ChatBot using Ollama : Overview

Table of Contents

Overview

1. This project uses `Ollama` to download the `llama` model and use it to create a chatbot. The chatbot is trained on the `llama` model and can be used to chat with the user.
2. `HuggingFace` free available models were used to create the embeddings of the input `pdf documents`.
3. The embeddings were then stored in the `Chroma` database which is a `Vector` database. Using this it becomes easier to search for the nearest embeddings and get the response.
4. Then the `llama` model is trained on the data and the chatbot is created.

Taking PDF Input

```
with st.sidebar:
        st.subheader("Your documents")
        pdf_docs = st.file_uploader("Upload your PDFs here and click on
'Process'", accept_multiple_files=True)
        if st.button("Process"):
            with st.spinner("Processing ..."):
                raw_text = get_pdf_text(pdf_docs)
                tables = get_pdf_tables(pdf_docs)
                text_chunks = get_text_chunks(raw_text)

                st.session_state.pages_data = get_images_from_pdf(pdf_docs)
```

```
                vectorstore = get_vectorstore(text_chunks, tables)
                st.session_state.conversation =
get_conversation_chain(vectorstore)

        st.subheader("Image Display Options")
        show_highest_similarity = st.checkbox("Show images from the page with
highest similarity", value=True)
        if not show_highest_similarity:
            similarity_threshold = st.slider("Images would be displayed from
the pages with more than below similarity", min_value=0.0, max_value=1.0,
value=0.5, step=0.05)
        else:
            similarity_threshold = None

        st.subheader("Number of Images to Display")
        num_images = st.slider("Set number of images to display per page of
the PDF", min_value=1, max_value=10, value=5, step=1)
```

1. The user uploads the pdf document.
2. The pdf document is then converted to text using the `pdfplumber` library.
3. The text is then passed to the `HuggingFace` model to get the embeddings.
4. The embeddings are then stored in the `Chroma` database.
5. The different function are used to perform the above tasks and then finally create the conversation chain based on it and train the `llama` model on it.
    1. The vectorstore is passed to the `llama` model and the chain is created.
    2. The chain is created to store the memory of the past chats and result to be displayed to the user. Also an initial prompt is added to the memory to be displayed inorder to tune the response of the model.

```
def get_conversation_chain(vectorstore):
llm = Ollama(model="llama3")
memory = ConversationBufferMemory(memory_key='chat_history',
return_messages=True)

initial_instruction = (
    "You are an AI Assistant that follows instructions extremely well.
Please be truthful and give direct answers. If you do not know the answer
or if the user query is not in context, please say 'I don't know'. "
    "You will lose the job if you answer out-of-context questions. If a
query is confusing or unclear, ask a probing question to clarify. Remember
to only return the AI's answer."
```

```
)
initial_response = "OK."

memory.save_context({"user": initial_instruction},{"response":
initial_response})

conversation_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    memory=memory
)
return conversation_chain
```

6. 1. `RecursiveCharacterTextSplitter` is used to create `chunk` out of the texts of the pdf.

```
def get_text_chunks(text):
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=128,
    length_function=len,
    separators=[" ", ",", "\n", "."]
)
chunks = text_splitter.split_text(text)
return chunks
```

2. An attempt is made to extract `table data` specifically using inbuilt function from the library `pdfplumber`. This resulted is some question getting answered correctly from the table.

```
def get_pdf_tables(pdf_docs):
tables = []
for pdf in pdf_docs:
    with pdfplumber.open(pdf) as pdf_file:
        for page in pdf_file.pages:
            tables.append(page.extract_tables())
return tables
```

3. In below function, text data and table data is merged together for further processing.

```
def get_vectorstore(text_chunks, tables):
# Convert tables into a list of strings
```

```
    table_texts = []
    for table in tables:
        for row in table:
            # Flatten the row if it's a list of lists
            if all(isinstance(cell, list) for cell in row):
                row = [item for sublist in row for item in sublist]
            # Filter out None values
            row = [item for item in row if item is not None]
            table_texts.append(' '.join(row))

    # Combine text_chunks and table_texts
    all_texts = text_chunks + table_texts

    if(torch.backends.mps.is_available()):
        device = 'mps'
    elif(torch.cuda.is_available()):
        device = 'cuda'
    else:
        device = 'cpu'

    embeddings = HuggingFaceInstructEmbeddings(model_name="sentence-
    transformers/all-MiniLM-L6-v2", model_kwargs={'device': device},
    encode_kwargs={'device': device})

    vectorstore = Chroma.from_texts(texts=all_texts, embedding=embeddings)

    return vectorstore
```

Handling User Input

```
def handle_userinput(user_question, show_highest_similarity,
similarity_threshold,num_images):
    response = st.session_state.conversation({'question': user_question})
    st.session_state.chat_history = response['chat_history']

    llm_response = response['answer']

    st.write(user_template.replace("{{MSG}}", user_question),
unsafe_allow_html=True)
    st.write(bot_template.replace("{{MSG}}", llm_response),
unsafe_allow_html=True)

    # Check similarity between the response and text in pages_data
    embeddings = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
```

```
    response_embedding = embeddings.encode(llm_response,
convert_to_tensor=True)

    similarities = []

    for page_data in st.session_state.pages_data:
        page_text_embedding = embeddings.encode(page_data['text'],
convert_to_tensor=True)
        similarity = util.pytorch_cos_sim(response_embedding,
page_text_embedding).item()
        similarities.append((similarity, page_data))

    # Sort pages by similarity in descending order
    similarities.sort(key=lambda x: x[0], reverse=True)

    is_image_displayed = False
    # Display images based on user selection
    if show_highest_similarity:
        for similarity, page_data in similarities:
            if page_data['images']:
                st.write(f"Related images from PDF '{page_data['pdf_name']}',
page {page_data['page_number']} with similarity {similarity:.2f}:")
                is_image_displayed = True
                images_threshold = min(num_images, len(page_data['images']))
                for image_path in page_data['images']:
                    st.image(image_path)

                    images_threshold -= 1

                    if images_threshold == 0:
                        break
                break
    else:
        for similarity, page_data in similarities:
            if similarity ≥ similarity_threshold and page_data['images']:
                st.write(f"Related images from PDF '{page_data['pdf_name']}',
page {page_data['page_number']} with similarity {similarity:.2f}:")
                is_image_displayed = True
                images_threshold = min(num_images, len(page_data['images']))
                for image_path in page_data['images']:
                    st.image(image_path)

                    images_threshold -= 1

                    if images_threshold == 0:
                        break
```

```
    if not is_image_displayed:
        st.write(f"No images with the given similarity threshold
{similarity_threshold:.2f} found.")
```

1. This function is called when user askes a question.
2. The question is passed to the model and the model returns the response stored in the `response` variable.
3. The response is then stored in the `chat_history` to be displayed to the user.
4. The for loop is to display the response and question to the user in respective `html templates`.
5. To display the images to the user, the similarity between the response and the text in the pdf document is calculated.
6. The images are then displayed to the user based on the similarity of the response and the text in the pdf document. The images corresponding to that part of the text is shown to the user.

Images as Output

```
def get_images_from_pdf(pdf_docs):
    pages_data = []
    for pdf in pdf_docs:
        pdf_path = os.path.join(tempfile.gettempdir(), pdf.name)
        with open(pdf_path, 'wb') as f:
            f.write(pdf.getbuffer())
        pdf_document = fitz.open(pdf_path)
        for page_number in range(pdf_document.page_count):
            page = pdf_document[page_number]
            text = page.get_text()
            images = []
            for image_index, img in enumerate(page.get_images(full=True)):
                xref = img[0]
                base_image = pdf_document.extract_image(xref)
                try:
                    image = Image.open(io.BytesIO(base_image["image"]))

                    # Rotate image if necessary
                    image = ImageOps.exif_transpose(image)
                    image_path = os.path.join(tempfile.gettempdir(), f"
{pdf.name}_image_{page_number}_{image_index}.jpg")
                    image.save(image_path)
                    images.append(image_path)
```

```
                except (UnidentifiedImageError, KeyError) as e:
                    print(f"Error processing image on page {page_number},
index {image_index}: {e}")
            pages_data.append({
                "pdf_name": pdf.name,
                "page_number": page_number,
                "text": text,
                "images": images
            })
    return pages_data
```

1. The function is used to extract the images from the pdf document.
2. The images are then stored in the `temp` directory and the path is stored in the `pages_data` variable.
3. The `pages_data` variable is then used to display the images to the user based on the similarity of the response and the text in the pdf document.
4. The images are displayed to the user based on the similarity of the response and the text in the pdf document. The images corresponding to that part of the text is shown to the user.

Sidebar Options

1. The user can choose to display the images based on the similarity of the response and the text in the pdf document.
2. The user can also choose the number of images to be displayed to the user.
3. The user can also choose the similarity threshold to display the images to the user.