

Manual de Implementación

En este documento, se dará una explicación sobre el código que constituye la implementación integral del proyecto. Este conjunto de elementos engloba tanto un notebook alojado en Google Colab, identificado como "Document_Splicer+Embeddings.ipynb", que maneja la creación y actualización de la base de datos vectorial y el código que maneja la página web en la cual se despliega el bot de conversación.

En primer lugar, nos adentraremos en el análisis del script presente en Colab, el cual desempeña un papel esencial en el proceso global del proyecto. Este notebook, denominado "Document_Splicer+Embeddings.ipynb", se ha diseñado y estructurado para ejecutarse en el entorno colaborativo de Google Colab. Aquí, se lleva a cabo una serie de tareas específicas que abarcan desde la segmentación de documentos hasta la incorporación de embeddings.

Simultáneamente, la aplicación en Python, que constituye una parte fundamental de la implementación, se encarga de gestionar la interfaz y la lógica de la página web. Este código, permite la interacción efectiva con el bot de charla. La integración de este código con la página web es esencial para el despliegue eficiente y la operación adecuada del bot de conversación.

Generación y actualización de la base de datos vectorial

Document_Splicer+Embeddings.ipynb

En esta sección, se realizará un análisis detallado del contenido en las celdas del notebook de Colab, central en la ejecución del proyecto. Este documento asume la responsabilidad del procesamiento de los documentos, desde la segmentación inicial hasta la transformación de los textos en vectores de embedding. La transferencia de estos vectores a la base de datos de Pinecone es crucial para la gestión eficiente y recuperación de información. La conexión efectiva entre el notebook de Colab y la base de datos de Pinecone es fundamental para la cohesión del proyecto.

Esta parte se realizó en un notebook de Colab para facilitar la conexión con Google Drive, debido a que ambos servicios hacen parte del ecosistema de trabajo de Google y una de las restricciones de diseño era que la información estaría guardada en Google Drive.

En primera instancia se instalan las librerías necesarias para la ejecución del código, de entre esta destacan openai, langchain, tiktoken y pinecone-client

La librería OpenAI en Python permite interactuar con estos modelos para realizar tareas como generación de texto, traducción y otras aplicaciones relacionadas con el procesamiento del lenguaje natural (NLP).

La biblioteca LangChain facilita la interoperabilidad entre diferentes bibliotecas de procesamiento de lenguaje natural. Ofrece una interfaz unificada para trabajar con múltiples modelos y herramientas de NLP.

Tiktoken es una biblioteca ligera que ayuda a contar el número de tokens en un texto sin la necesidad de realizar solicitudes a una API. Esto es útil para controlar límites de uso en servicios de procesamiento de lenguaje natural que facturan según la cantidad de tokens procesados.

La biblioteca Pinecone-client proporciona una interfaz para interactuar con la base de datos de Pinecone.

```
!pip install --upgrade langchain openai -q
!pip install unstructured -q
!pip install unstructured[local-inference] -q
!apt-get install poppler-utils -q
!pip install "unstructured[all-docs]" -q
!pip install tiktoken -q
!pip install pinecone-client -q
```

Nota. Muestra de la librerías utilizadas para el script. Captura tomada del script de Colab

En la siguiente celda, se establece la clave de API de OpenAI, esencial para efectuar solicitudes de procesamiento a los modelos de lenguaje. Además, esta clave se utiliza para la facturación asociada al uso de dichos modelos, subrayando la importancia de mantenerla actualizada para garantizar un funcionamiento fluido y preciso del sistema.

```
import os
os.environ["OPENAI_API_KEY"] = "*****"
```

Nota. Establecimiento de la llave de uso de la API de OpenAI. Captura tomada del script de Colab

En este bloque de código, se configuran los permisos de acceso a Google Drive, el repositorio que alberga los documentos destinados a ser utilizados como material de referencia en el proyecto.

```
from google.colab import drive
drive.mount('/content/drive')
```

Nota. Establecimiento de los permisos de acceso a Drive. Captura tomada del script de Colab

En esta sección, se lleva a cabo la importación de componentes específicos de las bibliotecas previamente instaladas, incorporándose al espacio de trabajo. Este proceso es esencial para acceder a las funcionalidades particulares de estas bibliotecas durante la ejecución del código.

```
import openai
from langchain.document_loaders import DirectoryLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import Pinecone
from langchain.llms import OpenAI
from langchain.chains.question_answering import load_qa_chain
from langchain.document_loaders import UnstructuredFileLoader
from langchain.document_loaders import GoogleDriveLoader
import pinecone
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
```

Nota. Importación de las librerías al espacio de trabajo. Captura tomada del script de Colab

Esta celda carga los documentos a utilizar y se transforman en un “Document”. En el contexto de Langchain, un Document se refiere a un archivo o una pieza de información que se carga o se utiliza en el sistema para su procesamiento o análisis. Puede ser un documento de texto, un archivo PDF, un archivo de imagen u otro tipo de archivo. Los Documentos se pueden cargar desde diferentes fuentes (Langchain)

Esta transformación se realiza a través de un DocumentLoader. DirectoryLoader es una clase en Langchain que se utiliza para cargar documentos desde un directorio en el sistema de archivos. Permite especificar la ruta del directorio y un patrón de búsqueda para encontrar los archivos deseados. (Langchain)

```
directory = '/content/drive/MyDrive/MyDataFolder'

def load_docs(directory):
    loader = DirectoryLoader(directory)
    documents = loader.load()
    return documents

documents = load_docs(directory)
len(documents)
```

Nota. Carga de documentos y transformación en un "Document". Captura tomada del script de Colab

En esta celda, se aplican secciones a los documentos utilizando la función de LangChain llamada "RecursiveCharacterTextSplitter" creando la función "split_docs". Los parámetros clave de esta función son "chunk_size" y "chunk_overlap".

"chunk_size" determina el tamaño de los segmentos en caracteres. Este fraccionamiento es necesario para enviar la información a la API de OpenAI, ya que la cantidad de datos que pueden enviarse de una vez está limitada en términos de tokens. Es importante destacar que la conversión de caracteres a tokens no es de uno a uno.

"chunk_overlap" se refiere a los caracteres compartidos entre segmentos adyacentes. Por ejemplo, con un "chunk_overlap" de 100, los primeros 100 caracteres de un segmento coinciden con los últimos 100 del segmento anterior. Este solapamiento se implementa para evitar cortes disruptivos en la información de oraciones o párrafos, ya que tales cortes podrían afectar la coherencia en el bot.

Finalmente se hace uso de la función "split_docs" para generar una variable con los elementos seccionados llamada "docs".

```
def split_docs(documents, chunk_size=1000, chunk_overlap=100):  
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=chunk_overlap)  
    docs = text_splitter.split_documents(documents)  
    return docs  
  
docs = split_docs(documents)  
print(len(docs))
```

Nota. Seccionamiento de los documentos . Captura tomada del script de Colab

En esta celda, se crea una variable que especifica el modelo utilizado para generar vectores de embeddings más adelante. Además, se realiza una prueba para verificar la activación de la llave de OpenAI. Se logra esto enviando una cadena de texto, y al recibir un vector de embeddings como respuesta, se confirma la validez de la llave.

El vector resultante tiene una dimensión de 1536. Este valor representa la dimensionalidad de los embeddings generados, un dato fundamental para la correcta estructuración de la base de datos que contendrá estos vectores.

```
embeddings= OpenAIEmbeddings(model="text-embedding-ada-002")  
  
query_result = embeddings.embed_query("Hello world")  
len(query_result)  
  
1536
```

Nota. Generación de la variable embeddings . Captura tomada del script de Colab

En esta celda, se establece la configuración de la base de datos vectorial en Pinecone, designándola como un índice. En el inicio, se realiza la conexión con Pinecone mediante otra llave de API y la especificación del entorno de la nube.

Posteriormente, se procede a la eliminación de índices previos, permitiendo así la actualización de información de manera eficiente. Dado el bajo costo del modelo de embeddings de OpenAI y la ausencia de cargos por parte de Pinecone en este proceso, se simplifica la tarea de actualización que, de otro modo, requeriría intervención manual.

La creación del nuevo índice incluye la asignación de un nombre, la especificación de la dimensión de los vectores a actualizar (en el caso de los embeddings de OpenAI, 1536), y la elección de la métrica de consulta de similitud, que en este caso corresponde a la medida coseno utilizada por los modelos de OpenAI. Este proceso configura la base de datos para admitir la inserción y búsqueda eficiente de vectores de embeddings actualizados.

```
pinecone.init(  
    api_key="*****",  
    environment="gcp-starter"  
)  
  
pinecone.delete_index('langchain-demo')  
  
pinecone.create_index("langchain-demo", dimension=1536, metric="cosine")
```

Nota. Configuración de la Base de datos vectorial en pinecone . Captura tomada del script de Colab

En este bloque de código, generamos los vectores de embeddings a partir de los documentos y los insertamos en el índice creado. La librería de Pinecone facilita este proceso al encargarse directamente del manejo de los vectores. Es importante destacar que Pinecone realiza las solicitudes a OpenAI para obtener los embeddings, por lo que es crucial haber

configurado previamente la variable "embeddings" con la llave de uso de OpenAI. Este enfoque permite una integración fluida entre Pinecone y OpenAI, optimizando el flujo de trabajo para la generación y gestión de embeddings en la base de datos vectorial.

```
index_name = "langchain-demo"

index = Pinecone.from_documents(docs, embeddings, index_name=index_name)
```

Nota. Generación de los vectores de embeddings . Captura tomada del script de Colab

Con esta secuencia de celdas, se ha completado la generación de la base de datos vectorial. Las últimas celdas proporcionan varios métodos para realizar consultas al bot. Aprovechando la capacidad de Colab para ejecutar secciones de código sin necesidad de ejecutar el script completo, se facilita la interacción y prueba rápida de distintas consultas al sistema. Este enfoque modular y la flexibilidad de ejecución permiten explorar y evaluar eficientemente el rendimiento del bot en diversas situaciones y preguntas específicas.

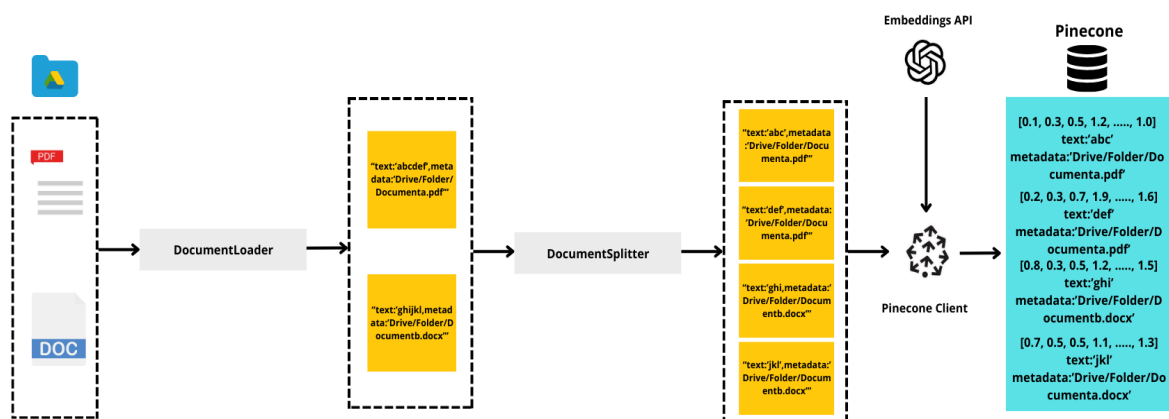
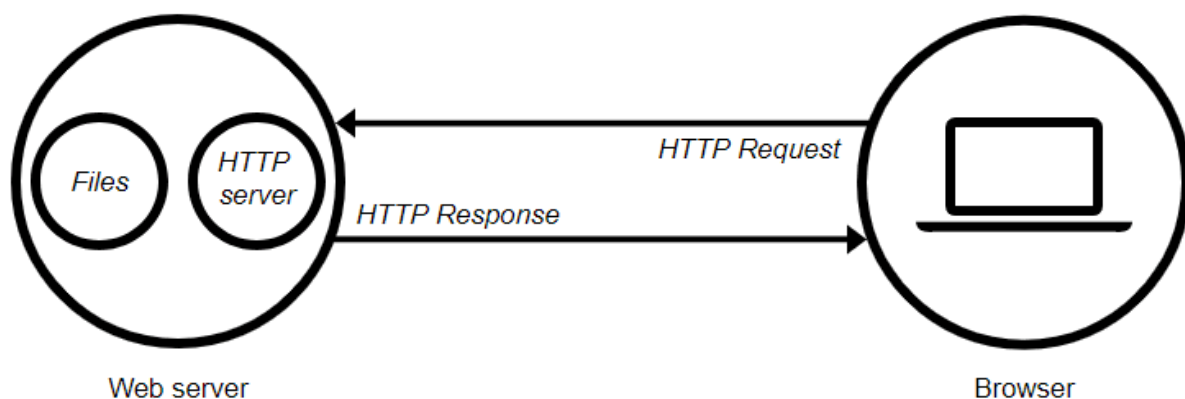


Diagrama de la Creación de Base de Datos Vectorial en Pinecone.

Implementación de la página web

Cuando escribimos la dirección de una página web en nuestro navegador web (ej. Chrome o Firefox), se realiza una petición de datos a un servidor donde se encuentra alojada la página usando un conjunto de protocolos (TCP/IP y HTTP). Si la petición se realiza de forma

correcta, el servidor nos responderá enviando paquetes de archivos entre los que se encuentra código y ficheros que nuestro navegador (cliente) organizará para mostrarnos el resultado en pantalla. Dentro de estos archivos se encuentra principalmente código HTML que definirá la estructura de la página web. A continuación, se aplicará el estilo y la apariencia gracias a CSS. Y, para terminar, toda la funcionalidad y dinamismo de la página será posible gracias a JavaScript. (Soriano, 2022)



Nota. Esquema petición/respuesta HTTP. Fuente: MDN Web docs

En esta implementación, el servidor web se ejecutará en el servicio en la nube PythonAnywhere. Este servicio se elige debido a su facilidad para ejecutar aplicaciones en Flask, un marco de trabajo en Python.

Servidor HTTP

requirements.txt

Este documento de texto incluye las bibliotecas necesarias para la ejecución adecuada del servidor web y facilita su configuración en diversos entornos, ya sea en la nube o de manera local. Con esta selección de bibliotecas y la estructuración adecuada, se busca asegurar una implementación fluida y adaptable del servidor web en diferentes plataformas.

pinecone_embed.py

Este script de Python establece la conexión del servidor web con los servicios de OpenAI y Pinecone, además de definir la función encargada de realizar preguntas al bot.

En la primera sección, se importan las bibliotecas necesarias, que son las mismas utilizadas en el notebook de Google Colab, con la adición de *“pinecone.core.client.configuration”*. Esta adición es necesaria para establecer una conexión a un sitio permitido a través del servidor proxy de PythonAnywhere. Dado que PythonAnywhere utiliza un proxy HTTP en *“proxy.server:3128”*, algunas bibliotecas requieren configuraciones específicas para utilizar este proxy.

La configuración especial se realiza para permitir el acceso a la API desde este servicio, siguiendo las pautas proporcionadas por PythonAnywhere. Esto asegura que la conexión con los servicios externos, como OpenAI y Pinecone, se realice de manera adecuada en el entorno específico de ejecución.

```

import os
os.environ["OPENAI_API_KEY"] = "*****"
from langchain.chains.question_answering import load_qa_chain
from langchain.chat_models import ChatOpenAI
import pinecone
from langchain.vectorstores import Pinecone
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.prompts import PromptTemplate
from pinecone.core.client.configuration import Configuration as OpenApiConfiguration

openapi_config = OpenApiConfiguration.get_default_copy()
# Here I am trying to connect to an insecure local proxy at 0.0.0.0:8081,
# however you can keep the verify_ssl=True if you are using a secure connection
openapi_config.verify_ssl = False
openapi_config.proxy = "http://proxy.server:3128"

```

Nota. Importación de librerías en pinecone_embed.py. Captura tomada del código guardado en pythonanywhere.

La conexión con la API de Pinecone se establece de manera similar a como se hizo en Colab, con la diferencia de que se incorporan los ajustes del servidor proxy previamente definidos. Una vez configurada la conexión, se conecta al mismo índice que fue creado en el entorno de Colab.

```

pinecone.init(
    api_key="*****",
    environment="gcp-starter",
    openapi_config=openapi_config
)

index_name = "langchain-demo"
embeddings= OpenAIEmbeddings(model="text-embedding-ada-002")
index=Pinecone.from_existing_index(index_name, embeddings)

```

Nota. Configuración de conexión con Pinecone en pinecone_embed.py. Captura tomada del código guardado en pythonanywhere.

Finalmente, se define la función encargada de obtener respuestas, haciendo uso de LangChain. La función se estructura de la siguiente manera:

Pregunta del Usuario: La pregunta planteada por el usuario.

Template: Una cadena que describe el prompt que se enviará a la API de OpenAI. Se podría enviar solo la pregunta del usuario, pero utilizar un prompt proporciona un contexto valioso y ayuda a mantener el enfoque del modelo. El template en la función inicia contextualizando al bot, proporciona pautas para limitar su comportamiento y deja espacio para el contexto extraído de la base de datos, la pregunta del usuario y la respuesta del modelo.

Configuraciones del Modelo: Se incluye el modelo a utilizar (en este caso, gpt-4) y el parámetro de "temperatura". La "temperatura" es un factor utilizado durante la generación de texto para controlar la diversidad y aleatoriedad de las respuestas. Una temperatura más baja (por ejemplo, 0.1) hace que el modelo sea más conservador y genere respuestas predecibles, mientras que una temperatura más alta (por ejemplo, 1.0) genera respuestas más diversas y creativas, pero a veces también más incoherentes. Debido a la necesidad de entregar información factual se escogió una temperatura baja, sin ser 0, pues este valor limita la capacidad del bot de responder ciertas preguntas.

```
def get_answer(query):
    question = query
    template = """El usuario está interesado en saber más sobre los estándares de acreditación de ABET para programas de ingeniería de la E3T.
    Utiliza los siguientes fragmentos de contexto para responder la pregunta al final de manera concisa. Ignora los signos de interrogación.
    {context}
    Question: {question}
    Helpful Answer: """
    QA_CHAIN_PROMPT = PromptTemplate.from_template(template)

    llm = ChatOpenAI(model_name="gpt-4", temperature=0.1)
    qa_chain = RetrievalQA.from_chain_type(
        llm,
        retriever=index.as_retriever(),
        chain_type_kwargs={"prompt": QA_CHAIN_PROMPT}
    )
    result = qa_chain({"query": question})
    return result["result"]
```

Nota. Definición de la Función 'get_answer' Captura tomada del código guardado en pythonanywhere.

flask_app.py

Este código es un script en Python que utiliza el framework web Flask para implementar un simple sistema de chat con un chatbot.

Importación de módulos:

```
from flask import Flask, render_template, request, jsonify, redirect
from pinecone_embed import get_answer
```

Nota. Importación de funciones y clases en 'flask_app.py' Captura tomada del código guardado en pythonanywhere.

Importa las funciones y clases necesarias de Flask para crear una aplicación web. Importa la función 'get_answer' desde el módulo 'pinecone_embed', que contiene la lógica del chatbot.

Creación de la aplicación Flask, definición de la ruta principal e inicialización de una lista de conversaciones:

```

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('chat.html')

conversations = []

```

Nota. Creación una instancia de la clase Flask en 'flask_app.py'. Captura tomada del código guardado en pythonanywhere.

Crea una instancia de la clase Flask para la aplicación web con `'app=Flask(__name__)'`, luego define una ruta principal que renderiza un template HTML llamado 'chat.html' al acceder a la raíz del sitio e inicia una lista vacía llamada `'conversations'` que se utilizará para almacenar las interacciones entre los usuarios y el chatbot.

Ruta para enviar mensajes al chatbot:

```

@app.route('/send_message', methods=['POST'])
def send_message():
    user_message = request.form['user_message']

    # Llama a la función get_answer para obtener la respuesta del chatbot
    bot_response = get_answer(user_message)
    #registra la conversación
    conversations.append({"user_message": user_message, "bot_response": bot_response})

    return jsonify({'bot_response': bot_response})

```

Nota. Definición de una ruta para enviar mensajes al bot en 'flask_app.py'. Captura tomada del código guardado en pythonanywhere.

Define una ruta que espera mensajes del usuario a través de una solicitud POST. Llama a la función `'get_answer'` para obtener la respuesta del chatbot basándose en el mensaje del usuario. Registra la conversación (mensaje del usuario y respuesta del bot) en la lista `conversations`. Devuelve la respuesta del bot en formato JSON.

Ruta para obtener conversaciones respondidas:

```
@app.route("/conversations", methods=["GET"])
def get_answered_conversations():
    # Devuelve las conversaciones como JSON
    return render_template("admin.html", answered=conversations)
```

Nota. Definición de una ruta para guardar conversaciones al bot en 'flask_app.py'. Captura tomada del código guardado en pythonanywhere.

Define una ruta para obtener las conversaciones respondidas. Renderiza un template HTML llamado 'admin.html' y pasa las conversaciones como una variable llamada 'answered'.

Ruta para eliminar preguntas:

```
@app.route("/delete_question", methods=["POST"])
def delete_question():
    question_to_delete = request.form.get("question") # Obtiene la pregunta a eliminar desde el formulario

    # Busca la pregunta en la lista de preguntas y eliminala
    for conversation in conversations:
        if conversation["user_message"] == question_to_delete:
            conversations.remove(conversation)
            break

    return redirect("/conversations")
```

Nota. Definición de una ruta para eliminar conversaciones del bot en 'flask_app.py'. Captura tomada del código guardado en pythonanywhere.

Define una ruta para eliminar preguntas y obtiene la pregunta a eliminar desde el formulario. Busca la pregunta en la lista de conversaciones y la elimina. Redirige a la página de conversaciones.

Bloque principal de ejecución:

```
if __name__ == '__main__':
    app.run()
```

Nota. Definición del bloque principal de ejecución en 'flask_app.py'. Captura tomada del código guardado en pythonanywhere.

Verifica si el script está siendo ejecutado directamente (no importado como un módulo) y, en ese caso, inicia la aplicación Flask.