Documentation

March 10, 2025

1 LLM Fine-tuning Challenge: Enhancing Qwen 2.5 3B for AI Research QA

This notebook delivers documentation for the task from start to end which was stored in several notebooks

1.1 Generating Embeddings Vector for Chunks

Installing Major Dependancies

```
[ ]: !pip install langchain pypdf markdown numpy
     !pip install -U langchain-community
```

1.1.1 Reading and Loading Dataset

In this cell, we are reading and loading the dataset from PDF and Markdown files. We use the `glob` library to find all files with `.pdf` and `.md` extensions in the `rawdata` directory.

Libraries Used:

- `langchain.document_loaders.PyPDFLoader`: This class is used to load PDF documents. It reads the content of PDF files and converts them into a format that can be processed further.
- `langchain.document_loaders.TextLoader`: This class is used to load text documents, including Markdown files. It reads the content of text files and converts them into a format that can be processed further.
- `glob`: This module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell.

Process:

1. **Reading PDF Files**:
   - We use `glob.glob("rawdata/*.pdf")` to get a list of all PDF files in the `rawdata` directory.
   - For each PDF file, we create an instance of `PyPDFLoader` and load the documents using the `load()` method.
   - The loaded documents are then added to the `raw_data` list.
2. **Reading Markdown Files**:
   - Similarly, we use `glob.glob("rawdata/*.md")` to get a list of all Markdown files in the `rawdata` directory.

- For each Markdown file, we create an instance of `TextLoader` with UTF-8 encoding and load the documents using the `load()` method.
- The loaded documents are then added to the `raw_data` list.

By the end of this cell, the `raw_data` list contains all the documents from the specified PDF and Markdown files, ready for further processing.

[ ]:
```
from langchain.document_loaders import PyPDFLoader, TextLoader
from langchain.schema import Document
import glob

#### Reading Dataset

pdf_files = glob.glob("rawdata/*.pdf")
md_files = glob.glob("rawdata/*.md")
raw_data=[]
for file in pdf_files:     # Load a PDF files
  pdf_loader = PyPDFLoader(file)
  pdf_documents = pdf_loader.load()
  raw_data.extend(pdf_documents)
for file in md_files:# Load a Markdown file
  md_loader = TextLoader(file, encoding="utf-8")
  md_documents = md_loader.load()
  raw_data.extend(md_documents)
```

[ ]:
```
for data in raw_data[0:5]:  #checking content of first 5 documents
  print(data.page_content)
```

1.1.2 Splitting Documents into Chunks

Here, we are splitting the loaded documents into smaller chunks to facilitate better processing and embedding generation. We use the `RecursiveCharacterTextSplitter` from the `langchain.text_splitter` module to achieve this.

Libraries Used:

- `langchain.text_splitter.RecursiveCharacterTextSplitter`: This class is used to split text documents into smaller chunks. It ensures that the chunks are of a manageable size and maintains context by allowing some overlap between chunks.

Process:

1. **Initialize Text Splitter**:
   - We create an instance of `RecursiveCharacterTextSplitter` with the following parameters:
     - `chunk_size=400`: This sets the maximum size of each chunk to 400 characters.
     - `chunk_overlap=60`: This sets the overlap between consecutive chunks to 60 characters to maintain context.
2. **Split Documents**:

    - We use the `split_documents()` method of the `text_splitter` instance to split the `raw_data` documents into smaller chunks.
    - The resulting chunks are stored in the `document_chunks` list.
3. **Print Sample Chunks**:
    - We print the content and metadata of the first 5 chunks to verify the splitting process.

By the end of this cell, the `document_chunks` list contains the smaller chunks of the original documents, ready for further processing and embedding generation.

[ ]:
```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=400,  # Maximum size of each chunk
    chunk_overlap=60  # Overlap between chunks to maintain context
)

# Split the documents into chunks
document_chunks = text_splitter.split_documents(raw_data)



for chunk in document_chunks[:5]:  # Print the first 5 chunks
    print("Chunk Content:", chunk.page_content)
    print("Metadata:", chunk.metadata)
```

1.1.3 Creating Vector Embeddings

In this cell, we are generating vector embeddings for the document chunks using a pre-trained model from the `sentence_transformers` library. These embeddings will be used for various downstream tasks such as similarity search, clustering, and more.

Libraries Used:

- `sentence_transformers.SentenceTransformer`: This class is used to load pre-trained models and generate embeddings for text data.

[ ]:
```
!pip install sentence-transformers
```

Process:

1. **Load Embedding Model**:
    - We load the `multi-qa-mpnet-base-dot-v1` model using the `SentenceTransformer` class. This model is specifically designed for generating high-quality embeddings for question-answering tasks and general-purpose semantic search.
    - **Reason for Choosing `multi-qa-mpnet-base-dot-v1`**:
        - **Performance**: This model is known for its high performance in generating embeddings that capture semantic meaning effectively. It has larger dimensions compared to other available models like `multi-qa-MiniLM-L6-cos-v1`, `all-MiniLM-L6-v2`
        - **Versatility**: It is optimized for question-answering, which aligns with the goal of creating a chatbot that answers AI-related questions.

2. **Generate Embeddings**:
   - We extract the text content from each chunk in `document_chunks` and store them in the `chunk_texts` list.
   - We use the `encode()` method of the `embedding_model` to generate embeddings for the `chunk_texts`.
   - The resulting embeddings are stored in the `embeddings` list.
3. **Print Sample Embedding**:
   - We print the embedding for the first chunk to verify the embedding generation process.

By the end of this cell, the `embeddings` list contains the vector embeddings for each document chunk, ready for use in further analysis and tasks.

[ ]:
```
######### Create vector Embeddings
from sentence_transformers import SentenceTransformer

# Load an embedding model
embedding_model = SentenceTransformer('multi-qa-mpnet-base-dot-v1')

# Generate embeddings for each chunk
chunk_texts = [chunk.page_content for chunk in document_chunks]
embeddings = embedding_model.encode(chunk_texts)

# Example: Print the embedding for the first chunk
print("First Chunk Embedding:", embeddings[0])
```

1.1.4 Normalizing and Converting Embeddings to NumPy Array

In this cell, we are converting the generated embeddings into a NumPy array and normalizing them. This step ensures that the embeddings are in a consistent format and have unit length, which is important for many machine learning algorithms and similarity calculations.

Libraries Used:

- `numpy`: This library is used for numerical operations in Python. It provides support for arrays and matrices, along with a collection of mathematical functions to operate on these data structures.

Process:

1. **Convert to NumPy Array**:
   - We convert the list of embeddings into a NumPy array using `np.array(embeddings)`. This allows us to leverage NumPy's efficient numerical operations.
   - We specify the data type as `float32` to ensure that the embeddings are stored in a format that is compatible with most machine learning frameworks.
2. **Normalize Embeddings**:
   - We normalize the embeddings to have unit length using `np.linalg.norm()`. This is done by dividing each embedding vector by its L2 norm.
   - Normalization ensures that the embeddings have a consistent scale, which is important for similarity calculations and other downstream tasks.

`embeddings_array` contains the normalized embeddings in a NumPy array format, which will be used in future.

```
[ ]: import numpy as np
     # Convert embeddings to a NumPy array
     embeddings_array = np.array(embeddings).astype('float32')
     embeddings_array /= np.linalg.norm(embeddings_array, axis=1, keepdims=True)

     print(embeddings_array.shape) #checking the vector store
```

following is sample function which returns embeddings for a user query

```
[ ]: def generate_vector(chunk):
         '''This function generates an vector for a given chunk/query
         Make sure you have imported sentence_transformers and defined the␣
      ↪embedding_model
         For this task we will use the multi-qa-mpnet-base-dot-v1 model'''
         embeddings = embedding_model.encode(chunk_texts)
         query_embedding = embedding_model.encode([query_text]).astype('float32')
         query_embedding = query_embedding / np.linalg.norm(query_embedding)     #␣
      ↪Normalize query embedding
         return query_embedding
```

1.1.5 Saving Chunks and Embeddings

We save the document chunks and their corresponding embeddings to files for later access.

Libraries Used:

- `pickle`: This module is used for serializing and deserializing Python objects.
- `numpy`: This library is used for numerical operations and saving arrays.

Process:

- We use `pickle.dump()` to save the `document_chunks` list to a binary file named `chunks.pkl`.
- We use `np.save()` to save the `embeddings_array` to a file named `vector_store.npy`.

the document chunks and embeddings are saved to disk, allowing for easy retrieval and reuse in future tasks.

```
[ ]: ######### Saving chunks for later access
     import pickle

     # Save chunks to a binary file
     with open("chunks.pkl", "wb") as f:
         pickle.dump(document_chunks, f)

     np.save("vector_store.npy", embeddings_array)
```

1.2 Generating Dataset for Model Training

we are generating a dataset to train the model using the pre-trained `Qwen2.5-3B model`. This approach is chosen due to resource(Unavailability of free APIs for better models) and time constraints.

you will be skipping initial steps as we already have our data loaded.

1.2.1 Overview

- **Objective**: Create a dataset suitable for training the chatbot model that answers AI-related questions.
- **Approach**: Utilize the `Qwen2.5-3B model` to generate the dataset while using the`multi-qa-mpnet-base-dot-v1` model to generate embeddings efficiently.

1.2.2 Steps:

1. **Load Data**: Loading the vector embeddings previously generated.
2. **Generate dataset**: leveraging `faiss` to get similar embeddings. feed them to model using predefined prompts for generating data.
3. **Save Dataset**: Store the generated dataset for use in model training.

Installing Dependancies

[ ]:
```
!pip install faiss-cpu
```

1.2.3 Finding Similar Embeddings with FAISS

we use FAISS to find similar embeddings for interactions.

Libraries Used:

- `faiss`: A library for efficient similarity search and clustering of dense vectors.

Process:

1. **Normalize Embeddings**:
    - Ensure embeddings are normalized for cosine similarity.
2. **Create FAISS Index**:
    - Create a FAISS index for inner product (cosine similarity) using the embeddings.
3. **Define Functions**:
    - `get_embeddings(query_text)`: Encodes a query text, normalizes it, and finds the top 3 similar document chunks.
    - `get_similar_indices(chunk_index, k)`: Finds the top `k` similar document chunks for a given chunk index, excluding itself.

[ ]:
```
import faiss

# Normalize embeddings for cosine similarity

# Create a FAISS index for cosine similarity (using Inner Product)
```

```
dimension = embeddings_array.shape[1]
index = faiss.IndexFlatIP(dimension)
index.add(embeddings_array)

def get_embeddings(query_text=""):
  query_embedding = embedding_model.encode([query_text]).astype('float32')
  # Normalize query embedding before searching
  query_embedding = query_embedding / np.linalg.norm(query_embedding)
  distances, indices = index.search(query_embedding, k=3)
  related_indices = [i for i in indices[0] if i!=-1]
  return [document_chunks[i].page_content for i in related_indices]


def get_similar_indices(chunk_index=-1,k=3):
    chunk_embedding = embeddings_array[chunk_index].reshape(1, -1)  # Ensure␣
 ↪correct shape

    # Perform FAISS search
    distances, indices = index.search(chunk_embedding, k=k+1)  # k+1 to exclude␣
 ↪itself

    # Remove the chunk itself from results (if present)
    related_indices = [i for i in indices[0] if (i != chunk_index and i!=-1)]

    # Return related chunks
    return [document_chunks[i].page_content for i in related_indices]
```

testing the retrieval of embeddings

[ ]:
```
# Query the vector database
query_text = "What are the first-generation reasoning models?"
retrieved_chunks= get_embeddings(query_text)
print(retrieved_chunks)
```

1.2.4 Prompt Templates for Dataset Generation

We define various prompt templates used to generate responses and queries for the dataset. We are doing query generation+query evolution along with Expected response generation for optimal dataset Creation

Functions:

1. **`Queryprompt(contexts)`**:
    - Generates a prompt to create JSON objects with questions or statements related to the given contexts.
2. **`multi_context_template(context, original_input)`**:
    - Rewrites an input to require information from all elements in the context, ensuring conciseness and relevance.

3. **`reasoning_template(context, original_input)`**:
    - Rewrites an input to explicitly request multi-step reasoning, ensuring it is concise and understandable.
4. **`hypothetical_scenario_template(context, original_input)`**:
    - Rewrites an input to incorporate a hypothetical scenario, encouraging application of knowledge from the context.
5. **`response_template(context, input)`**:
    - Generates a detailed and informative response to the given input, using relevant details from the context.

These templates help in structuring the prompts for generating diverse and relevant dataset entries.

[ ]:
```python
def Queryprompt(contexts):
   return f"""I want you act as a copywriter. Based on the given context,
which is list of strings, Generate a list of JSON objects
with a `input` key. The `input` can be a question or a statement
 related to the contexts .

contexts:
{contexts}"""



def multi_context_template(context,original_input):
  return f"""
  I want you to rewrite the given `input` so that it requires readers to use␣
 ↪information from all elements in `Context`.

1. `Input` should require information from all `Context` elements.
2. `Rewritten Input` must be concise and fully answerable from `Context`.
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` should not exceed 15 words.

Context: {context}
Input: {original_input}
Rewritten Input:
"""

def reasoning_template(context,original_input):
  return f"""
I want you to rewrite the given `input` so that it explicitly requests␣
 ↪multi-step reasoning.

1. `Rewritten Input` should require multiple logical connections or inferences.
2. `Rewritten Input` should be concise and understandable.
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` must be fully answerable from `Context`.
5. `Rewritten Input` should not exceed 15 words.
```

```
Context: {context}
Input: {original_input}
Rewritten Input:
"""

def hypothetical_scenario_template(context,original_input):
  return f"""
I want you to rewrite the given `input` to incorporate a hypothetical or␣
↪speculative scenario.

1. `Rewritten Input` should encourage applying knowledge from `Context` to␣
↪deduce outcomes.
2. `Rewritten Input` should be concise and understandable.
3. Do not use phrases like 'based on the provided context.'
4. `Rewritten Input` must be fully answerable from `Context`.
5. `Rewritten Input` should not exceed 15 words.

Context: {context}
Input: {original_input}
Rewritten Input:
"""

def response_template(context, input):
  return f"""
I want you to generate a well-structured response to the given `input`, using␣
↪relevant details from `Context`.
1. `Response` must be detailed, informative.
2. `Response` should provide clear and accurate information without unnecessary␣
↪repetition.
3. Do not use phrases like 'based on the provided context.'
4. Maintain a professional and engaging tone.

Context: {context}
Input: {input}
Response:
"""
```

The response of the model contains input and some other unnecessary information for dataset. So we process the responses with following functions in different cases.

```
[ ]: import re
     def extract_questions(response):
         chatbot_response = tokenizer.decode(response[0], skip_special_tokens=True)

         cleaned_response = re.sub(r"```json|```", "", chatbot_response).strip()

         # Try to find the JSON structure using regex
```

```
    questions = re.findall(r'"input":\s*"([^"]+)"', cleaned_response)

    return questions

def extract_response(response):
    text = tokenizer.decode(response[0], skip_special_tokens=True)
    start_index = text.find('Response:')

    if start_index != -1:
        # Slice the text from the 'Rewritten Input:' point onwards
        extracted_text = text[start_index+9:].strip()
        return extracted_text
    else:
        return None

def extract_rewritten_input(response):
    # Pattern to extract text between "Rewritten Input:" and "<|endoftext|>"
    text = tokenizer.decode(response[0], skip_special_tokens=True)
    start_index = text.find('Rewritten Input:')

    if start_index != -1:
        # Slice the text from the 'Rewritten Input:' point onwards
        extracted_text = text[start_index+16:].strip()
        return extracted_text
    else:
        return None
```

1.2.5 Evolving Queries and Generating Responses

In this cell, we define a function to perform random evolution steps on queries and generate responses, creating entries for the training dataset.

Libraries Used:

- `transformers.TextStreamer`: Used for streaming text generation.
- `re`: Regular expression operations.

Process:

1. **Templates and Evolution Steps**:
    - Define a list of evolution templates (`multi_context_template`, `reasoning_template`, `hypothetical_scenario_template`).
    - Set the number of evolution steps to apply.
2. **Function `evolve_query(original_input, context)`**:
    - Randomly choose a template from `evolution_templates`.
    - Generate an evolved prompt using the chosen template.
    - Tokenize and generate a response using the model.
    - Extract the rewritten input from the response.

- Generate a final response using the `response_template`.
- Append the original input, final response, and context to the `training_dataset`.

`training_dataset` contains evolved queries and their corresponding responses, ready for use in model training.

```
[ ]: !pip install torch transformers
     !pip install tqdm pandas
```

```
[ ]: from transformers import TextStreamer
     import re

     evolution_templates = [multi_context_template, reasoning_template,␣
      ↪hypothetical_scenario_template]

     # Number of evolution steps to apply
     num_evolution_steps = 3


     training_dataset=[]


     # Function to perform random evolution steps
     def evolve_query(original_input, context):
           # Choose a random (or using custom logic) template from the list
           chosen_template = random.choice(evolution_templates)
           # Replace the placeholders with the current context and input
           evolved_prompt = chosen_template(str(context), original_input)
           # Update the current input with the "Rewritten Input" section

           inputs = tokenizer(evolved_prompt, return_tensors = "pt").to("cuda")
           text_streamer = TextStreamer(tokenizer)
           response = model.generate(**inputs, streamer = text_streamer,␣
      ↪max_new_tokens = 250)
           evolved_question = extract_rewritten_input(response)

           prompt_for_response = response_template(str(context), evolved_question)

           inputs = tokenizer(prompt_for_response, return_tensors = "pt").to("cuda")
           text_streamer = TextStreamer(tokenizer)
           response = model.generate(**inputs, streamer = text_streamer,␣
      ↪max_new_tokens = 250)
           final_response = extract_response(response)

           training_dataset.append({"Input": original_input, "Response":␣
      ↪final_response, "Context": context})
           original_input = evolved_question
```

1.3 Generating training dataset

So far we were laying the groundwork for dataset generation. Now we will start the process by loading the required libraries and our base model #### Libraries Used: - `torch`: PyTorch library for tensor operations and deep learning. - `transformers`: Hugging Face library for transformer models. - `numpy`: Library for numerical operations. - `datasets`: Hugging Face library for loading and processing datasets. - `sklearn.model_selection`: Provides tools for splitting datasets into training and testing sets. - `unsloth.FastLanguageModel`: Custom library for efficient language model operations.

By importing these libraries, we set up the environment for model loading, data processing, and training.

[ ]:
```
%%capture
import os
if "COLAB_" not in "".join(os.environ.keys()):
    !pip install unsloth
else:
    # Do this only in Colab notebooks! Otherwise use pip install unsloth
    !pip install --no-deps bitsandbytes accelerate xformers==0.0.29 peft trl␣
 ↪triton
    !pip install --no-deps cut_cross_entropy unsloth_zoo
    !pip install sentencepiece protobuf datasets huggingface_hub hf_transfer
    !pip install --no-deps unsloth
```

[ ]:
```
!pip install -U bitsandbytes
```

[ ]:
```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
import numpy as np
from datasets import load_dataset
from sklearn.model_selection import train_test_split
from transformers import Trainer, TrainingArguments
from unsloth import FastLanguageModel
import torch
```

1.3.1 Loading and Configuring the Model

Here we load and configure the Qwen2.5-3B model using the `FastLanguageModel` class with specific settings to optimize performance and memory usage.

Parameters:

- **`max_seq_length`**: Set to 2048 to define the maximum sequence length.
- **`dtype`**: Automatically detected or set to `float16` for certain GPUs.
- **`load_in_4bit`**: Enabled to use 4-bit quantization, reducing memory usage.

Process:

1. **Model Selection**:

   - Choose from a list of available models, including various versions of Qwen2.5 and other models from the `unsloth` repository.
2. **Load Model**:
   - Use `FastLanguageModel.from_pretrained()` to load the selected model (`unsloth/Qwen2.5-3B`), with specified parameters for sequence length, data type, and quantization.
3. **Configure PEFT (Parameter-Efficient Fine-Tuning)**:
   - Apply PEFT to the model using `FastLanguageModel.get_peft_model()` with specified parameters for target modules, LoRA settings, and gradient checkpointing.
4. **Enable Faster Inference**:
   - Call `FastLanguageModel.for_inference()` to enable native 2x faster inference.

By the end of this cell, the Qwen2.5-3B model is loaded and configured for efficient training and inference.

```
[ ]: max_seq_length = 2048 # Choose any! We auto support RoPE Scaling internally!
     dtype = None # None for auto detection. Float16 for Tesla T4, V100, Bfloat16␣
      ↪for Ampere+
     load_in_4bit = True # Use 4bit quantization to reduce memory usage. Can be␣
      ↪False.

     fourbit_models = [
         "unsloth/Meta-Llama-3.1-8B-bnb-4bit",      # Llama-3.1 15 trillion tokens␣
      ↪model 2x faster!
         "unsloth/Meta-Llama-3.1-8B-Instruct-bnb-4bit",
         "unsloth/Meta-Llama-3.1-70B-bnb-4bit",
         "unsloth/Meta-Llama-3.1-405B-bnb-4bit",    # We also uploaded 4bit for 405b!
         "unsloth/Mistral-Nemo-Base-2407-bnb-4bit", # New Mistral 12b 2x faster!
         "unsloth/Mistral-Nemo-Instruct-2407-bnb-4bit",
         "unsloth/mistral-7b-v0.3-bnb-4bit",        # Mistral v3 2x faster!
         "unsloth/mistral-7b-instruct-v0.3-bnb-4bit",
         "unsloth/Phi-3.5-mini-instruct",           # Phi-3.5 2x faster!
         "unsloth/Phi-3-medium-4k-instruct",
         "unsloth/gemma-2-9b-bnb-4bit",
         "unsloth/gemma-2-27b-bnb-4bit",            # Gemma 2x faster!
     ] # More models at https://huggingface.co/unsloth


     model, tokenizer = FastLanguageModel.from_pretrained(
         # Can select any from the below:
         # "unsloth/Qwen2.5-0.5B", "unsloth/Qwen2.5-1.5B", "unsloth/Qwen2.5-3B"
         # "unsloth/Qwen2.5-14B",  "unsloth/Qwen2.5-32B",  "unsloth/Qwen2.5-72B",
         # And also all Instruct versions and Math. Coding verisons!
         model_name = "unsloth/Qwen2.5-3B",
         max_seq_length = max_seq_length,
         dtype = dtype,
         load_in_4bit = load_in_4bit,
         # token = "hf_...", # use one if using gated models like meta-llama/
       ↪Llama-2-7b-hf
     )

     model = FastLanguageModel.get_peft_model(
         model,
         r = 16, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
         target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                           "gate_proj", "up_proj", "down_proj",],
         lora_alpha = 16,
         lora_dropout = 0, # Supports any, but = 0 is optimized
         bias = "none",    # Supports any, but = "none" is optimized
         # [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
         use_gradient_checkpointing = "unsloth", # True or "unsloth" for very long␣
       ↪context
         random_state = 3407,
         use_rslora = False,  # We support rank stabilized LoRA
         loftq_config = None, # And LoftQ
     )
     FastLanguageModel.for_inference(model) # Enable native 2x faster inference
```

Again due time contraints and colab environment limitations we skipped evolution of queries with this function for faster dataset generation

[ ]:
```
def without_evolve(original_input, context):
      prompt_for_response = response_template(str(context), original_input)
      inputs = tokenizer(prompt_for_response, return_tensors = "pt").to("cuda")
      text_streamer = TextStreamer(tokenizer)
      response = model.generate(**inputs, streamer = text_streamer,␣
  ↪max_new_tokens = 250)
      final_response = extract_response(response)

      training_dataset.append({"Input": original_input, "Response":␣
  ↪final_response, "Context": context})
```

1.3.2 Generating Queries and Saving Dataset

In this cell, we generate queries based on document chunks, process them, and save the resulting dataset.

Process:

1. **Generate Queries**:
    - Loop 800 times to generate queries.
    - Randomly select a reference index from `document_chunks`.
    - Retrieve similar contexts using `get_similar_indices(reference_index)`.
    - Generate a prompt using `Queryprompt(contexts)`.
    - Tokenize the prompt and generate a response using the model.

    - Extract questions from the response.
    - If questions are found, randomly select one as `test_question`.
2. **Process Query**:
    - Call `without_evolve(test_question, contexts)` to process the query without evolution (or use `evolve_query` if needed).
3. **Save Dataset**:
    - Save the `training_dataset` to a JSON file named `dataset.json`.

the generated queries and their corresponding contexts are processed and saved in a JSON file for use in model training.

[ ]:
```python
import random

for i in range(800):
  reference_index = random.randint(0, len(document_chunks) - 1)
  contexts = get_similar_indices(reference_index)
  prompt = Queryprompt(contexts)
  inputs = tokenizer(prompt, return_tensors = "pt").to("cuda")

  text_streamer = TextStreamer(tokenizer)
  response = model.generate(**inputs, streamer = text_streamer, max_new_tokens␣
 ↪= 150)
  questions= extract_questions(response)
  if (len(questions)!=0):
    test_question = random.choice(questions)
  else :
    continue
  #evolve_query(test_question,contexts)
  without_evolve(test_question,contexts)

  with open('dataset.json', 'w') as json_file:
      json.dump(training_dataset, json_file, indent=4)
```

1.4 Training the Model

We will be skipping some steps. refer model_training.ipynb for more details

1.4.1 Overview

- **Objective**: Train the Qwen2.5-3B model to answer AI-related questions using the generated dataset.
- **Approach**: Load the dataset, configure the training parameters, and train the model using the `SFTTrainer` from the `trl` library.

1.4.2 Steps:

1. **Preprocess Data**: Tokenize the data and prepare it for training.
2. **Configure Training**: Set up the training arguments and configure the model for training.
3. **Train Model**: Use the `SFTTrainer` to train the model on the dataset.

By following these steps, we will fine-tune the Qwen2.5-3B model to improve its ability to answer AI-related questions.

1.4.3 Loading the Pre-trained Model

We have already loaded our model for dataset creation

1.4.4 Loading and Formatting the Dataset

Here we will split our dataset into training and testing sets, and format the prompts for training.

Process:

1. **Load Dataset**:
    - Load the dataset from the JSON file using `load_dataset("json", data_files="/content/dataset.json")`.
    - Split the dataset into training and testing sets with a 99:1 ratio using `train_test_split`.
2. **Define Prompt Template**:
    - Define the `alpaca_prompt` template to structure the instruction, input, and response for training.
    - The Alpaca format is used here to provide a clear and consistent structure for the model to learn from. It includes:
        - **Instruction**: Describes the task to be performed.
        - **Input**: Provides context or additional information needed to complete the task.
        - **Response**: The expected output or answer to the instruction based on the input.
3. **Format Prompts**:
    - Define the `formatting_prompts_func` function to format the dataset examples using the `alpaca_prompt` template.
    - Add the end-of-sequence token (`EOS_TOKEN`) to each formatted text to indicate the end of the response.
4. **Apply Formatting**:
    - Apply the `formatting_prompts_func` to the training dataset using the `map` function.

Reason for using Alpaca Format:

- **Consistency**: The Alpaca format provides a consistent structure for each training example, making it easier for the model to learn the relationship between instructions, inputs, and responses.
- **Clarity**: By clearly separating the instruction, input, and response, the model can better understand the context and generate appropriate answers.
- **Flexibility**: This format can accommodate a wide range of tasks and contexts, making it suitable for training a versatile chatbot model.

[ ]:
```python
from datasets import load_dataset, concatenate_datasets
# Load the dataset from the JSON file
dataset = load_dataset("json", data_files="/content/dataset.json")
dataset = dataset['train'].train_test_split(test_size=0.01, shuffle=True)

train_dataset = dataset['train']
```

```
test_dataset = dataset['test']

alpaca_prompt = """Below is an instruction that describes a task, paired with␣
↪an input that provides further context. Write a response that appropriately␣
↪completes the request.

### Instruction:
{}

### Input:
{}

### Response:
{}"""

EOS_TOKEN = tokenizer.eos_token

def formatting_prompts_func(examples):
    instructions = examples["Input"]
    inputs = examples["Context"]
    outputs = examples["Response"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return {"text": texts}

#dataset = load_dataset("json", data_files="/content/output.json")
#dataset = dataset.map(formatting_prompts_func, batched=True)
train_dataset = train_dataset.map(formatting_prompts_func, batched=True)

# Concatenate the datasets using concatenate_datasets
#dataset = concatenate_datasets([dataset['train'], train_dataset])
```

```
[ ]: print(dataset['train'][302])  # Access the 'train' split and then the element
```

1.4.5 Configuring and Initializing the Trainer

Now we configure and initialize the `SFTTrainer` from the `trl` library to train the model using the formatted dataset.

Parameters and Reasoning:

1. **Trainer Initialization**:
   - **`model`**: The pre-trained Qwen2.5-3B model loaded earlier.
   - **`tokenizer`**: The tokenizer associated with the model.
   - **`train_dataset`**: The training dataset prepared and formatted in previous steps.
   - **`dataset_text_field`**: `"text"` - The field in the dataset containing the formatted text.

    - **`max_seq_length`**: `2048` - Maximum sequence length for the model.
    - **`dataset_num_proc`**: `2` - Number of processes to use for data loading.
2. **Training Arguments**:
    - **`per_device_train_batch_size`**: `2` - Small batch size due to limited dataset size and to fit within memory constraints.
    - **`gradient_accumulation_steps`**: `4` - Accumulate gradients over 4 steps to effectively increase the batch size without requiring more memory.
    - **`warmup_steps`**: `5` - Number of warmup steps to gradually increase the learning rate at the start of training.
    - **`max_steps`**: `60` - Total number of training steps, kept low due to the small dataset size.
    - **`learning_rate`**: `2e-4` - Learning rate for the optimizer, chosen to balance between convergence speed and stability.
    - **`optim`**: `"adamw_8bit"` - Use the AdamW optimizer with 8-bit precision to reduce memory usage.
    - **`weight_decay`**: `0.01` - Weight decay for regularization to prevent overfitting.
    - **`lr_scheduler_type`**: `"linear"` - Linear learning rate scheduler for gradual learning rate decay.
    - **`seed`**: `3407` - Seed for reproducibility.
    - **`output_dir`**: `"outputs"` - Directory to save training outputs and checkpoints.
    - **`report_to`**: `"none"` - Disable reporting to external services (e.g., WandB).

the `SFTTrainer` is configured and initialized, ready to train the model on the provided dataset.

[ ]:
```python
from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported

trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_dataset,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    dataset_num_proc=2,
    packing=False,
    args=TrainingArguments(
        per_device_train_batch_size=2,
        gradient_accumulation_steps=4,
        warmup_steps=5,
        max_steps=60,
        learning_rate=2e-4,
        fp16=not is_bfloat16_supported(),
        bf16=is_bfloat16_supported(),
        logging_steps=1,
        optim="adamw_8bit",
        weight_decay=0.01,
        lr_scheduler_type="linear",
        seed=3407,
        output_dir="outputs",
        report_to="none",  # Use this for WandB etc
    ),
)
```

1.5 Training the model

```
[ ]: trainer_stats = trainer.train()
```

1.5.1 Testing the Trained Model

Here we test out newly trained model for sample instructions manually

By the end of this cell, the model generates a response to the given instruction using the specified prompt format, demonstrating its ability to provide concise and relevant answers.

```
[ ]: query_alpaca_prompt = """Below is an instruction that describes a task, paired␣
  ↪with an input that provides further context. Write a brief and precise␣
  ↪response that directly answers the question without unnecessary information.

1. Keep the response brief and concise.
2. Avoid elaborating on unrelated topics.
3. Provide only the most relevant information from the input.
### Instruction:
{}

### Input:
{}

### Response:
{}"""

FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
   query_alpaca_prompt.format(
        "is deepseek R1 Zero fast? ", # instruction
        " ", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 300)
```

Save Trained Model Using 4 bit Quantizising

```
[ ]: # Save to 4-bit Q4_K_M GGUF
     model.save_pretrained_gguf("model", tokenizer, quantization_method="q4_k_m")
```

1.5.2 Loading and Using Embeddings for Similarity Search (RAG Retrieval)

In this cell, we load embeddings and document chunks, and set up a FAISS index for similarity search.

Process:

1. **Load Embedding Model**:
    - Load the `multi-qa-mpnet-base-dot-v1` model from `SentenceTransformer` for generating embeddings.
2. **Load Document Chunks**:
    - Load the document chunks from a pickle file (`chunks.pkl`).
    - Convert the `LangChain Document` objects to a list of dictionaries containing text and metadata.
3. **Load Embeddings**:
    - Load the precomputed embeddings from a NumPy file (`vector_store.npy`).
    - Print the shape of the loaded embeddings to verify.
4. **Normalize Embeddings**:
    - Normalize the embeddings to ensure they are suitable for cosine similarity calculations.
5. **Create FAISS Index**:
    - Create a FAISS index for inner product (cosine similarity) using the loaded embeddings.
    - Add the embeddings to the FAISS index.
6. **Define Function for Similarity Search**:
    - `get_embeddings(query_text)`: Encodes a query text, normalizes it, and finds the top 3 similar document chunks using the FAISS index.
    - Returns the text content of the related document chunks.

```
[ ]: embedding_model = SentenceTransformer('multi-qa-mpnet-base-dot-v1')



     with open("chunks.pkl", "rb") as f:      #loading chunks
         document_chunks = pickle.load(f)




     # Convert LangChain Document objects to dict
     chunk_data = [{"text": doc.page_content, "metadata": doc.metadata} for doc in␣
       ↪document_chunks]


     embeddings_array = np.load("vector_store.npy")

     # Check loaded data
     #print(pick_chunks[0].page_content)  # Print first chunk text
```

```
print(embeddings_array.shape)


# Normalize embeddings for cosine similarity

# Create a FAISS index for cosine similarity (using Inner Product)
dimension = embeddings_array.shape[1]
index = faiss.IndexFlatIP(dimension)
index.add(embeddings_array)

def get_embeddings(query_text=""):
  query_embedding = embedding_model.encode([query_text]).astype('float32')
  # Normalize query embedding before searching
  query_embedding = query_embedding / np.linalg.norm(query_embedding)
  distances, indices = index.search(query_embedding, k=3)
  related_indices = [i for i in indices[0] if i!=-1]
  return [document_chunks[i].page_content for i in related_indices]
```

1.6 ~~Add LSTM for trained model~~

This part is not finished

[ ]:
```
import torch
import torch.nn as nn

class LSTMMemory(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers=1):
        super(LSTMMemory, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers, batch_first=True)
        self.hidden_dim = hidden_dim

    def forward(self, input_sequence, hidden_state=None):
        output, (hn, cn) = self.lstm(input_sequence, hidden_state)
        return output, (hn, cn)

    def init_hidden(self, batch_size):
        # Initialize hidden state and cell state for LSTM
        return (torch.zeros(1, batch_size, self.hidden_dim),
                torch.zeros(1, batch_size, self.hidden_dim))

# Initialize the LSTM Memory
input_dim = 768  # Embedding dimension (based on your model)
hidden_dim = 512
lstm_memory = LSTMMemory(input_dim, hidden_dim)
```

1.7 Response generate

Response generating prompt and procedure is defined here. Chat History integration in not integrated yet.

```
[ ]: chat_history = []



     # Generate Response Function
     def generate_response(query, chat_history, model, tokenizer, lstm_memory):
         # Retrieve relevant chunks
         relevant_chunks = get_embeddings(query)
         print(f"relevant chunks: \n{relevant_chunks}")
         print("\n\n\n")
         context = ' '.join(chat_history + relevant_chunks)
         context=relevant_chunks
         prompt= f"""Below is an instruction that describes a task, paired with an␣
      ↪input that provides further context. Write a brief and precise response that␣
      ↪directly answers the question without unnecessary information.

     1. Keep the response brief and concise.
     2. Avoid elaborating on unrelated topics.
     3. Provide only the most relevant information from the input.
     ### Instruction:
     {query}

     ### Input:
     {context}

     ### Response:"""
         # Prepare input for LSTM
       #  inputs = tokenizer(context, return_tensors='pt')
       #  embeddings = model(**inputs).last_hidden_state.mean(dim=1).unsqueeze(0)

         # Update LSTM memory
      #   lstm_output, hidden_state = lstm_memory(embeddings, hidden_state)

         # Generate response
     #    input_ids = tokenizer.encode(query + tokenizer.eos_token,␣
      ↪return_tensors="pt")
     #    response_ids = model.generate(input_ids, max_length=100)
     #    response = tokenizer.decode(response_ids[0], skip_special_tokens=True)
         inputs= tokenizer(
           [
             prompt# output - leave this blank for generation!
           ], return_tensors = "pt").to("cuda")
         text_streamer = TextStreamer(tokenizer)
```

```
    response = model.generate(**inputs, streamer = text_streamer,␣
↪max_new_tokens = 200)
    return response
```

1.8 Final Chat Bot

You can keep chatting with newly trained model here.

[ ]:
```
# Continuous Chat Loop
print("Start chatting with your bot (type 'exit' to stop):")
while True:
    user_input = input("You: ")
    if user_input.lower() == 'exit':
        break

    response = generate_response(
        user_input, chat_history, model, tokenizer, lstm_memory
    )
    response = tokenizer.decode(response[0],skip_special_tokens=True)

    # Update chat history for context and memory
    chat_history.append(user_input)
    chat_history.append(response)
```

1.9 Journey

Our journey in developing and training a custom AI model involved multiple challenges, adaptations, and optimizations. Initially, we attempted to train the model on our local machines but encountered significant **dependency issues**, making it impractical. As a result, we transitioned to **Google Colab**, which provided a cloud-based environment to run the training. However, this introduced new constraints, particularly **resource limitations** such as restricted GPU availability and execution timeouts.

To create a meaningful dataset, we leveraged **LangChain** to process documents by splitting them into manageable **text chunks**. We then generated **embeddings** using **sentence-transformers** and stored them in a **FAISS similarity index** for efficient retrieval. To generate **synthetic queries**, we selected random chunks and retrieved semantically similar ones using FAISS before passing them to a **pre-trained LLM** for query generation. Initially, we planned to further **augment these queries** using additional prompts to improve dataset diversity, but due to **time constraints**, we had to limit the augmentation process.

Once we had a sufficiently large dataset, we proceeded with training. We used the **same dataset** to fine-tune the pre-trained model for our specific task. After completing the fine-tuning process, we implemented a **Retrieval-Augmented Generation (RAG) architecture** to ensure that responses were **more relevant and concise** by dynamically retrieving relevant context during inference. However, due to **limited time and resources**, we were unable to fully integrate an **in-memory database**, which could have further optimized the retrieval process.

1.10 Conclusion

This project provided **valuable insights into the end-to-end process of fine-tuning LLMs**, from **data preparation** and **synthetic query generation** to **model training** and **RAG-based retrieval**. We encountered and overcame several challenges, including **dataset generation complexities, computational resource constraints in Colab, and limited dataset size** for effective training.

Despite these challenges, we successfully **trained and deployed a task-specific model** that improved response quality through **RAG**. Future improvements could include **enhanced query augmentation**, **a larger dataset**, and **integration of an in-memory vector database** to optimize retrieval speed and accuracy. The project reinforced the importance of **efficient dataset creation, resource-aware training, and real-time information retrieval** in building effective AI systems.

1.11 References

1. **Synthetic Data Generation using LLMs** - Confident AI Blog

2. **Qwen2.5 Fine-Tuning on Colab** - Unsloth Colab Notebook