Construa um Chatbot para Seus Dados

Tempo estimado necessário: 60 min

Introdução

Neste projeto, você criará um chatbot para seu próprio arquivo pdf usando Flask, um popular framework web, e LangChain, outro framework popular para trabalhar com grandes modelos de linguagem (LLMs). O chatbot que você desenvolverá não apenas interagirá com os usuários por meio de texto, mas também compreenderá e responderá perguntas relacionadas ao conteúdo de um documento específico.

Clique no link da demonstração abaixo para experimentar a aplicação final que você criará!

Experimente a aplicação de demonstração

Ao final deste projeto, você terá uma compreensão mais profunda sobre chatbots, desenvolvimento de aplicações web usando Flask e Python, e o uso do framework LangChain na interpretação e resposta a uma ampla gama de entradas dos usuários. E o mais importante, você terá construído uma aplicação de chatbot abrangente e impressionante!

Uma pessoa procura um documento em uma pilha enorme de papéis.

Chatbots

Chatbots são aplicações de software projetadas para engajar em conversas semelhantes às humanas. Eles podem responder a entradas de texto dos usuários e são amplamente utilizados em diversos domínios, incluindo atendimento ao cliente, eCommerce e educação. Neste projeto, você construirá um chatbot capaz de não apenas envolver os usuários em uma conversa geral, mas também responder a consultas com base em um documento específico.

LangChain

LangChain é uma ferramenta versátil para construir aplicações de linguagem impulsionadas por IA. Ela fornece várias funcionalidades, como recuperação de texto, sumarização, tradução e muito mais, aproveitando modelos de linguagem pré-treinados. Neste projeto, você integrará o LangChain ao seu chatbot, capacitando-o a entender e responder a diversas entradas dos usuários de forma eficaz.

Flask

Flask é um framework web leve e flexível para Python, conhecido por sua simplicidade e velocidade. Um framework web é um framework de software projetado para suportar o desenvolvimento de aplicações web, incluindo a criação de servidores web e gerenciamento de requisições e respostas HTTP.

Você usará o Flask para criar o lado do servidor ou backend do seu chatbot. Isso envolve lidar com mensagens recebidas dos usuários, processar essas mensagens e enviar respostas apropriadas de volta ao usuário.

Rotas no Flask

Rotas são uma parte essencial do desenvolvimento web. Quando sua aplicação recebe uma solicitação de um cliente (tipicamente um navegador web), ela precisa saber como lidar com essa solicitação. É aqui que o roteamento entra.

No Flask, as rotas são criadas usando o decorador @app.route para vincular uma função a uma rota de URL. Quando um usuário visita essa URL, a função associada é executada. No seu projeto de chatbot, você usará rotas para lidar com as requisições POST que contêm mensagens dos usuários e para processar os dados do documento.

HTML - CSS - JavaScript

Você tem à disposição um front-end de chatbot pronto para uso, construído com HTML, CSS e JavaScript. HTML estrutura o conteúdo web, CSS estiliza e JavaScript adiciona interatividade. Essas tecnologias criam uma interface de chatbot visualmente atraente e interativa.

Aqui está uma captura de tela da interface:

Interface do Chatbot

Objetivos de aprendizagem

Ao final deste projeto, você será capaz de:

- Explicar os fundamentos do Langchain e das aplicações de IA
- Configurar um ambiente de desenvolvimento para construir um assistente usando Python Flask
- Implementar a funcionalidade de upload de PDF para permitir que o assistente compreenda a entrada de arquivos dos usuários
- Integrar o assistente com modelos de código aberto para conferir a ele um alto nível de inteligência e a capacidade de entender e responder a solicitações dos usuários
- (Opcional) Implantar o assistente de PDF em um servidor web para uso por um público mais amplo

Pré-requisitos

Conhecimento básico de HTML/CSS, JavaScript e Python é desejável, mas não essencial. Cada etapa do processo e do código terá uma explicação abrangente neste laboratório.

Com esse contexto em mente, vamos começar seu projeto!

Configurando e entendendo a interface do usuário

Neste projeto, o objetivo é criar um chatbot com uma interface que permita a comunicação.

Primeiro, vamos configurar o ambiente executando o seguinte código:

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

O frontend usará HTML, CSS e JavaScript. A interface do usuário será semelhante a muitos chatbots que você vê e usa online. O código para a interface é fornecido e o foco deste projeto guiado é conectar essa interface com o backend que gerencia o upload dos seus documentos personalizados e o integra com um modelo LLM para obter respostas personalizadas. O código fornecido ajudará você a entender como o frontend e o backend interagem, e à medida que você avança, aprenderá sobre as partes importantes e como funciona, proporcionando uma compreensão clara de como o frontend funciona e como criar esta página da web simples.

Execute os seguintes comandos para recuperar o projeto, dar um nome apropriado e, finalmente, mover para esse diretório executando o seguinte:

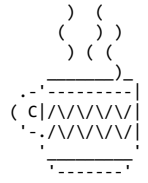
```
git clone https://github.com/sinanazeri/build_own_chatbot_without_open_ai.git
mv build_own_chatbot_without_open_ai build_chatbot_for_your_data
cd build_chatbot_for_your_data
```

instalando os requisitos para o projeto

```
pip install -r requirements.txt
```

```
pip install langchain-community
```

Tome uma xícara de café, levará de 5 a 10 minutos para instalar os requisitos (Você pode continuar este projeto enquanto os requisitos estão sendo instalados).



A próxima seção fornece uma breve compreensão de como o frontend funciona.

HTML, CSS e JavaScript

O arquivo `index.html` é responsável pelo layout e estrutura da interface web. Este arquivo contém o código para incorporar bibliotecas externas como JQuery, Bootstrap e FontAwesome Icons, além do código CSS (`style.css`) e JavaScript (`script.js`) que controlam o estilo e a interatividade da interface.

O arquivo `style.css` é responsável por personalizar a aparência visual dos componentes da página. Ele também gerencia a animação de carregamento usando keyframes CSS. Keyframes são uma forma de definir os valores de uma animação em vários pontos no tempo, permitindo uma transição suave entre diferentes estilos e criando animações dinâmicas.

O arquivo `script.js` é responsável pela interatividade e funcionalidade da página. Ele contém a maior parte do código e gerencia todas as funções necessárias, como alternar entre o modo claro e escuro, enviar mensagens e exibir novas mensagens na tela. Ele até permite que os usuários gravem áudio.

Compreendendo o trabalhador: Processamento de documentos e gerenciamento de conversas, parte 1

`worker.py` é parte de uma aplicação de chatbot que processa mensagens e documentos dos usuários. Ele utiliza a biblioteca `langchain`, que é uma biblioteca Python para construir aplicações de IA conversacional. É responsável por configurar o modelo de linguagem, processar documentos PDF em um formato que pode ser usado para recuperação de conversas e lidar com os prompts dos usuários para gerar respostas com base nos documentos processados. Aqui está uma visão geral do script:

Open **worker.py** in IDE

Sua tarefa é preencher os comentários do `worker.py` com o código apropriado.

Vamos detalhar cada seção no arquivo do trabalhador.
O worker.py é projetado para fornecer uma interface conversacional que pode responder perguntas com base no conteúdo de um determinado documento PDF.

O diagrama ilustra o procedimento de processamento de documentos e recuperação de informações, integrando perfeitamente um grande modelo de linguagem (LLM) para facilitar a tarefa de resposta a perguntas. Todo o processo acontece em `worker.py`. *crédito da imagem link.*

1. Inicialização `init_llm()`:
    - Configurando variáveis de ambiente: A variável de ambiente para o token da API do HuggingFace é configurada.
    - Carregando o modelo de linguagem: O modelo de linguagem WatsonX é inicializado com parâmetros especificados.
    - Carregando embeddings: Embeddings são inicializados usando um modelo pré-treinado.
2. Processamento de documentos `process_document(document_path)`:
   Esta função é responsável por processar um documento PDF fornecido.
    - Carregando o documento: O documento é carregado usando PyPDFLoader.
    - Dividindo texto: O documento é dividido em partes menores usando RecursiveCharacterTextSplitter.
    - Criando banco de dados de embeddings: Um banco de dados de embeddings é criado a partir dos fragmentos de texto usando Chroma.
    - Configurando a cadeia RetrievalQA: Uma cadeia RetrievalQA é configurada para facilitar o processo de resposta a perguntas. Esta cadeia utiliza o modelo de linguagem inicializado e o banco de dados de embeddings para responder perguntas com base no documento processado.
3. Processamento de prompts do usuário `process_prompt(prompt)`:
   Esta função processa um prompt ou pergunta do usuário.
    - Recebendo o prompt do usuário: O sistema recebe um prompt do usuário (pergunta).
    - Consultando o modelo: O modelo é consultado usando a cadeia de recuperação, e gera uma resposta com base no documento processado e no histórico de chat anterior.
    - Atualizando o histórico de chat: O histórico de chat é atualizado com o novo prompt e resposta.

Explorando cada seção

> IBM watsonX utiliza vários modelos de linguagem, incluindo os modelos Llama da Meta, que têm sido alguns dos modelos de linguagem de código aberto mais fortes publicados até agora (em fev de 2024).

1. Inicialização `init_llm()`:

Este código é para configurar e usar um modelo de linguagem de IA, do IBM watsonX:

1. **Configuração de credenciais**: Inicializa um dicionário com a URL do serviço e um token de autenticação (`"skills-network"`).
2. **Configuração de parâmetros**: Configura os parâmetros do modelo, como a geração máxima de tokens (`256`) e temperatura (`0.1`, controlando a aleatoriedade).
3. **Inicialização do modelo**: Cria uma instância do modelo com um `model_id` específico, usando as credenciais e parâmetros definidos acima, e especifica `"skills-network"` como o ID do projeto.
4. **Uso do modelo**: Inicializa uma interface (`WatsonxLLM`) com o modelo configurado para interação.

Este script é especificamente configurado para um projeto ou ambiente associado à “skills-network”.

Complete o seguinte código em `worker.py` inserindo os embeddings.

Neste projeto, você não precisa especificar seu próprio `Watsonx_API` e `Project_id`. Você pode apenas especificar `project_id="skills-network"` e deixar `Watsonx_API` em branco.

Mas é importante notar que este método de acesso é exclusivo para este ambiente Cloud IDE. Se você estiver interessado em usar o modelo/API fora deste ambiente (por exemplo, em um ambiente local), instruções detalhadas e mais informações estão disponíveis neste tutorial.

```
# placeholder for Watsonx_API and Project_id incase you need to use the code outside this environment
Watsonx_API = "Your WatsonX API"
Project_id= "Your Project ID"
# Function to initialize the language model and its embeddings
def init_llm():
    global llm_hub, embeddings
    logger.info("Initializing WatsonxLLM and embeddings...")
```

```
# Llama Model Configuration
MODEL_ID = "meta-llama/llama-3-3-70b-instruct"
WATSONX_URL = "https://us-south.ml.cloud.ibm.com"
PROJECT_ID = "skills-network"
# Use the same parameters as before:
#   MAX_NEW_TOKENS: 256, TEMPERATURE: 0.1
model_parameters = {
    # "decoding_method": "greedy",
    "max_new_tokens": 256,
    "temperature": 0.1,
}
# Initialize Llama LLM using the updated WatsonxLLM API
llm_hub = WatsonxLLM(
    model_id=MODEL_ID,
    url=WATSONX_URL,
    project_id=PROJECT_ID,
    params=model_parameters
)
logger.debug("WatsonxLLM initialized: %s", llm_hub)
#Initialize embeddings using a pre-trained model to represent the text data.
embeddings =  # create object of Hugging Face Instruct Embeddings with (model_name,  model_kwargs={"device": DEVICE} )

logger.debug("Embeddings initialized with model device: %s", DEVICE)
```

▼ Clique aqui para ver a solução

```
 embeddings = HuggingFaceInstructEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={"device": DEVICE}
)
logger.debug("Embeddings inicializados com o dispositivo do modelo: %s", DEVICE)
    )
```

Compreendendo o trabalhador, parte 2

2. **Processamento de documentos:** A função `process_document` é responsável por processar os documentos PDF. Ela utiliza o PyPDFLoader para carregar o documento, divide o documento em partes usando o `RecursiveCharacterTextSplitter` e, em seguida, cria um armazenamento vetorial (Chroma) a partir das partes do documento usando as incorporações do modelo de linguagem. Este armazenamento vetorial é então usado para criar uma interface de recuperação, que é utilizada para criar um `ConversationalRetrievalChain`.

    - **Carregamento de documentos:** O documento PDF é carregado usando a classe `PyPDFLoader`, que recebe o caminho do documento como argumento. (Exercício a fazer: atribuir PyPDFLoader(…) a `loader`)
    - **Divisão de documentos:** O documento carregado é dividido em partes usando a classe `RecursiveCharacterTextSplitter`. O `chunk_size` e o `overlap` podem ser especificados. (Exercício a fazer: atribuir RecursiveCharacterTextSplitter(…) a `text_splitter`)
    - **Criação de armazenamento vetorial:** Um armazenamento vetorial, que é uma espécie de índice, é criado a partir das partes do documento usando as incorporações do modelo de linguagem. Isso é feito usando a classe `Chroma`.
    - **Configuração do sistema de recuperação:** Um sistema de recuperação é configurado usando o armazenamento vetorial. Este sistema chama um `ConversationalRetrievalChain`, utilizado para responder perguntas com base no conteúdo do documento.

A fazer: complete as partes em branco.

```
# Function to process a PDF document
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Loading document from path: %s", document_path)
    # Load the document
    loader =  # ---> use PyPDFLoader and document_path from the function input parameter <---
    documents = loader.load()
    logger.debug("Loaded %d document(s)", len(documents))
    # Split the document into chunks, set chunk_size=1024, and chunk_overlap=64. assign it to variable text_splitter
    text_splitter = # ---> use Recursive Character TextSplitter and specify the input parameters <---
    texts = text_splitter.split_documents(documents)
    logger.debug("Document split into %d text chunks", len(texts))
    # Create an embeddings database using Chroma from the split text chunks.
    logger.info("Initializing Chroma vector store from documents...")
    db = Chroma.from_documents(texts, embedding=embeddings)
    logger.debug("Chroma vector store initialized.")
    # Optional: Log available collections if accessible (this may be internal API)
    try:
        collections = db._client.list_collections()  # _client is internal; adjust if needed
        logger.debug("Available collections in Chroma: %s", collections)
    except Exception as e:
        logger.warning("Could not retrieve collections from Chroma: %s", e)
    # Build the QA chain, which utilizes the LLM and retriever for answering questions.
    conversation_retrieval_chain = RetrievalQA.from_chain_type(
        llm=llm_hub,
```

```
        chain_type="stuff",
        retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
        return_source_documents=False,
        input_key="question"
        # chain_type_kwargs={"prompt": prompt}  # if you are using a prompt template, uncomment this part
    )
    logger.info("RetrievalQA chain created successfully.")
```

▼ Clique aqui para ver a solução

```
# Função para processar um documento PDF
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Carregando documento do caminho: %s", document_path)
    # Carregar o documento
    loader = PyPDFLoader(document_path)
    documents = loader.load()
    logger.debug("Carregado %d documento(s)", len(documents))
    # Dividir o documento em partes
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024, chunk_overlap=64)
    texts = text_splitter.split_documents(documents)
    logger.debug("Documento dividido em %d partes de texto", len(texts))
    # Criar um banco de dados de embeddings usando Chroma a partir das partes de texto divididas.
    logger.info("Inicializando o armazenamento vetorial Chroma a partir dos documentos...")
    db = Chroma.from_documents(texts, embedding=embeddings)
    logger.debug("Armazenamento vetorial Chroma inicializado.")
    # Opcional: Registrar coleções disponíveis, se acessíveis (isso pode ser uma API interna)
    try:
        collections = db._client.list_collections()  # _client é interno; ajuste se necessário
        logger.debug("Coleções disponíveis no Chroma: %s", collections)
    except Exception as e:
        logger.warning("Não foi possível recuperar coleções do Chroma: %s", e)
    # Construir a cadeia de QA, que utiliza o LLM e o recuperador para responder perguntas.
    conversation_retrieval_chain = RetrievalQA.from_chain_type(
        llm=llm_hub,
        chain_type="stuff",
        retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
        return_source_documents=False,
        input_key="question"
        # chain_type_kwargs={"prompt": prompt}  # se você estiver usando um modelo de prompt, descomente esta parte
    )
    logger.info("Cadeia RetrievalQA criada com sucesso.")
```

3. **Processamento de prompt (`process_prompt` function)**: Esta função lida com o prompt ou pergunta de um usuário, recupera uma resposta com base no conteúdo do documento PDF processado anteriormente e mantém um histórico de chat. Ela faz o seguinte:
    - Passa o prompt e o histórico de chat para o objeto `ConversationalRetrievalChain`. `conversation_retrieval_chain` é a ferramenta principal usada para consultar o modelo de linguagem e obter uma resposta com base no conteúdo do documento PDF processado.
    - Anexa o prompt e a resposta do bot ao histórico de chat.
    - Retorna a resposta do bot.

Aqui está um esqueleto da função `process_prompt` para o exercício:

```
# Function to process a user prompt
def process_prompt(prompt):
    global conversation_retrieval_chain
    global chat_history
    logger.info("Processing prompt: %s", prompt)
    # Query the model using the new .invoke() method
    output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
    answer = output["result"]
    logger.debug("Model response: %s", answer)
    # Update the chat history
    # TODO: Append the prompt and the bot's response to the chat history using chat_history.append and pass `prompt` `answer` as arguments
    # --> write your code here <--

    logger.debug("Chat history updated. Total exchanges: %d", len(chat_history))
    # Return the model's response
    return answer
```

▼ Clique aqui para ver a solução

```
# Função para processar um prompt do usuário
def process_prompt(prompt):
    global conversation_retrieval_chain
    global chat_history
    logger.info("Processando prompt: %s", prompt)
    # Consultar o modelo usando o novo método .invoke()
    output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
    answer = output["result"]
    logger.debug("Resposta do modelo: %s", answer)
    # Atualizar o histórico de chat
    chat_history.append((prompt, answer))
    logger.debug("Histórico de chat atualizado. Total de trocas: %d", len(chat_history))
    # Retornar a resposta do modelo
    return answer
```

4. **Variáveis globais:**
    - `llm` e `llm_embeddings` são usados para armazenar o modelo de linguagem e suas embeddings `conversation_retrieval_chain` e `chat_history` são usados para armazenar o chat e o histórico. `global` é usado dentro das funções `init_llm`, `process_document`, e `process_prompt` para indicar que as variáveis `llm`, `llm_embeddings`, `conversation_retrieval_chain`, e `chat_history` são variáveis globais. Isso significa que quando essas variáveis são modificadas dentro dessas funções, as mudanças persistirão fora das funções também, afetando o estado global do programa.

Aqui está o arquivo completo `worker.py`. O código final pode ser encontrado em `Worker_completed.py` também.

▼ Clique aqui para ver o worker.py completo

```
import os
import torch
import logging
# Configurar logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
from langchain_core.prompts import PromptTemplate  # Importação atualizada conforme aviso de depreciação
from langchain.chains import RetrievalQA
from langchain_community.embeddings import HuggingFaceInstructEmbeddings  # Novo caminho de importação
from langchain_community.document_loaders import PyPDFLoader  # Novo caminho de importação
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma  # Novo caminho de importação
from langchain_ibm import WatsonxLLM
# Verificar a disponibilidade de GPU e definir o dispositivo apropriado para computação.
DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"
# Variáveis globais
conversation_retrieval_chain = None
chat_history = []
llm_hub = None
embeddings = None
# Função para inicializar o modelo de linguagem e suas embeddings
def init_llm():
    global llm_hub, embeddings
    logger.info("Inicializando WatsonxLLM e embeddings...")
    # Configuração do Modelo Llama
    MODEL_ID = "meta-llama/llama-3-3-70b-instruct"
    WATSONX_URL = "https://us-south.ml.cloud.ibm.com"
    PROJECT_ID = "skills-network"
    # Usar os mesmos parâmetros de antes:
    #   MAX_NEW_TOKENS: 256, TEMPERATURE: 0.1
    model_parameters = {
        # "decoding_method": "greedy",
        "max_new_tokens": 256,
        "temperature": 0.1,
    }
    # Inicializar Llama LLM usando a API WatsonxLLM atualizada
    llm_hub = WatsonxLLM(
        model_id=MODEL_ID,
        url=WATSONX_URL,
        project_id=PROJECT_ID,
        params=model_parameters
    )
    logger.debug("WatsonxLLM inicializado: %s", llm_hub)
    # Inicializar embeddings usando um modelo pré-treinado para representar os dados de texto.
    ### --> se você estiver usando a API huggingFace:
    # Configurar a variável de ambiente para HuggingFace e inicializar o modelo desejado, e carregar o modelo no HuggingFaceHub
    # não se esqueça de remover llm_hub para watsonX
    # os.environ["HUGGINGFACEHUB_API_TOKEN"] = "SUA CHAVE DE API"
    # model_id = "tiiuae/falcon-7b-instruct"
    #llm_hub = HuggingFaceHub(repo_id=model_id, model_kwargs={"temperature": 0.1, "max_new_tokens": 600, "max_length": 600})
    embeddings = HuggingFaceInstructEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2",
        model_kwargs={"device": DEVICE}
    )
    logger.debug("Embeddings inicializadas com dispositivo do modelo: %s", DEVICE)
# Função para processar um documento PDF
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Carregando documento do caminho: %s", document_path)
    # Carregar o documento
    loader = PyPDFLoader(document_path)
    documents = loader.load()
    logger.debug("Carregado %d documento(s)", len(documents))
    # Dividir o documento em partes
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024, chunk_overlap=64)
    texts = text_splitter.split_documents(documents)
    logger.debug("Documento dividido em %d partes de texto", len(texts))
```

```
        # Criar um banco de dados de embeddings usando Chroma a partir das partes de texto divididas.
        logger.info("Inicializando banco de dados de vetores Chroma a partir dos documentos...")
        db = Chroma.from_documents(texts, embedding=embeddings)
        logger.debug("Banco de dados de vetores Chroma inicializado.")
        # Opcional: Registrar coleções disponíveis, se acessíveis (isso pode ser uma API interna)
        try:
            collections = db._client.list_collections()  # _client é interno; ajustar se necessário
            logger.debug("Coleções disponíveis no Chroma: %s", collections)
        except Exception as e:
            logger.warning("Não foi possível recuperar coleções do Chroma: %s", e)
        # Construir a cadeia de QA, que utiliza o LLM e o recuperador para responder perguntas.
        conversation_retrieval_chain = RetrievalQA.from_chain_type(
            llm=llm_hub,
            chain_type="stuff",
            retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
            return_source_documents=False,
            input_key="question"
            # chain_type_kwargs={"prompt": prompt}  # se você estiver usando um modelo de prompt, descomente esta parte
        )
        logger.info("Cadeia RetrievalQA criada com sucesso.")
    # Função para processar um prompt do usuário
    def process_prompt(prompt):
        global conversation_retrieval_chain
        global chat_history
        logger.info("Processando prompt: %s", prompt)
        # Consultar o modelo usando o novo método .invoke()
        output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
        answer = output["result"]
        logger.debug("Resposta do modelo: %s", answer)
        # Atualizar o histórico de chat
        chat_history.append((prompt, answer))
        logger.debug("Histórico de chat atualizado. Total de trocas: %d", len(chat_history))
        # Retornar a resposta do modelo
        return answer
    # Inicializar o modelo de linguagem
    init_llm()
    logger.info("Inicialização do LLM e embeddings concluída.")
```

Executando o aplicativo no CloudIDE

Para implementar seu chatbot, você precisa executar o `server.py` primeiro.

```
python3 server.py
```

Você terá a seguinte saída no terminal. Isso mostra que o servidor está em execução.

Server.py

Agora clique no seguinte botão para abrir sua aplicação:

Open the application

Uma nova janela será aberta para sua aplicação.

Aplicação Chatbot

Ótimo trabalho, você completou seu projeto!

Se você quiser entender o arquivo do servidor e JavaScript, continue com o projeto. Você também aprenderá a containerizar o aplicativo para implantação usando Docker.

Depois de ter a chance de executar e brincar com a aplicação, pressione `Crtl` (também conhecido como `control (^)` para Mac) e `C` ao mesmo tempo para parar o contêiner e continuar o projeto (como também é mencionado no terminal).

(opcional) Usando a API HuggingFace para o trabalhador

Outra opção para o watsonX é usar a API HuggingFace. No entanto, o tipo de modelo e o desempenho são muito limitados na versão gratuita. A partir de setembro de 2023, o Llama2 não é suportado gratuitamente, e você usará o modelo `falcon7b` em vez disso.

Instale as versões especificadas do LangChain e do HuggingFace Hub, copie e cole os seguintes comandos no seu terminal:

```
pip install langchain==0.1.17
pip install huggingface-hub==0.23.4
```

Você precisa atualizar `init_llm` e inserir sua chave da API do Hugging Face.

- O `HuggingFace` está hospedando alguns modelos LLM que podem ser chamados gratuitamente (se o modelo tiver menos de 10GB). Você também pode baixar qualquer modelo e executá-lo localmente.

O objeto `HuggingFaceHub` é criado com o `repo_id` especificado e parâmetros adicionais como `temperature`, `max_new_tokens` e `max_length` para controlar o comportamento do modelo. Aqui você pode encontrar mais exemplos.

- As embeddings são inicializadas usando uma classe chamada `HuggingFaceInstructEmbeddings`, um modelo pré-treinado chamado `sentence-transformers/all-MiniLM-L6-v2`, e uma lista de classificações de embeddings está disponível aqui. Este modelo de embedding mostrou um bom equilíbrio entre desempenho e velocidade.

- O modelo usa o dispositivo especificado (CPU ou GPU) para computação.

A fazer: Complete a função `init_llm()`

```
def init_llm():
    global llm_hub, embeddings
    # Set up the environment variable for HuggingFace and initialize the desired model.
    os.environ["HUGGINGFACEHUB_API_TOKEN"] = "Your HuggingFace API"
    # Insert the name of repo model
    model_id = "tiiuae/falcon-7b-instruct"

    # load the model into the HuggingFaceHub
    llm_hub = # --> specify hugging face hub object with (repo_id, model_kwargs={"temperature": 0.1, "max_new_tokens": 600, "max_length": 600})

    #Initialize embeddings using a pre-trained model to represent the text data.
    embeddings = # --> create object of Hugging Face Instruct Embeddings with (model_name,  model_kwargs={"device": DEVICE} )
```

▶ Clique aqui para ver a solução

Você também precisa inserir sua chave de API LLM. Aqui está uma demonstração para mostrar como obter sua chave de API.

Inicialize a chave de API HuggingFace da sua conta com os seguintes passos:

1. Vá para https://huggingface.co/
2. Faça login na sua conta (ou crie uma conta gratuita se for a sua primeira vez)
3. Vá para Configurações -> Tokens de Acesso -> clique em Novo Token (veja a imagem abaixo)
4. Selecione a opção de leitura ou escrita e copie o token

Token HuggingFace

▶ Clique aqui para ver o worker.py completo para a versão huggingface

Para implementar seu chatbot, você precisa executar o arquivo `server.py` primeiro.

```
python3 server.py
```

Agora clique no botão a seguir para abrir seu aplicativo:

Abrir o aplicativo

Uma nova janela será aberta para o seu aplicativo.

Aplicativo de Chatbot

Compreendendo o servidor

O servidor é como a aplicação será executada e se comunicará com todos os seus serviços. Flask é um framework de desenvolvimento web para Python e pode ser usado como um backend para a aplicação. É um framework leve e simples que torna rápido e fácil construir aplicações web.

Com Flask, você pode criar páginas e aplicações web sem precisar conhecer uma codificação complexa ou usar ferramentas ou bibliotecas adicionais. Você pode criar suas próprias rotas e lidar com solicitações de usuários, e também permite conectar-se a APIs e serviços externos para recuperar ou enviar dados.

Este projeto guiado usa Flask para lidar com o backend do seu chatbot. Isso significa que você usará Flask para criar rotas e lidar com solicitações e respostas HTTP. Quando um usuário interage com o chatbot através da interface frontal, a solicitação será enviada para o backend do Flask. O Flask então processará a solicitação e a enviará para o serviço apropriado.

Open **server.py** in IDE

Em `server.py`, no topo do arquivo, você importa `worker`, que se refere ao arquivo worker.py que você usará para lidar com a lógica central do seu chatbot. Abaixo das importações, a aplicação Flask é inicializada e uma política de CORS é definida. Uma política de CORS é usada para permitir ou impedir que páginas web façam solicitações a diferentes domínios do que aquele que serviu a página web. Atualmente, está definida como `*` para permitir qualquer solicitação.

O arquivo `server.py` consiste em 3 funções que são definidas como rotas, e o código para iniciar o servidor.

A primeira rota é:

```
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')
```

Quando um usuário tenta carregar a aplicação, ele inicialmente envia uma solicitação para ir ao endpoint `/`. Em seguida, ele acionará esta função `index` e executará o código acima. Atualmente, o código retornado pela função é uma função de renderização para mostrar o arquivo `index.html`, que é a interface do frontend.

As segunda e terceira rotas são o que será usado para processar todas as solicitações e manejar o envio de informações entre a aplicação. A função `process_document_route()` é responsável por lidar com a rota quando um usuário faz o upload de um documento PDF, processando o documento e retornando uma resposta. A função `process_message_route()` é responsável por processar a mensagem ou consulta de um usuário sobre o documento processado e retornar uma resposta do bot.

Por fim, a aplicação é iniciada com o comando `app.run` para rodar na porta `8080` e o host como `0.0.0.0` (ou seja, `localhost`).

(Opcional) Explicando o arquivo JavaScript `Script.js`

O arquivo JavaScript é responsável por gerenciar a interface do usuário e as interações de uma aplicação de chatbot. Ele está localizado na pasta `static`. Os principais componentes do arquivo são os seguintes:

1. **Processamento de mensagens:** A função `processUserMessage(userMessage)` envia uma solicitação POST para o servidor com a mensagem do usuário e aguarda uma resposta. O servidor processa a mensagem e retorna uma resposta que é exibida na janela de chat.

```
const processUserMessage = async (userMessage) => {
  let response = await fetch(baseUrl + "/process-message", {
    method: "POST",
    headers: { Accept: "application/json", "Content-Type": "application/json" },
    body: JSON.stringify({ userMessage: userMessage }),
  });
  response = await response.json();
  console.log(response);
  return response;
};
```

2. **Loading animations:** The functions `showBotLoadingAnimation()` and `hideBotLoadingAnimation()` show and hide a loading animation while the server is processing a message or document.

3. **Exibição de mensagens:** As funções `populateUserMessage(userMessage, userRecording)` e `populateBotResponse(userMessage)` formatam e exibem mensagens do usuário e respostas do bot na janela de chat.

```
```javascript
const populateUserMessage = (userMessage, userRecording) => {
  $("#message-input").val("");
  $("#message-list").append(
    `<div class='message-line my-text'><div class='message-box my-text${
      !lightMode ? " dark" : ""
    }'><div class='me'>${userMessage}</div></div></div>`
  );
  scrollToBottom();
};
```
```

```
const populateBotResponse = async (userMessage) => {
  // ... omitted for brevity
  $("#message-list").append(
    `<div class='message-line'><div class='message-box${!lightMode ? " dark" : ""}'>${response.botResponse.trim()}<br>${uploadButtonHtml}</div><
  );
  scrollToBottom();
};
```

4. **Input cleaning:** The function `cleanTextInput(value)` cleans the user's input to remove unnecessary spaces, newlines, tabs, and HTML tags.

```
```javascript
const cleanTextInput = (value) => {
  return value
    .trim() // remove espaços no início e no fim
    .replace(/[\n\t]/g, "") // remove quebras de linha e tabulações
    .replace(/<[^>]*>/g, "") // remove tags HTML
    .replace(/[<>&;]/g, ""); // sanitiza entradas
};
```

5. **Envio de arquivo:** O listener de evento para `$("#file-upload").on("change", ...)` gerencia o processo de envio de arquivos. Quando um arquivo é selecionado, ele lê os dados do arquivo e os envia para o servidor para processamento.

```
$("#file-upload").on("change", function () {
  const file = this.files[0];
  const reader = new FileReader();
  reader.onload = async function (e) {
    // Now send this data to /process-document endpoint
    let response = await fetch(baseUrl + "/process-document", {
      method: "POST",
      headers: { Accept: "application/json", "Content-Type": "application/json" },
      body: JSON.stringify({ fileData: e.target.result }),
    });
    response = await response.json6. **Chat Reset:** The event listener for `$("#reset-button").click(...)` provides a way to reset the chat, cl
```

6. **Redefinição do chat:** Isso fornece uma maneira de redefinir o chat, limpando todas as mensagens e começando de novo com a saudação inicial do bot.

```
$("#reset-button").click(async function () {
  // Clear the message list
  $("#message-list").empty();
  // Reset the responses array
  responses.length = 0;
  // Reset isFirstMessage flag
  isFirstMessage = true;
  // Start over
  populateBotResponse();
});
```

7. **Light/Dark mode switch:** The event listener for `$("#light-dark-mode-switch").change(...)` allows the user to switch between light and dark modes for the chat interface.

```
```javascript
$("#light-dark-mode-switch").change(function () {
  $("body").toggleClass("dark-mode");
  $(".message-box").toggleClass("dark");
  $(".loading-dots").toggleClass("dark");
  $(".dot").toggleClass("dark-dot");
  lightMode = !lightMode;
});
```

```
::page{title="Running the application"}
First, in the `server.py` you have the following code:
```python
```python
if __name__ == "__main__":
    app.run(debug=True, port=8000, host='0.0.0.0')
```

```
The line `if __name__ == "__main__":` checks if the script is being run directly. If so, it starts the Flask application with `app.run(debug=Tru
### Creating the Docker container
Docker allows for the creation of "containers" that package an application and its dependencies together. This allows the application to run con
The `git clone` from the second page already comes with a `Dockerfile` and `requirements.txt` for this application. These files are used to buil
There are 3 different containers that need to run simultaneously for the application to run and interact with Text-to-Speech and Speech-to-Text
### Starting the application
This image is quick to build as the application is quite small. These commands first build the application running the commands in the `Dockerfi
```bash
```markdown
docker build . -t build_chatbot_for_your_data
docker run -p 8000:8000 build_chatbot_for_your_data
```

```

Open app

*The application must be opened in a new tab since the minibrowser in this environment does not support certain required features.*

Your browser may deny “pop-ups” but please allow them for the new tab to open up.

At this point, the application will run but return `null` for any input.

Once you've had a chance to run and play around with the application, please press `Crtl` (a.k.a. `control (^)` for Mac) and `C` at the same time to stop the container and continue the project.

The application will only run while the container is up. If you make new changes to the files and would like to test them, you will have to rebuild the image.

# Conclusion

**Congratulations on completing this guided project!**
You learned how to implement “Retrieval Augmented Search”, in Generative AI. You also learned how to work with LLMs, and vector store, how to create Embeddings, and how to integrate everything using Langchain.You created a real application, a chatbot, using Python, Flask, and JavaScript and you packaged and deployed it using containers and Kubernetes. You can share your achievements on LinkedIn, Twitter, and other social media. The guided project detail page has buttons to help you do this.

## Next steps

If generative AI and large language models (LLMs) interest you, we encourage you to apply for a free trial of the IBM watsonx.ai. WatsonX is IBM enterprise-ready AI and data platform designed to multiply the impact of AI across your business. The platform comprises three powerful products: the watsonx.ai studio for new foundation models, generative AI and machine learning; and the watsonx.data fit-for-purpose data store, built on an open lakehouse architecture; and the watsonx.governance toolkit, to accelerate AI workflows that are built with responsibility, transparency, and explainability.

Moving forward, dive deeper into chatbot creation. The following guided project can assist you in acquiring the necessary skills for that endeavor.

**Create a voice assistant with IBM Watson**

Furthermore, you can delve into learning about Langchain and its functionalities. This allows you to add more capabilities to the chatbot, such as analyzing various types of files and generating output plots. Following guided project can be helpful.

**Create AI-powered apps with open source LangChain**

## Author(s)

**Sina Nazeri**

**Talha Siddiqui**
```