Price Waterhouse Coopers S.L.

Documentación Monty Bot

Autores :
Santiago Mourenza Rivero

September 23, 2024

Contents

1 Introducción

MontyBot es un bot de IA Generativa diseñado por Álvaro Montorio Piñeiro para crear DEMOS personalizadas. El bot tiene la capacidad de conectarse a cualquier modelo de lenguaje, ya sea de código abierto o de pago (la opción de modelos de pago estará disponible próximamente). Su objetivo es facilitar la demostración y prueba de diferentes modelos de IA en entornos controlados.

1.1 Instrucciones para ejecutar MontyBot

MontyBot es sencillo de utilizar, pero es importante tener en cuenta algunas características clave para asegurar su correcto funcionamiento. Para evitar fallos, sigue estas recomendaciones:

1. **Instalar Microsoft C++ Build Tools**

   Esto se puede hacer desde *este enlace*.

   Nota: Es necesario solicitar un ticket para su instalación.

2. **Desactivar la VPN antes de ejecutar el Bot**

   Dentro de la opción de desactivar la VPN, copiar la referencia del ticket en la página de este enlace. Una vez se reciba la resolución del ticket, copiar el código del mismo y seleccionar "desactivar". Es importante seleccionar la opción *Client Engagement - Need to use client supplied VPN* en la solicitud del ticket para que los permisos se otorguen sin demoras.

3. **Conectarse a una red *wifi* externa**

   Se recomienda el uso de un *Personal Hotspot* del iPhone o conectarse a una red doméstica.

   Nota: El bot no funcionará con la red *wifi* de la oficina.

4. **Añadir los documentos para los *chunks***

   Coloca los documentos necesarios para las respuestas del bot en la carpeta `data`.

5. **Crear un entorno virtual**

   Ejecutar el comando `py -m venv nombre_venv` para crear el entorno y `nombre_venv\Scripts\activate` para activarlo.

6. **Instalar librerías necesarias**

   Ejecutar el comando `pip install -r requirements.txt` para instalar las dependencias dentro del entorno virtual.

7. **Añadir Documentación**

   Asegúrate de que todos los documentos relevantes estén bien documentados dentro de la carpeta designada.

8. **Indexar Documentos**

   Utiliza el archivo `create_index.py` para indexar los documentos necesarios.

9. **Ejecutar MontyBot**

   Ejecutar el siguiente comando para iniciar el bot: `streamlit run app.py`.

1.2 Futuras Actualizaciones

Se prevé, en un periodo corto de tiempo, mejorar la herramienta dotándola de diferentes características nuevas:

- Conexión con Gemini y OpenAI.
- Sistema de feedback para cada mensaje del bot.
- Mejoras de rendimiento.

1.3 Configuraciones extra

Además de las configuraciones básicas, es necesario configurar el flujo de funcionamiento y el entorno del bot. En cuanto al flujo de funcionamiento, es necesario modificar el método `invoke_chain` en el fichero `langchain_utils.py`. Por otro lado, en el fichero `.env` es necesario añadir la API KEY de GROQ, el `PATH_INDEX` y, de forma opcional, se puede integrar Langsmish.

```
# Ejemplo de .env
GROQ_API_KEY=tu_api_key_aqui
INDEX_PATH =tu_path_index
LANGSMITH_API_KEY=tu_api_key_aqui #(opcional)
```

2 Análisis del Código

En esta sección vamos a analizar las funciones más importantes de los diferentes ficheros de código para esclarecer el funcionamiento del Bot.

2.1 Descripción del Script `create_index.py`

En este script se efectúan los primeros pasos necesarios para utilizar un chatbot. Estos pasos incluyen, entre otros, cargar las variables de entorno y crear el índice que será almacenado en forma de vectores. Estos vectores se compararán más adelante para que el bot pueda determinar qué datos son similares de manera rápida y eficiente. El funcionamiento es el siguiente:

- **Importación de librerías**:
  - `dotenv` para cargar variables de entorno.
  - `FastEmbedEmbeddings` de `langchain_community` para generar embeddings.
  - `Chroma` de `langchain_chroma` para manejar el almacenamiento vectorial.
  - `os` para operaciones del sistema.
  - `TextLoader` de `langchain_community.document_loaders` para cargar documentos.
  - `CharacterTextSplitter` de `langchain_text_splitters` para dividir el texto en fragmentos.
- **Carga de variables de entorno**:
  - Se carga el archivo `.env` para obtener las variables de entorno necesarias.
- **Definición de rutas**:
  - `INDEX_PATH` se obtiene de las variables de entorno.
  - `DIRECTORY` se establece como el directorio que contiene los documentos a procesar.
- **Inicialización de embeddings**:
  - Se crea una instancia de `FastEmbedEmbeddings` para generar embeddings de los documentos.
- **Configuración del divisor de texto**:
  - Se configura `CharacterTextSplitter` para dividir los documentos en fragmentos de 2000 caracteres sin solapamiento.
- **Inicialización de la lista de documentos**:
  - Se crea una lista vacía `all_docs` para almacenar los fragmentos de texto.
- **Carga y división de documentos**:

  - Itera sobre todos los archivos en el directorio especificado.
  - Verifica que cada archivo sea válido.
  - Carga cada archivo usando `TextLoader` con codificación UTF-8.
  - Divide el contenido del archivo en fragmentos usando `text_splitter`.
  - Agrega los fragmentos a la lista `all_docs`.
  - Imprime información sobre los archivos y fragmentos procesados.

- **Creación o carga del almacenamiento vectorial**:
  - Si el índice ya existe en `INDEX_PATH`, se carga usando `Chroma`.
  - Si no existe, se crea un nuevo almacenamiento vectorial a partir de los documentos en `all_docs` y se guarda en `INDEX_PATH`.

- **Mensajes de estado**:
  - Imprime mensajes de estado para indicar el inicio y fin del proceso, así como la cantidad de documentos procesados y si se ha cargado o creado el almacenamiento vectorial.

2.2 Descripción del Script `langchain_utils.py`

El script `langchain_utils.py` contiene la lógica para determinar qué documentos deben ser utilizados por el chatbot y qué modelo se va a emplear para generar respuestas. También maneja todo lo relacionado con las *queries*. Antes de describir las funciones es necesaria una configuración inicial:

- Se importan varias funciones y clases de `langchain` y otras librerías necesarias.
- Se cargan las variables de entorno utilizando `load_dotenv()`.

Función `get_model`

Devuelve un modelo de lenguaje basado en el nombre del modelo, la temperatura y el número máximo de *tokens* especificados.

```
@lru_cache(maxsize=None)
def get_model(model_name, temperature, max_tokens):
llm = {
    "llama3-70b-8192":
     ChatGroq(temperature=temperature,
     model_name="llama3-70b-8192",
     max_tokens=max_tokens),

     "llama3-8b-8192":
     ChatGroq(temperature=temperature,
     model_name="llama3-8b-8192",
     max_tokens=max_tokens),
```

```
    "mixtral-8x7b-32768":
    ChatGroq(temperature=temperature,
    model_name="mixtral-8x7b-32768",
    max_tokens=max_tokens),

    "gemma-7b-it":
    ChatGroq(temperature=temperature,
    model_name="mixtral-8x7b-32768",
    max_tokens=max_tokens),
    }
  return llm[model_name]
```

Configuración del Vectorstor y Retriever

Para hacer esta configuración se inicializa el embeddings mediante la llamada a `FastEmbedEmbeddings()`. También se configura el `Chroma` para persistir el vectorstore en el directorio especificado por `INDEX_PATH`. Y por último se configura el retriever para buscar en el vectorstore, el parámetro indica el número de chunks que son seleccionados. Veamos como está implementado:

```
embeddings = FastEmbedEmbeddings()
vectorstore = Chroma(
 persist_directory=INDEX_PATH,
 embedding_function=embeddings
)
retriever = vectorstore.as_retriever(
 search_kwargs={"k": 2}
)
```

Plantillas de Prompt

Se definen plantillas de prompt para generar consultas de búsqueda y para la interacción principal del chatbot.

```
prompt_query = ChatPromptTemplate.from_messages(
    [
        ("placeholder", "{chat_history}"),
        ("user", "{input}"),
        (
            "user",
            """Dado el contenido anterior, genera una
            consulta de busqueda para obtener informacion
            relevante para la conversacion.\
               La consulta debe utilizar palabras clave
               para que se entienda la esencia del mensaje.
            """,
        ),
    ]
)
```

```
prompt_main = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """Eres un asistente virtual centrado en el
            mundo de la cocina. Tu especilidad es ayudar
            a cocineros a realizar sus recetas.
            Te conocen como Monty, el Chef.
            Responde a la pregunta del usuario utilizando
            UNICAMENTE el contexto que tienes a continuacion:
            \n\n{context}.

            Solo debes incluir informacion que aparece en el
            contexto. En caso de no disponer de la informacion
            indica que no dispones de los datos suficientes.
            """,
        ),
        ("placeholder", "{chat_history}"),
        ("user", "{input}"),
    ]
)
```

Función `get_rag_chain` **y** `create_history`

Crea y devuelve una cadena de generación aumentada por recuperación, por sus siglas RAG (Retrieval Augmented Generation).

```
@lru_cache(maxsize=None)
def get_rag_chain(model_name, temperature, max_tokens):
 model = get_model(model_name, temperature, max_tokens)
 retriever_chain = create_history_aware_retriever(model,
                   retriever, prompt_query)
 document_chain = create_stuff_documents_chain(model,
                  prompt_main)
 rag_chain = create_retrieval_chain(retriever_chain,
             document_chain)
 return rag_chain
```

Crea un objeto `ChatMessageHistory` que se basa en el historial de mensajes del contexto del chat.

```
def create_history(messages):
     history = ChatMessageHistory()
     for message in messages:
         if message["role"] == "user":
             history.add_user_message(message["content"])
         else:
             history.add_ai_message(message["content"])
     return history
```

Función `invoke_chain`

Invoca el modelo de cadena de lenguaje para generar una respuesta basada en la pregunta dada y el historial dentro del contexto del chat.

```
def invoke_chain(question, messages,
                 model_name="llama3-70b-8192",
                 temperature=0,max_tokens=8192):

    chain = get_rag_chain(model_name, temperature, max_tokens)
    history = create_history(messages)
    response = ""

    for chunk in chain.stream({"input": question,
                               "chat_history":
                               history.messages}):
        if "answer" in chunk.keys():
            response += chunk["answer"]
            yield chunk["answer"]

    history.add_user_message(question)
    history.add_ai_message(response)
    invoke_chain.response = response
    invoke_chain.history = history
    invoke_chain.aux = aux
```

2.3 Descripción del Script `app.py`

El *script* `app.py` es el archivo principal para ejecutar Monty Bot utilizando Streamlit. Este *script* principalmente tiene las funciones que manejan la web en la que se va a utilizar el bot.

Configuración Inicial y Actualización de la Web

Se prepara la configuración de la página web dónde se interactuará con el bot y se utilizará la función `render_or_update_model_info` para renderizar o actualizar la información del modelo en la página web. Es importante decir que en caso de que la sesión se cierre se reiniciará el historial del chat para crear así un nuevo contexto.

Funcionalidad del Script

Una vez hemos definido las funciones que van a ser utilizadas pasamos a explicar paso a paso como funciona el código para obtener así una web con un chatbot funcional:

- **Opciones de modelos y tokens**:
  - Se definen las opciones de modelos disponibles y sus respectivos máximos de tokens.
- **Inicialización del modelo y del historial de chat**:
  - Si no están definidos en el estado de la sesión, se inicializan con valores predeterminados.
- **Configuración de la barra lateral**:
  - Título de la configuración del modelo.
  - Selección del modelo mediante un *selectbox*.
  - Ajuste de la temperatura del modelo con un *slider*.
  - Selección del máximo de tokens con un `number_input`.
  - Botón para vaciar el historial de chat que llama a `reset_chat_history`.
- **Renderizado o actualización de la información del modelo**:
  - Llama a `render_or_update_model_info` con el modelo seleccionado.
- **Mostrar mensajes del historial de chat:**
  - Itera sobre los mensajes en el historial y los muestra en la interfaz.
  - Si hay gráficos en los mensajes, los muestra usando `plotly_chart`.
- **Entrada del usuario:**
  - Campo de entrada para que el usuario escriba su mensaje.

- **Procesamiento de la entrada del usuario:**
  - Muestra el mensaje del usuario en el chat.
  - Llama a `invoke_chain` para obtener la respuesta del modelo.
  - Muestra la respuesta del modelo en el chat.
  - Si hay gráficos en la respuesta, los muestra usando `plotly_chart`.
  - Si hay recursos adicionales, los muestra como botones.
- **Actualización del historial de chat:**
  - Añade el mensaje del usuario y la respuesta del asistente al historial de chat en el estado de la sesión.

3 Resumen y Conclusión

El Monty Bot es una herramienta desarrollada por Álvaro Montorio Piñeiro para poder presentar demos a clientes de funcionamientos de chatbots basados en inteligencia artificial generativa. Es por ello por lo que que del modelo básico que hemos comentado se pueden hacer modificaciones al ser muy flexible. En el caso de uso particular que está en el repositorio traba un chatbot que te ayuda a hacer recetas de cocina. No obstante con los documentos y configuraciones adecuadas se adapta a las necesidades del usuario, además como sigue en desarrollo se espera mejorar las habilidades de la herramienta.

Imporante: Esta documentación es una descripción de alto nivel del funcionamiento del bot. Para información más detallada es importante leer el código documentado del repositorio en el cual se encuentra la herramienta desarrollada.