Steps to generate text with your own data

This text was adapted from Data Science Academy, whith Artificial Intelligence Engineer training, thrue the Natural Language Processing with Transformers course and adapting the Project 7: Applying LLM for Text Analytics to Your Own Data.

First install Anaconda and install Python Packages.

Use Streamlit that is a faster way to build and share data apps. It turns data scripts into shareable web apps in minutes. All in pure Python. No front-end experience required. To run the program open the terminal, go to the folder and type: 'streamlit run app.py'.

Start creating columns for page layout and sets the aspect ratio of the columns, Configure the first column to display the project title. Input OpenAI API key field. To create your API on OpenAI consult https://platform.openai.com/, https://platform.openai.com/api-keys and https://platform.openai.com/docs/quickstart?context=python .

To check API Key use this commands:

```
'if not openai_api_key:
    st.info("Add your OpenAI API key in the left column to continue.")
    st.stop()

if openai_api_key:
    st.info("Wait for processing.")'
```

To defining the OpenAI API:

```
'llm_api = OpenAI(openai_api_key=openai_api_key)'
```

Download the Hugging Face Sentence Transformers Template, wicth maps sentences and paragraphs to a dense 384-dimensional vector space and can be used for tasks such as clustering or semantic search: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Creates a function to load the embeddings model passing model_path and normalize_embedding=Tru as arguments. This function returns an instance of the HuggingFaceEmbeddings class. The 'model_name' is the identifier of the embeddings model to be loaded, the 'model_kwargs' is a dictionary of additional arguments for the template configuration, in this case setting the device to 'cpu' and the 'encode_kwargs' is a dictionary of arguments for the encoding method, here specifying whether embeddings should be normalized.

Load the Embedding model named all-MiniLM-L6-v2.

Creates a unction to load the pdf, creates an instance of the PyMuPDFLoader class, passing the PDF file path as an argument. Uses the 'load' method of the 'loader' object to load the PDF content, this returns an object or data structure containing the PDF pages with their content. The function returns the loaded content of the PDF.

Upload the pdf file with your own data.

Creates a function to divide documents into several chunks, creates an instance of the RecursiveCharacterTextSplitter class. This class divides long texts into smaller chunks. The 'chunk_size' defines the size of each chunk, and 'chunk_overlap' defines the overlap between consecutive chunks. Uses the 'split_documents' method of the 'text_splitter' object to split the given document, the 'documents' is a variable that contains the text or set of texts to be divided. Returns the chunks of text resulting from the split. Creates a variable documens to split the file into chunks.

Uses the FAISS (Facebook AI Similarity Search), FAISS is a library that allows developers to quickly search for embeddings of multimedia documents that are similar to each other. It solves limitations of traditional query search engines that are optimized for hash-based searches, and provides more scalable similarity search functions.
The link of FAISS is: https://ai.meta.com/tools/faiss/.

```
# Load the vectorstore with the FAISS, if it doesn't exist, create the
vectorstore
file_path = "./model/vectorstore/index.faiss"
storing_path = "model/vectorstore"


# Function to create embeddings using FAISS
def create_embeddings(chunks, embedding_model, storing_path =
"model/vectorstore"):

    # Creates a 'vectorstore' (a FAISS index) from the given documents.
    # 'chunks' is the list of text segments and 'embedding_model' is the
embedding model used to convert text to embeddings.
    vectorstore = FAISS.from_documents(chunks, embedding_model)

    # Saves the created 'vectorstore' to a local path specified by
'storing_path'.
    # This allows persistence of the FAISS index for future use.
    vectorstore.save_local(storing_path)

    # Returns the created 'vectorstore', which contains the embeddings
and can be used for similarity search and comparison operations.
    return vectorstore



if os.path.exists(file_path):
    vectorstore = FAISS.load_local(storing_path, embed,
allow_dangerous_deserialization=True)
else:
    vectorstore = create_embeddings(documents, embed)


# Convert vectorstore to a retriever
retriever = vectorstore.as_retriever()
```

```
template = """
### System:
You are an experienced technology analyst. You have to answer user
questions\
using only the context provided to you. If you don't know the answer, \
just say you don't know. Don't try to invent an answer.

### Context:
{context}

### User:
{question}

### Response:
"""


# Creating the prompt from the template
prompt = PromptTemplate.from_template(template)


# Creating the chain
def load_qa_chain(retriever, llm, prompt):

    # Retorna uma instância da classe RetrievalQA.
    # Returns an instance of the RetrievalQA class.
    # 'llm' refers to the large-scale language model (such as a GPT or
BERT model).
    # 'retriever' is a component used to retrieve relevant information
(like a search engine or document retriever).
    # 'chain_type' defines the type of chain or strategy used in the QA
process. Here, it is set to "stuff",
    # a placeholder for a real type.
    # 'return_source_documents': a boolean that, when True, indicates
that the source documents
    # (i.e. the documents from which the answers are extracted) must be
returned along with the answers.
    # 'chain_type_kwargs' is a dictionary of additional arguments
specific to the chosen chain type.
    # Here, it is passing 'prompt' as an argument.
    return RetrievalQA.from_chain_type(llm = llm,
                                       retriever = retriever,
                                       chain_type = "stuff",
                                       return_source_documents = True,
                                       chain_type_kwargs = {'prompt':
prompt})


# Creating the chain (pipeline)
qa_chain = load_qa_chain(retriever, llm_api, prompt)


st.info("")

# In the second column, receive the user's text
#with col2:
    #input_text = st.text_input("Enter your question:")
```

```
input_text = st.text_input("Enter your question:")

# Function to obtain LLM (Large Language Model) answers
def get_response(query, chain):

    # Invokes the 'chain' (processing chain, a Question Answering
pipeline) with the provided 'query'.
    # 'chain' is a function that takes a query and returns a response,
using LLM.
    response = chain({'query': query})

    # Uses the textwrap library to format the response. 'textwrap.fill'
wraps the text of the
    # response in lines of specified width (100 characters in this case),
    # making it easier to read in environments like Jupyter Notebook.
    wrapped_text = textwrap.fill(response['result'], width=100)

    # Imprime o texto formatado
    # print(wrapped_text)

    return wrapped_text


# Displays the generated text

if input_text:
#    st.write("Você digitou:", input_text)
    st.write("Generated text:", get_response(input_text, qa_chain))

# End
```