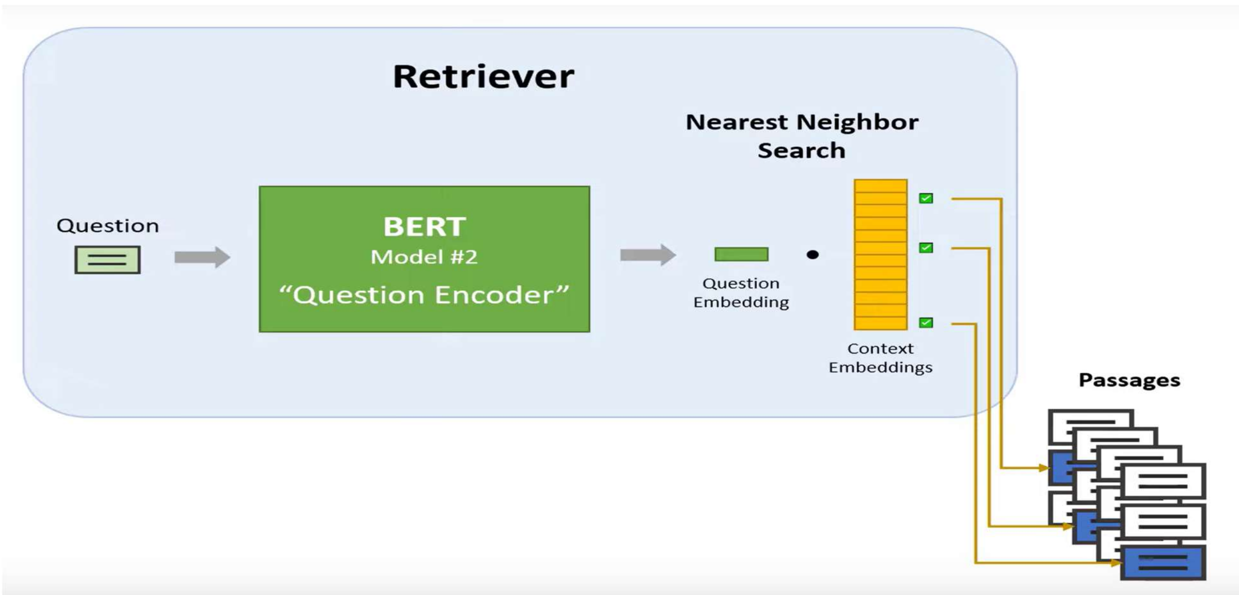
It combines elements of BERT and GPT and uses a standard transformer-based encoder-decoder architecture, similar to those used in sequence-to-sequence models. The encoder maps an input sequence to a continuous representation, and the decoder generates an output sequence from this representation.

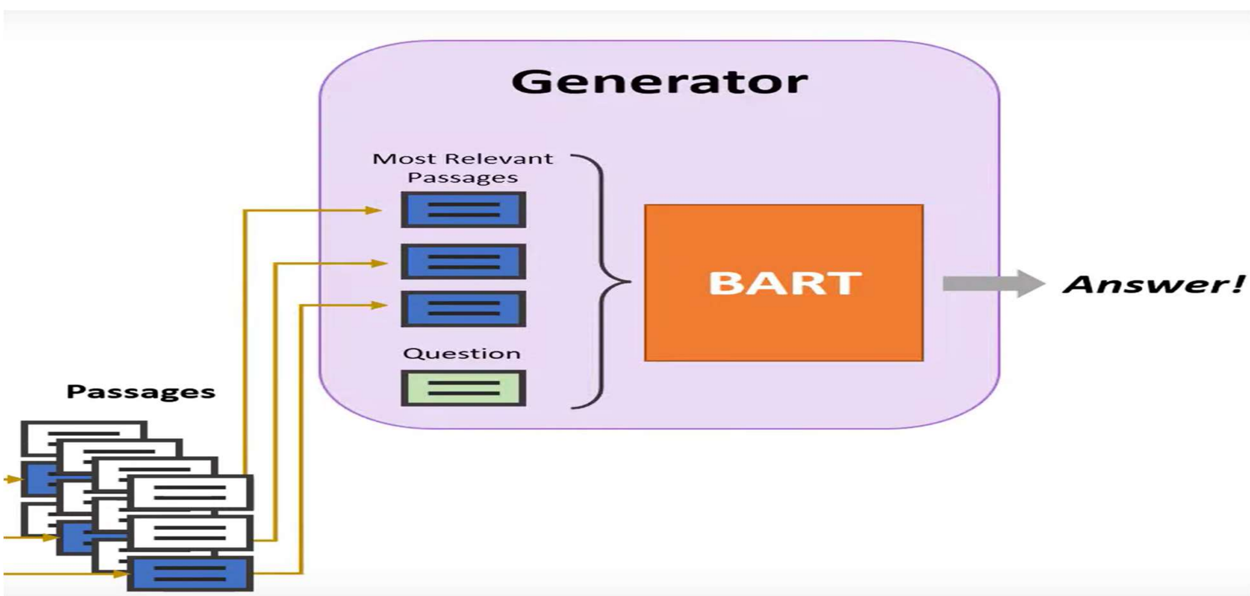
Pre-processing

First the context (reference texts) is divided into several passages which are, in turn, fed to the BERT model #1 (context encoder). This model will generate embeddings of this context.



About Dense Passage Retrieval
This is a technique used in natural language processing (NLP) for retrieving relevant passages of text from a large corpus in response to a query. It is particularly useful in open-domain question answering systems where the goal is to find the most relevant pieces of information from a vast amount of text data.





[SEP] token can be useful to determine a title for each respective passage.

Extracting text from PDF files manually.

```
[71]: import gdown
      import PyPDF2
      import os
      import urllib
      import torch
```

```
[77]: # List of file paths
      file_paths = [
          r'C:\\Users\\ankit\\Downloads\\RAG Project\\Retrieval Augmented Generation.pdf'
      ]

      # Initialize a dictionary to store the content of each PDF
      pdf_contents = {}

      # Iterate over each file path
      for file_path in file_paths:
          with open(file_path, 'rb') as file:
              reader = PyPDF2.PdfReader(file)
              num_pages = len(reader.pages)

              content = ""
              for page_num in range(num_pages):
                  page = reader.pages[page_num]
                  content += page.extract_text()

              # Store the content in the dictionary with the file name as the key
              pdf_contents[file_path] = content

      # Print the extracted text from each PDF
      for file_path, content in pdf_contents.items():
          print(f"Content of {file_path}:\n")
          print(content)
          print("\n" + "-"*80 + "\n")
```

Fetching the files from Google Drive.

```
[24]: output = 'contextFiles.pdf'
      fileID = '1BWozL5mWl94JwepL2M3qk_O01bpq6BQG'
      gdown.download('https://drive.google.com/uc?id=' + fileID, output, quiet=False)
      print('The file has been retrieved from Google Drive')
```

```
Downloading...
From: https://drive.google.com/uc?id=1BWozL5mWl94JwepL2M3qk_O01bpq6BQG
To: C:\Users\ankit\Downloads\RAG Project\contextFiles.pdf
100%|██████████████████████████████████████████████████| 961k/961k [00:00<00:00, 2.85MB/s]
The file has been retrieved from Google Drive
```

Extracted Text.

```
Okapi BM25  (BM – Best Matching) is a powerful ranking algorithm and valuable tool for enhancing
search relevance and delivering more accurate and useful user results. It is a bag-of-words
retrieval function that ranks a set of documents based on the query terms appearing in each
document, regardless of their proximity within the document.


TF-IDF (Term Frequency-Inverse Document Frequency): A statistical measure used to evaluate the
importance of a word in a document relative to a collection of documents. It combines term
frequency (how often a term appears in a document) and inverse document frequency (how
common or rare a term is across all documents).

Extending Familiarity towards few Technical terms
1. Index
It is a data structure that stores and organizes the vectors (arrays of information in numerical form)
of document/information pieces in a way that helps us to retrieve efficiently. We can avoid the need
to scan the entire dataset, in turn increasing the efficiency of this retrieval step by using the indexes.
Prompt/Query                      Converted to vectors                 This vector is compared to the vector (V-DB)
Documents                            Converted to vectors                        Stored in vector DB
```

Detecting the number of context files to be read for future storage.

```
titles = []
articles = []
print('The files are being read.\n')
i = 0

# Scan each file in the directory
for filename in os.listdir("Files"):
    if not filename.endswith('.txt'):  # Use endswith() for more robust check
        continue
    with open("Files/" + filename, "rb") as f:
        title = urllib.parse.unquote(filename[:-4])  # Decode any characters not allowed in URLs
        title = title.replace('_', ' ')  # Replace underscores with spaces
        if not title.strip():  # Check if the title is empty after stripping whitespace
            print('Empty title for', filename)
            continueSS
        titles.append(title)

        # Read the file using different encodings until one succeeds
        for encoding in ['utf-8', 'latin-1']:  # Add more encodings if needed
            try:
                article = f.read().decode(encoding)
                break  # Stop trying encodings if successful
            except UnicodeDecodeError:
                pass  # Try the next encoding

        articles.append(article)  # Append the decoded article
    i += 1
    if (i % 500) == 0:
        print(' Processed {:,}'.format(i))
print('DONE.\n')
print('There are {:,} articles.'.format(len(articles)))
```

```
The files are being read.

DONE.

There are 2 articles.
```

Differentiating titles from the passages.

```
[53]: titles[0:5]
```

```
[53]: ['Generative pre-trained transformers Text',
 'Retrieval Augmented Generation Text']
```

```
[55]: articles[1]
```

```
[55]: 'Retrieval Augmented Generation (RAG) \r\n \r\nAnalysis \r\nWhen a client provides a prompt to a LLM, the LLM uses its own knowledge that it collected duri
ng its training and compares it to the content from a source document (it might be online or offline), and \r\nforms a response that is provided to the cli
ent. This process ensures that the response has more \r\naccurate and updated information. \r\nA Retrieval Augmented Generation (RAG) system is split into
3 main components: \r\nIngestion: clean, chunk, embed, and load your data to a vector DB  \r\nRetrieval: query your vector DB for context  \r\nGeneration:
attach the retrieved context to your prompt and pass it to an LLM  \r\n\r\nFew Useful algorithms  \r\n \r\nOkapi BM25  (BM – Best Matching) is a powerful r
anking algorithm and valuable tool for enhancing \r\nsearch relevance and delivering more accurate and useful user results. It is a bag-of-words \r\nretrie
val function that ranks a set of documents based on the query terms appearing in each \r\ndocument, regardless of their proximity within the document. \r\n
\r\nTF-IDF (Term Frequency-Inverse Document Frequency): A statistical measure used to evaluate the \r\nimportance of a word in a document relative to a col
lection of documents. It combines term \r\nfrequency (how often a term appears in a document) and inverse document frequency (how \r\ncommon or rare a term
is across all documents). \r\n \r\nExtending Familiarity towards few Technical terms \r\n1. Index \r\nIt is a data structure that stores and organizes the
vectors (arrays of information in numerical form) \r\nof document/information pieces in a way that helps us to retrieve efficiently. We can avoid the need
\r\nto scan the entire dataset, in turn increasing the efficiency of this retrieval step by using the indexes. \r\n \r\n2. Vector Databases  \r\nA vector d
atabase is used to find a similar item related to a query. This is done by calculating the \r\ndistance between vectors in the multi-dimensional space. The
two vectors are prompt-converted to-\r\nvector and documents-converted to-vector. If these both vectors are close to each other then they \r\nare considere
d similar. Consider vector database as a database full of embeddings. \r\n \r\nIn this image, we can see the array of distances where smaller differences i
ndicate a higher degree \r\nof similarity. \r\n  \r\n3. LlamaIndex  \r\nThis is a framework for connecting your data to LLMs and getting the results into p
roduction. \r\nFor example, we start with a source data like a PDFs, APIs, SQL or any other document and we would \r\nlike to store this data in our LLM so
that we can unlock the capabilities of, for instance, ChatGPT. \r\nThis involves loading the data from somewhere and putting it into a storage system. So d
ata structure \r\nis a part where we index, process and embed your data so that it can be retrieved by a language \r\nmodel and retrieval system later on.
\r\nThe next phase is retrieval and query interface where, given that the data is processed and stored in \r\nsomething like a vector database like active
Loop, we can then perform retrieval to fetch relevant \r\ncontexts. This includes QA, Summarization and more. \r\n\r\nCurrent RAG Stack for building a QA S
ystem \r\n\r\nProcess: \r\n1. Split up documents into even chunks. \r\n2. Each chunk is a piece of raw text. \r\n3. Generate embedding for each chunk (exam
ple – OpenAI embeddings, sentence \r\ntransformer). \r\n4. Store each chunk into a vector database. \r\n\r\nAbout Embeddings: \r\n1. Creating Embeddings
\r\nText to Vector: An embedding converts a text into a vector (a list of numbers) that captures \r\nthe essential meaning of the text. For example, the se
ntence "The cat sat on the mat" might \r\nbe represented as a vector like [0.1, 0.3, 0.7, ...]. \r\nPre-trained Models: OpenAI provides models that can gen
erate these embeddings, trained \r\non large datasets to understand language context and semantics. \r\n \r\n2. Usage in RAG \r\nDocument Embeddings: All d
ocuments in the database are converted into embeddings and \r\nstored in a vector index. \r\nQuery Embeddings: When a user submits a query, it is also conv
erted into an embedding \r\nusing the same OpenAI model. \r\nSimilarity Search: The query embedding is compared against the document embeddings in \r\nthe
vector index to find the most similar documents. This is often done using measures like \r\ncosine similarity or Euclidean distance. \r\n3. The request bod
y of an embedding may include the following components: \r\nID of the model to use. \r\nInput string or array (for embedding multiple inputs in a single ar
ray) \r\nA unique identifier representing your end-user. \r\n \r\nIn this image, we can see how similar objects have been grouped together in multi-dimensi
```

Splitting the articles into chunks.

```
[58]: print('Before splitting, {:,} articles.\n'.format(len(titles)))
passage_titles = []
passages = []
print('Splitting into chunks.')
for i in range(len(titles)):
    title = titles[i]
    article = articles[i]
    if len(article) == 0:
        print('Skipping empty article:', title)
        continue
    words = article.split()
    for i in range(0, len(words), 100):
        chunk_words = words[i : i + 100]
        chunk = " ".join(chunk_words)
        chunk = chunk.strip()
        if len(chunk) == 0:
            continue
        passage_titles.append(title)
        passages.append(chunk)
print('Splitting done.\n')
chunked_corpus = {'title': passage_titles, 'text':passages}
print('After splitting, {:,} "passages".'.format(len(chunked_corpus['title'])))
```

```
Before splitting, 2 articles.

Splitting into chunks.
Splitting done.

After splitting, 21 "passages".
```

Tokenizing the content and adding IDs to respective tokens.

```
from transformers import DPRQuestionEncoderTokenizer
#question_tokenizer = DPRQuestionEncoderTokenizer.from_pretrained("facebook/dpr-question_encoder-single-nq-base")
ctx_tokenizer = DPRQuestionEncoderTokenizer.from_pretrained('facebook/dpr-ctx_encoder-multiset-base')
```

```
num_passages = len(chunked_corpus['title'])
print('Tokenizing {:,} passages for DPR...'.format(num_passages))
outputs = ctx_tokenizer(
    chunked_corpus["title"],
    chunked_corpus["text"],
    truncation = True,
    padding = "longest",
    return_tensors = "pt"
)
print('Tokenization process completed.')
input_ids = outputs["input_ids"]
```

```
Tokenizing 21 passages for DPR...
Tokenization process completed.
```

```
print(input_ids.shape)
```

```
torch.Size([21, 238])
```

GPU’s availability status.

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print('There are %d GPU(s) available.' % torch.cuda.device_count())
    print('We will use the GPU:', torch.cuda.get_device_name(0))
else:
    print('No GPU is available.')
```

```
No GPU is available.
```

Due to unavailability of the GPU, I shifted my code to Google Colab.

```
[3] if torch.cuda.is_available():
        device = torch.device("cuda")
        print('There are %d GPU(s) available.' % torch.cuda.device_count())
        print('We will use the GPU:', torch.cuda.get_device_name(0))
    else:
        print('No GPU is available.')
```

```
There are 1 GPU(s) available.
We will use the GPU: Tesla T4
```

I transferred the DPR to the GPU.

```
[6] # Import the DPRContextEncoder class from the transformers library
    from transformers import DPRContextEncoder

    # Initialize the DPRContextEncoder model from the pretrained 'facebook/dpr-ctx_encoder-multiset-base' model
    ctx_encoder = DPRContextEncoder.from_pretrained("facebook/dpr-ctx_encoder-multiset-base")

    # Move the DPRContextEncoder model to the specified device (GPU if available, otherwise CPU)
    ctx_encoder = ctx_encoder.to(device=device)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: 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), set it as secret in your Google Colab and restart your session
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(
config.json: 100% 492/492 [00:00<00:00, 33.0kB/s]
pytorch_model.bin: 100% 438M/438M [00:10<00:00, 55.1MB/s]
Some weights of the model checkpoint at facebook/dpr-ctx_encoder-multiset-base were not used when initializing DPRContextEncoder: ['ctx_encoder.bert_model.pooler.dense.bias', 'ctx_enc
- This IS expected if you are initializing DPRContextEncoder from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceCl
- This IS NOT expected if you are initializing DPRContextEncoder from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification m
```

Transfer the input IDs to the GPU as well and we move further to collect embeddings in batches.

```
[11] import time
     torch.set_grad_enabled(False)

     t0 = time.time()
     step = 0
     batch_size = 16
     num_passages = input_ids.size()[0]  # Assuming input_ids is defined earlier as the tokenized passage inputs
     num_batches = math.ceil(num_passages / batch_size)
     embeds_batches = []

     print('Generating embeddings for {:,} passages...'.format(num_passages))

     for i in range(0, num_passages, batch_size):
         batch_ids = input_ids[i:i + batch_size].to(device)  # Get the batch input IDs and move them to the device

         outputs = ctx_encoder(batch_ids, return_dict=True)
         embeddings = outputs["pooler_output"].cpu().numpy()  # Move embeddings to CPU and convert to numpy array
         embeds_batches.append(embeddings)  # Collect embeddings in batches

         step += 1

         if step % 100 == 0:
             elapsed = format_time(time.time() - t0)
             print('Batch {:>5,} of {:>5,}. Elapsed: {:}.'.format(step, num_batches, elapsed))

     print('Done.')

     # Concatenate all the embedding batches into a single array
     embeddings = np.concatenate(embeds_batches, axis=0)

     # Print the size of the dataset embeddings to verify the shape of the concatenated array
     print('Size of dataset embeddings:', embeddings.shape)
```

```
Generating embeddings for 21 passages...
Done.
Size of dataset embeddings: (21, 768)
```

Later, we can use FAISS (Facebook AI Similarity Search). This library helps us perform k-Nearest Neighbor (kNN) search. Here, we are using HNSW index for similarity search.

But, if we are using the cosine similarity as a metric, we can refer to kNN as Maximum Inner-Product Search (MIPS).

```
[12] # Concatenate all the embedding batches into a single array
     # The concatenation is done along the first axis (rows) to create a single array of embeddings
     embeddings = np.concatenate(embeds_batches, axis=0)

     # Print the size of the dataset embeddings to verify the shape of the concatenated array
     print('Size of dataset embeddings:', embeddings.shape)
```

```
Size of dataset embeddings: (21, 768)
```

```
# Set the dimensionality of the vectors to be indexed
dim = 768

# Set the number of neighbors for the HNSW (Hierarchical Navigable Small World) index
m = 128

# Create a HNSW index for dense vectors using inner product as the similarity metric
# This index type is useful for approximate nearest neighbor search
index = faiss.IndexHNSWFlat(dim, m, faiss.METRIC_INNER_PRODUCT)
```

All the embedded content is then added to the FAISS index.

```
[14] # Print a message indicating the start of building the FAISS index
     print('Building of the FAISS index is in progress.')

     # Record the current time to measure the duration of the indexing process
     t0 = time.time()

     # Train the FAISS index with the embeddings
     index.train(embeddings)

     # Add the embeddings to the FAISS index
     index.add(embeddings)

     # Print a message indicating the completion of the indexing process
     print('Done.')

     # Print the time taken to add the embeddings to the index
     print('Adding embeddings to index took', format_time(time.time() - t0))
```

```
Building of the FAISS index is in progress.
Done.
Adding embeddings to index took 0:00:00
```

We can choose to use the DPRQuestionEncoderTokenizer and DPRQuestionEncoder from the transformers library. DPRQuestionEncoderTokenizer tokenizes questions into the format required by the DPR question encoder. DPRQuestionEncoder encodes questions into dense vector representations.

```
[15] # Import the DPRQuestionEncoder class from the transformers library
     from transformers import DPRQuestionEncoder

     # Initialize the question encoder with the pretrained 'facebook/dpr-question_encoder-multiset-base' model
     q_encoder = DPRQuestionEncoder.from_pretrained("facebook/dpr-question_encoder-multiset-base")

     # Move the question encoder model to the specified device (GPU or CPU)
     q_encoder = q_encoder.to(device=device)
```

```
Some weights of the model checkpoint at facebook/dpr-question_encoder-multiset-base were not used when initializing DPRQuestionEncoder: ['question_encoder.bert_model.pooler.dense.bias', 'question_encoder.be
- This IS expected if you are initializing DPRQuestionEncoder from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model frc
- This IS NOT expected if you are initializing DPRQuestionEncoder from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSec
```

```
[17] from transformers import DPRQuestionEncoderTokenizer, DPRQuestionEncoder

     # Initialize the question tokenizer with the pretrained 'facebook/dpr-question_encoder-multiset-base' model
     q_tokenizer = DPRQuestionEncoderTokenizer.from_pretrained("facebook/dpr-question_encoder-multiset-base")

     # Initialize the question encoder with the pretrained 'facebook/dpr-question_encoder-multiset-base' model
     q_encoder = DPRQuestionEncoder.from_pretrained("facebook/dpr-question_encoder-multiset-base")

     # Move the question encoder model to the specified device (GPU or CPU)
     q_encoder = q_encoder.to(device)

     # Encode the query into input IDs using the question tokenizer, returning PyTorch tensors
     input_ids = q_tokenizer.encode("How many models have been created by Cerebras?", return_tensors = "pt")

     # Move the input IDs tensor to the specified device (GPU or CPU)
     input_ids = input_ids.to(device)

     # Pass the input IDs through the question encoder to obtain the outputs
     outputs = q_encoder(input_ids)

     # Extract the query embedding from the outputs
     q_embed = outputs['pooler_output']

     # Move the query embedding tensor to the CPU and convert it to a NumPy array
     q_embed = q_embed.cpu().numpy()

     # Print the shape of the query embedding
     print("Query embedding:", q_embed.shape)
```

```
tokenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 1.84kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 3.52MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 5.15MB/s]
Some weights of the model checkpoint at facebook/dpr-question_encoder-multiset-base were not used when initializing DPRQuestionEncoder: ['question_encoder.bert_model.pooler.dense.bias', 'question_encoder.bert_model.pooler.dense.weigh
- This IS expected if you are initializing DPRQuestionEncoder from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining mode
- This IS NOT expected if you are initializing DPRQuestionEncoder from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
Query embedding: (1, 768)
```

Then, we retrieve 3 closest matches by searching the FAISS index.

```
[18] # Search the FAISS index with the query embedding to find the k closest matches
     D, I = index.search(q_embed, k=3)

     # Print the indices of the closest matching passages
     print('Closest matching indices:', I)

     # Print the inner product scores of the closest matches
     print('Inner products:', D)
```

```
Closest matching indices: [[1 6 2]]
Inner products: [[67.11247 65.05115 64.29021]]
```

Here, we can observe that all the most relevant passages from the correct article have been fetched.

```
# Initialize a text wrapper to format the passage text to a specified width
wrapper = textwrap.TextWrapper(width=80)

# Iterate over the indices of the closest matching passages
for i in I[0]:
  # Print the index of the matching passage
  print('Index:', i)

  # Retrieve the title of the matching passage from the chunked corpus
  title = chunked_corpus['title'][i]

  # Retrieve the text of the matching passage from the chunked corpus
  passage = chunked_corpus['text'][i]

  # Print the title of the matching article
  print('Article Title: ', title, '\n')

  # Print the formatted passage text
  print('Passage:')
  print(wrapper.fill(passage))
  print('')
```

```
Index: 1
Article Title:  Generative pre-trained transformers Text

Passage:
these was significantly more capable than the previous, due to increased size
(number of trainable parameters) and training. The most recent of these, GPT-4,
was released in March 2023.[11] Such models have been the basis for their more
task-specific GPT systems, including models fine-tuned for instruction
following—which in turn power the ChatGPT chatbot service.[1] The term "GPT" is
also used in the names and descriptions of such models developed by others. For
example, other GPT foundation models include a series of models created by
EleutherAI,[12] and seven models created by Cerebras in 2023.[13] Also,
companies in different industries have developed

Index: 6
Article Title:  Generative pre-trained transformers Text

Passage:
whereas ChatGPT is further trained for conversational interaction with a human
user.[30][31] OpenAI's most recent GPT foundation model, GPT-4, was released on
March 14, 2023. It can be accessed directly by users via a premium version of
ChatGPT, and is available to developers for incorporation into other products
and services via OpenAI's API. Other producers of GPT foundation models include
EleutherAI (with a series of models starting in March 2021)[12] and Cerebras
(with seven models released in March 2023).[13]
```


```
Index: 1
Article Title:  Generative pre-trained transformers Text

Passage:
these was significantly more capable than the previous, due to increased size
(number of trainable parameters) and training. The most recent of these, GPT-4,
was released in March 2023.[11] Such models have been the basis for their more
task-specific GPT systems, including models fine-tuned for instruction
following—which in turn power the ChatGPT chatbot service.[1] The term "GPT" is
also used in the names and descriptions of such models developed by others. For
example, other GPT foundation models include a series of models created by
EleutherAI,[12] and seven models created by Cerebras in 2023.[13] Also,
companies in different industries have developed

Index: 6
Article Title:  Generative pre-trained transformers Text

Passage:
whereas ChatGPT is further trained for conversational interaction with a human
user.[30][31] OpenAI's most recent GPT foundation model, GPT-4, was released on
March 14, 2023. It can be accessed directly by users via a premium version of
ChatGPT, and is available to developers for incorporation into other products
and services via OpenAI's API. Other producers of GPT foundation models include
EleutherAI (with a series of models starting in March 2021)[12] and Cerebras
(with seven models released in March 2023).[13]
```


```
Index: 2
Article Title:  Generative pre-trained transformers Text

Passage:
task-specific GPTs in their respective fields, such as Salesforce's
"EinsteinGPT" (for CRM)[14] and Bloomberg's "BloombergGPT" (for finance).[15]
History Initial developments Generative pretraining (GP) was a long-established
concept in machine learning applications.[16][17][18] It was originally used as
a form of semi-supervised learning, as the model is trained first on an
unlabelled dataset (pretraining step) by learning to generate datapoints in the
dataset, and then it is trained to classify a labelled dataset.[19] While the
unnormalized linear transformer dates back to 1992,[20][21][22] the modern
transformer architecture was not available until 2017 when it was published by
researchers at Google in a paper
```

We convert the chunk of the corpus in a DataFrame to give it a structured display.

```
[20] chunked_corpus = {'title': passage_titles, 'text': passages}
```

```
[21] # Create a DataFrame from the chunked_corpus dictionary
     # This converts the chunked_corpus (a dictionary with titles and texts) into a pandas DataFrame
     df = pd.DataFrame(chunked_corpus)

     # Convert the pandas DataFrame into a Dataset object from the datasets library
     # This allows for easy handling and manipulation of the data in the dataset format
     dataset = Dataset.from_pandas(df)

     # Print the dataset to verify its contents
     # This displays the structure and contents of the dataset
     print(dataset)
```

```
Dataset({
    features: ['title', 'text'],
    num_rows: 21
})
```

```
# Initialize an empty list to store the embeddings
embs = []

# Iterate over each row in the embeddings matrix
for i in range(embeddings.shape[0]):
  # Append the i-th embedding (a row from the matrix) to the embs list
  embs.append(embeddings[i, :])
```

```
[23] dataset = dataset.add_column("embeddings", embs) # Add the embeddings as a new column to the dataset
     dataset # Display the dataset contents.
```

```
Dataset({
    features: ['title', 'text', 'embeddings'],
    num_rows: 21
})
```

```
[24] # Initialize the FAISS index with the specified dimensions and metric
     index = faiss.IndexHNSWFlat(dim, m, faiss.METRIC_INNER_PRODUCT)

     # Add the FAISS index to the dataset for the embeddings column
     dataset.add_faiss_index(
         column="embeddings",       # Column name in the dataset to index
         index_name="embeddings",   # Name of the index to be created
         custom_index=index,        # Custom FAISS index to use
         faiss_verbose=True         # Verbose output from FAISS (shows progress)
     )
```

```
100% 1/1 [00:00<00:00, 43.51it/s]
Dataset({
    features: ['title', 'text', 'embeddings'],
    num_rows: 21
})
```

To make things easier, we can use the Facebook AI developed RagRetriever that helps us to retrieve relevant passages from the large corpus based on our given query.

```
[25] from transformers import RagRetriever
     retriever = RagRetriever.from_pretrained(
         "facebook/rag-sequence-nq",
         use_dummy_dataset = False,
         indexed_dataset = dataset,
         index_name = "embeddings"
     )
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_download.py:1132: FutureWarning: `resume_download` is deprecated and will be removed in version 1.0.0. Downloads always resume when possible. If you want to force a new download, use `force_download=True`.
  warnings.warn(
config.json: 100% 4.60k/4.60k [00:00<00:00, 255kB/s]
(…)_encoder_tokenizer/tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 2.52kB/s]
question_encoder_tokenizer/vocab.txt: 100% 232k/232k [00:00<00:00, 3.29MB/s]
(…)ncoder_tokenizer/special_tokens_map.json: 100% 112/112 [00:00<00:00, 9.01kB/s]
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'DPRQuestionEncoderTokenizer'.
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'DPRQuestionEncoderTokenizerFast'.
(…)enerator_tokenizer/tokenizer_config.json: 100% 26.0/26.0 [00:00<00:00, 2.04kB/s]
generator_tokenizer/vocab.json: 100% 899k/899k [00:00<00:00, 15.9MB/s]
generator_tokenizer/merges.txt: 100% 456k/456k [00:00<00:00, 17.3MB/s]
(…)erator_tokenizer/special_tokens_map.json: 100% 772/772 [00:00<00:00, 31.0kB/s]
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'BartTokenizer'.
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'BartTokenizerFast'.
```

```
[26] from transformers import RagTokenizer
     tokenizer = RagTokenizer.from_pretrained("facebook/rag-sequence-nq")
```

```
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'DPRQuestionEncoderTokenizer'.
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'DPRQuestionEncoderTokenizerFast'.
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'BartTokenizer'.
The tokenizer class you load from this checkpoint is not the same type as the class this function is called from. It may result in unexpected tokenization.
The tokenizer class you load from this checkpoint is 'RagTokenizer'.
The class this function is called from is 'BartTokenizerFast'.
```

```
[27] # Import the RagSequenceForGeneration class from the transformers library
     from transformers import RagSequenceForGeneration

     # Load the pre-trained RAG sequence model from Facebook's "rag-sequence-nq"
     # Specify the retriever to be used with the model
     model = RagSequenceForGeneration.from_pretrained("facebook/rag-sequence-nq", retriever=retriever)
```

```
pytorch_model.bin: 100% 2.06G/2.06G [00:51<00:00, 45.2MB/s]
Some weights of the model checkpoint at facebook/rag-sequence-nq were not used when initializing RagSequenceForGeneration: ['rag.question_encoder.question_encoder.bert_model.pooler.dense.bias', 'rag.question_encoder.question_encoder.bert_model.pooler.dense.weight']
- This IS expected if you are initializing RagSequenceForGeneration from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing RagSequenceForGeneration from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

In the following picture, we can observe that I asked a random question related to our file named ‘Generative pre-trained transformers Text’ and the RAG model was able to give an accurate answer.

```
# Record the start time for measuring the response time
t0 = time.time()

# Define the question to be asked
question = "How many models have been created by Cerebras?"

# Tokenize the question using the question encoder tokenizer
# Convert the question into input IDs, which are numerical representations of the tokens
input_ids = tokenizer.question_encoder(question, return_tensors="pt")["input_ids"]

# Generate an answer using the RAG model
# The model generates a response based on the input IDs of the question
generated = model.generate(input_ids)

# Decode the generated answer from the model's output tokens to a readable string
generated_string = tokenizer.batch_decode(generated, skip_special_tokens=True)[0]

# Print the question and the generated answer
print("Q: " + question)
print("A: " + generated_string)

# Print the time taken to generate the response
print('\nResponse took %.2f seconds' % (time.time() - t0))
```

```
/usr/local/lib/python3.10/dist-packages/transformers/generation/utils.py:1168: UserWarning: Using the model-agnostic default `max_length` (=20) to control the generation length. We recommend setting `max_ne
  warnings.warn(
Q: How many models have been created by Cerebras?
A:  seven

Response took 177.28 seconds
```

I created a function to wrap the above steps and make it easier to implement them.

```
[29] # Define a function to ask a question and get a response from the model
     def ask_question(question):
       # Record the start time to measure the response time
       t0 = time.time()

       # Tokenize the question using the question encoder tokenizer
       # Convert the question into input IDs, which are numerical representations of the tokens
       input_ids = tokenizer.question_encoder(question, return_tensors="pt")["input_ids"]

       # Generate an answer using the RAG model
       # The model generates a response based on the input IDs of the question
       generated = model.generate(input_ids)

       # Decode the generated answer from the model's output tokens to a readable string
       generated_string = tokenizer.batch_decode(generated, skip_special_tokens=True)[0]

       # Print the question and the generated answer
       print("Q: " + question)
       print("A: '{:}'".format(generated_string))

       # Print the time taken to generate the response
       print('\nResponse took %.2f seconds' % (time.time() - t0))
```

```
[30] ask_question("How many models have been created by Cerebras?")

     Q: How many models have been created by Cerebras?
     A: ' seven'

     Response took 176.93 seconds
```

Another example from the first document ‘Retrieval Augmented Generation Text’.

```
[31] ask_question("What is Hierarchical RAG?")

     /usr/local/lib/python3.10/dist-packages/transformers/generation/utils.py:1168: UserWarning: Using the model-agnostic default `max_length` (=20) to control the generation length. We recommend setting `max_ne
       warnings.warn(
     Q: What is Hierarchical RAG?
     A: ' multi-level retrieval'

     Response took 235.99 seconds
```

Conclusion

Thus, this model was able to implement the RAG technique to create a question-answering system.