# Local KnowledgeBase Q&A based on LangChain+Llama2

```
!pip install -qU transformers accelerate einops xformers bitsandbytes faiss-gpu langchain openai chromadb tiktoken
pypdf panel sentence_transformers #langchain[docarray]
```

```
!pip install -qU unstructured Cython unstructured[local-inference] huggingface_hub
```

```
import os
os.environ["HUGGINGFACEHUB_API_TOKEN"] = "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"
```

```
!huggingface-cli login --token $HUGGINGFACEHUB_API_TOKEN
```

```
from langchain.chains.question_answering import load_qa_chain
from langchain import HuggingFaceHub
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.llms import HuggingFacePipeline
from langchain.text_splitter import CharacterTextSplitter, RecursiveCharacterTextSplitter
# from langchain.vectorstores import DocArrayInMemorySearch
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA,  ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.document_loaders import PyPDFLoader, UnstructuredPDFLoader
from langchain.document_loaders import DirectoryLoader
```

```
from langchain.vectorstores import Chroma
from langchain.indexes import VectorstoreIndexCreator
```

```
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
```

```
!wget -q https://www.dropbox.com/s/92bswsw28imv75r/ResourceAIDoc.zip  #?dl=0  #
https://www.dropbox.com/s/zoj9rnm7oyeaivb/new_papers.zip
!unzip -q ResourceAIDoc.zip -d ResourceAIDoc
```

```
from torch import cuda, bfloat16
import transformers

model_id = 'meta-llama/Llama-2-7b-chat-hf'
```

```
device = f'cuda:{cuda.current_device()}' if cuda.is_available() else 'cpu'
```

```
# set quantization configuration to load large model with less GPU memory
# this requires the `bitsandbytes` library
bnb_config = transformers.BitsAndBytesConfig(
    load_in_4bit=True,
```

QLoRA is an efficient fine-tuning method that reduces memory usage enough to fine-tune a 65B parametric model on a single 48GB GPU while retaining full 16-bit fine-tuning task performance.QLoRA backpropagates gradients to low-order adapters~ (LoRA) through a frozen 4-bit quantized pre-trained language model.

```
    bnb_4bit_quant_type='nf4',
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=bfloat16
)

# begin initializing HF items, you need an access token
hf_auth =os.environ["HUGGINGFACEHUB_API_TOKEN"]
model_config = transformers.AutoConfig.from_pretrained(
    model_id,
    use_auth_token=hf_auth
)
```

When using the transformer family of models provided by Huggingface, a pre-trained model is loaded via the model.from_pretrained function

```
model = transformers.AutoModelForCausalLM.from_pretrained(
    model_id,
    trust_remote_code=True,
    config=model_config,
    quantization_config=bnb_config,
    device_map='auto',
    use_auth_token=os.environ["HUGGINGFACEHUB_API_TOKEN"]
)

# enable evaluation mode to allow model inference
model.eval()

print(f"Model loaded on {device}")
```

A tokenizer is a tool used to preprocess text. The tokenizer converts human-readable plaintext into LLM-readable token IDs .

First, the tokenizer splits the input document, dividing a sentence into individual words (or parts of words, or punctuation marks).

These individual words after segmentation are called tokens.

In the second step, the tokens are converted by the tagger into numbers, which can then be fed into the model.

In order to implement this ability to convert tokens to numbers, the tagger has a vocabulary that is downloaded when we instantiate and specify the model, and the vocabulary used by the tagger is the same as the vocabulary used by the model during pre-training.

Initializing the Splitter

```
tokenizer = transformers.AutoTokenizer.from_pretrained(
    model_id,
    use_auth_token=hf_auth
)
```

AutoTokenizer is actually designed for some informally included big models such as chatglm, llama, etc.; it will go to path and read the token configuration file of each model, get the word list and the corresponding Tokennizer class of that model; eventually it will return the initialized Tokennizer class. (Actually it is initializing and loading custom Tokennizer classes in path, such as ChatGLMTokenizer)

```
import torch

stop_list = ['\nHuman:', '\n```\n']

stop_token_ids = [tokenizer(x)['input_ids'] for x in stop_list]
stop_token_ids = [torch.LongTensor(x).to(device) for x in stop_token_ids]
stop_token_ids

from transformers import StoppingCriteria, StoppingCriteriaList

# define custom stopping criteria object
```

The stop condition allows us to specify when the model should stop generating text. If we do not provide a stop condition, the model will go off-topic after answering the initial question.

must convert these stop token IDs to LongTensor objects.

```
class StopOnTokens(StoppingCriteria):
    def __call__(self, input_ids: torch.LongTensor, scores: torch.FloatTensor, **kwargs) -> bool:
        for stop_ids in stop_token_ids:
            if torch.eq(input_ids[0][-len(stop_ids):], stop_ids).all():
                return True
        return False

stopping_criteria = StoppingCriteriaList([StopOnTokens()])

transformers_pipeline = transformers.pipeline(
    model=model,
    tokenizer=tokenizer,
    return_full_text=True,  # langchain expects the full text
    task='text-generation',
    # we pass model parameters here too
    stopping_criteria=stopping_criteria,  # without this model rambles during chat
    temperature=0.2,  # 'randomness' of outputs, 0.0 is the min and 1.0 the max
    max_new_tokens=225,  # max number of tokens to generate in the output, 512, or default 2048
    repetition_penalty=1.1  # without this output begins repeating
)

llm_llama_2_chat = HuggingFacePipeline(pipeline=transformers_pipeline)
from torch import cuda


def load_db(file, chain_type, k):
    # load documents
    # LangChain can download various files inc. youtube, Notion, json, pdf, ...
    loader = DirectoryLoader(file, glob="./*.pdf", loader_cls=PyPDFLoader) # WHOobesity
    documents = loader.load()
    # split documents
    # here we use characters (textsplitter), alternative: TokenTextSplitter
    # if markdown document: use MarkdownHeaderTextSplitter
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000, chunk_overlap=20, separators=["\n\n", "\n", " ", ""])
    docs = text_splitter.split_documents(documents)
    # define embedding
    embed_model_id = 'sentence-transformers/all-mpnet-base-v2' # 'sentence-transformers/all-MiniLM-L6-v2'
    model_kwargs = {"device": "cuda"}
    # encode_kwargs= {'device': device, 'batch_size': 32}
    vectorstore_option = "FAISS"

    device = f'cuda:{cuda.current_device()}' if cuda.is_available() else 'cpu'

    embedding = HuggingFaceEmbeddings(
```

Pipeline is a high-level API of the Transformers library that makes it easy to combine multiple processing steps (e.g., disambiguation, entity recognition, text categorization, etc.) into a single pipeline, thus enabling multiple natural language processing tasks to be accomplished with a single instruction. Inside the Pipeline, the input text is first preprocessed. This typically involves converting the text into a sequence of tokens and performing operations such as stop word processing, punctuation removal, and so on. Next, Pipeline feeds the sequences into a model, which is usually composed of a Transformer architecture.

For each task, Pipeline selects an appropriate model with appropriate parameters. For example, for a text categorization task, Pipeline might select an appropriate classifier and optimize this classifier with appropriate parameters. This process usually consists of pre-defined pipeline components.

Finally, Pipeline converts the output of the model into a task-specific output format. For example, for text categorization, the output might be a string indicating which category the text belongs to. For named entity recognition, the output might be a list containing entity names and entity types.

You have to make sure to split the text into small pieces. You will need to initialize RecursiveCharacterTextSplitter and call it by passing the documents.

You have to create embeddings for each small chunk of text and store them in the vector store (i.e. FAISS). You will be using all-mpnet-base-v2 Sentence Transformer to convert all pieces of text in vectors while storing them in the vector store.

```
        model_name=embed_model_id,
        model_kwargs=model_kwargs,
        # encode_kwargs=encode_kwargs
    )
```

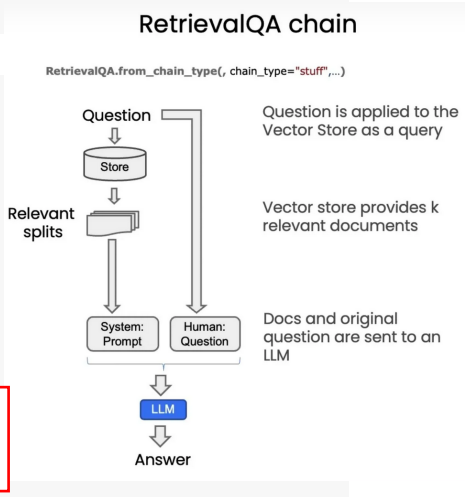
```
    # storing embeddings in the vector store
    # create vector database from data
    persist_directory = '/content/docs/vectorstore/' # Chroma FAISS
    if vectorstore_option == "Chroma":
        vectordb = FAISS.from_documents(documents=docs, embedding=embedding)
    else:
        vectordb = Chroma.from_documents(documents=docs,
                                         embedding=embedding,
                                         persist_directory=persist_directory)
```

Create conversation retrieval chains for integrating chat history and newly raised questions into a new standalone question

```
    # vectordb = DocArrayInMemorySearch.from_documents(docs, embedding)
    # define retriever
    retriever = vectordb.as_retriever(search_type="similarity", search_kwargs={"k": k})
    # create a chatbot chain. Memory is managed externally.
    qa = ConversationalRetrievalChain.from_llm(
        llm=llm_llama_2_chat,
        chain_type=chain_type,
        retriever=retriever,
        return_source_documents=True,
        return_generated_question=True,
    )
    return qa
```

You have to initialize ConversationalRetrievalChain. This chain allows you to have a chatbot with memory while relying on a vector store to find relevant information from your document.

Additionally, you can return the source documents used to answer the question by specifying an optional parameter i.e. return _source_documents=True when constructing the chain.




```
import panel as pn
import param
```

Using the panel component to create a GUI for chat conversations

```
class cbfs(param.Parameterized):
    chat_history = param.List([])
    answer = param.String("")
    db_query  = param.String("")
    db_response = param.List([])
```

Provide a user interface for interacting with chatbots

```
    def __init__(self,  **params):
        super(cbfs, self).__init__( **params)
        self.panels = []
        self.loaded_files = "/content/ResourceAIDoc/ResourceAIDoc/"
        self.qa = load_db(self.loaded_files,"stuff", 3)
```

```
    def call_load_db(self, count):
```

```
        if count == 0 or file_input.value is None:  # init or no file specified :
            return pn.pane.Markdown(f"Loaded File: {self.loaded_files}")
        else:
            file_input.save("temp.pdf")  # local copy
            self.loaded_files = file_input.filename
            button_load.button_style="outline"
            self.qa = load_db("temp.pdf", "stuff", 3)
            button_load.button_style="solid"
        self.clr_history()
        return pn.pane.Markdown(f"Loaded Files: {self.loaded_files}")
```

```
    def convchain(self, query):
        if not query:
            return pn.WidgetBox(pn.Row('User:', pn.pane.Markdown("", width=600)), scroll=True)
        result = self.qa({"question": query, "chat_history": self.chat_history})
        self.chat_history.extend([(query, result["answer"])])
        self.db_query = result["generated_question"]
        self.db_response = result["source_documents"]
        self.answer = result['answer']
        self.panels.extend([
            pn.Row('User:', pn.pane.Markdown(query, width=600)),
            pn.Row('ChatBot:', pn.pane.Markdown(self.answer, width=600, style={'background-color':
'#F6F6F6'}))def convchain(self, query):
        ])
        inp.value = ''  #clears loading indicator when cleared
        return pn.WidgetBox(*self.panels,scroll=True)
```

```
    @param.depends('db_query ', )
    def get_lquest(self):
        if not self.db_query :
            return pn.Column(
                pn.Row(pn.pane.Markdown(f"Last question to DB:", styles={'background-color': '#F6F6F6'})),
                pn.Row(pn.pane.Str("no DB accesses so far"))
            )
        return pn.Column(
            pn.Row(pn.pane.Markdown(f"DB query:", styles={'background-color': '#F6F6F6'})),
            pn.pane.Str(self.db_query )
        )
```

```
    @param.depends('db_response', )
    def get_sources(self):
        if not self.db_response:
            return
        rlist=[pn.Row(pn.pane.Markdown(f"Result of DB lookup:", styles={'background-color': '#F6F6F6'}))]
```

```
        for doc in self.db_response:
            rlist.append(pn.Row(pn.pane.Str(doc)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)
```

```
    @param.depends('convchain', 'clr_history')
    def get_chats(self):
        if not self.chat_history:
            return pn.WidgetBox(pn.Row(pn.pane.Str("No History Yet")), width=600, scroll=True)
        rlist=[pn.Row(pn.pane.Markdown(f"Current Chat History variable", styles={'background-color':
'#F6F6F6'}))]
        for exchange in self.chat_history:
            rlist.append(pn.Row(pn.pane.Str(exchange)))
        return pn.WidgetBox(*rlist, width=600, scroll=True)
```

```
    def clr_history(self,count=0):
        self.chat_history = []
        return
```

```
cb = cbfs()

file_input = pn.widgets.FileInput(accept='.pdf')
button_load = pn.widgets.Button(name="Load DB", button_type='primary')
button_clearhistory = pn.widgets.Button(name="Clear History", button_type='warning')
button_clearhistory.on_click(cb.clr_history)
inp = pn.widgets.TextInput( placeholder='Enter text here…')
```

```
bound_button_load = pn.bind(cb.call_load_db, button_load.param.clicks)
conversation = pn.bind(cb.convchain, inp)
```

```
# jpg_pane = pn.pane.Image('/content/convchain.jpg')
```

```
tab1 = pn.Column(
    pn.Row(inp),
    pn.layout.Divider(),
    pn.panel(conversation,  loading_indicator=True, height=400),
    pn.layout.Divider(),
)
tab2= pn.Column(
    pn.panel(cb.get_lquest),
    pn.layout.Divider(),
    pn.panel(cb.get_sources ),
)
tab3= pn.Column(
```

```
    pn.panel(cb.get_chats),
    pn.layout.Divider(),
)
tab4=pn.Column(
    pn.Row( file_input, button_load, bound_button_load),
    pn.Row( button_clearhistory, pn.pane.Markdown("Clears chat history. Can use to start a new topic" )),
    pn.layout.Divider(),
    # pn.Row(jpg_pane.clone(width=400))
)
dashboard = pn.Column(
    pn.Row(pn.pane.Markdown('# AIDoc_Bot v4')),
    pn.Tabs(('Conversation', tab1), ('Database', tab2), ('Chat History', tab3),('Configure', tab4))
)
```

```
pn.extension(comms='colab')
```

PS： funuction of langchain
When we enter a prompt into our new chatbot, LangChain looks up the relevant information in the vector store. You can think of it as a little Google dedicated to your documents. Once we find the relevant information, we use it to feed the LLM with the hint to generate our answer.