

# Manual del Script: Cargar, Fragmentar y Subir Documentos a Pinecone

El funcionamiento de este script es automatizar el proceso de cargar archivos PDF desde Google Drive, extraer su contenido textual, fragmentarlos y enviarlos como vectores a Pinecone usando embeddings de OpenAI.

## 1. Configuración inicial del Web Service y Carpeta

```
import requests
import os
import time
=====
# CONFIGURACIÓN EXTERNA
# =====
WEB_SERVICE_URL="https://script.google.com/macros/s/AKfycbxdmU108e4NVuVUHS1q3_TNtD0r8I8runzh4InjJipJnt-1RGf9v1NKA2R43owJ80-w/exec" #V3
FOLDER_ID = "1b65CINGwfEVc3DyFK1Z90_bV6xXH_0Tk"
# =====
```

Se inicia importando tres librerías.

- `import requests` : Se utiliza para hacer solicitudes HTTP. En este caso, el script hace una **solicitud tipo GET** al Web Service que devuelve los archivos de Google Drive en formato JSON.
- `import os`: Interactúa con variables de entorno del sistema operativo, como claves API.
- `import time`: Sirve para hacer pausas entre procesos, medir tiempos de ejecución y/o esperar por conexiones o recursos

Seguido de esto se realiza la configuración inicial del Web Service y la carpeta, entendiendo que:

- `WEB_SERVICE_URL`: es el endpoint de un servicio web desarrollado en Google Apps Script que permite acceder al contenido de una carpeta en Google Drive.
- `FOLDER_ID`: es el identificador de la carpeta de Google Drive que contiene los archivos PDF que se van a procesar.

## 2. Función para cargar documentos desde Drive

```

# =====
# 1. FUNCIÓN: CARGAR DOCUMENTOS VIA WEBSERVICE
# =====
def load_docs_via_webservice(id_carpeta):
    """
    Llama al webservice 'LeerArchivosCarpetaDrive' y retorna un diccionario
    con los nombres de los archivos PDF y su contenido textual extraído.

    Parámetros:
    | id_carpeta (str): ID de la carpeta en Google Drive.

    Retorna:
    | dict: Diccionario con la estructura {nombre_archivo: texto_extraído}.
    """
    params = {"folder_id": id_carpeta}
    response = requests.get(WEB_SERVICE_URL, params=params)

    if response.status_code == 200:
        try:
            return response.json() # Retorna el JSON parseado
        except ValueError:
            return {"error": "No se pudo parsear la respuesta como JSON", "raw_response": response.text}
    else:
        raise Exception(f"Error al llamar al webservice: {response.status_code} - {response.text}")

```

Esta función `load_docs_via_webservice`, se encarga de conectarse a un servidor web y traer documentos en formato texto desde una carpeta de Google Drive (usando su ID).

- `params = {"folder_id": id_carpeta}` : Se construye un diccionario de parámetros que se le va a enviar al web service. El parámetro que se espera es `folder_id`, que se asigna al valor del ID de carpeta que se pasó como argumento.
- `response = requests.get(WEB_SERVICE_URL, params=params)` : Se hace una solicitud HTTP tipo GET a la URL del `WEB_SERVICE_URL` y se pasan los parámetros (`folder_id`) para que el servicio sepa qué carpeta analizar.

Si la respuesta es exitosa (código 200), devuelve un diccionario con los nombres de los archivos como claves y su contenido textual como valores. Si hay error al parsear la respuesta como JSON, retorna un mensaje de error.

### 3. Cargar documentos y convertirlos a documentos LangChain

```
# =====
# 2. CARGAR DOCUMENTOS Y CONVERTIRLOS A Documentos LangChain
# =====
from langchain.docstore.document import Document

# Llamar al webservice y obtener un diccionario {nombre: contenido}
docs_dict = load_docs_via_webservice(FOLDER_ID)

# Convertir cada par (nombre, contenido) en un objeto Document (para mantener compatibilidad)
documents = [Document(page_content=contenido, metadata={"source": nombre})
              for nombre, contenido in docs_dict.items()]

print(f"Se cargaron {len(documents)} documentos.")
```

- `from langchain.docstore.document import Document`: Aquí se importa la clase `Document` de la librería `langchain`. Esta clase es la forma estándar de representar un documento en `LangChain`.

Cada `Document` tiene dos partes principales:

- `page_content`: el texto del documento.
- `metadata`: un diccionario con información adicional (como el nombre del archivo, fecha, etc).

Seguido de esto se llama al `Webservice` para obtener los documentos, con `docs_dict` se ejecuta la función `load_docs_via_webservice` con el ID de la carpeta en `Google Drive`.

- Finalmente, con `documents`, se recorre el diccionario anterior (`docs_dict`) con un bucle tipo `list comprehension`. Para cada par (`nombre, contenido`):
  - Se crea un objeto `Document`.
  - El texto del archivo (`contenido`) va en `page_content`.
  - El nombre del archivo (`nombre`) se guarda como `metadato` en `metadata["source"]`.
- Se genera una lista de objetos `Document` que `LangChain` puede utilizar más adelante para:
  - Fragmentar texto.
  - Generar `embeddings`.
  - Hacer búsquedas semánticas.
  - Crear chatbots sobre documentos, etc.

Con `print(f"Se cargaron {len(documents)} documentos.")` se muestra por consola cuántos documentos fueron convertidos y están listos para ser procesados.

#### 4. Fragmentar los documentos

```
# =====
# 3. FRAGMENTAR LOS DOCUMENTOS
# =====
from langchain.text_splitter import RecursiveCharacterTextSplitter

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

# Fragmentar los documentos en chunks
docs = split_docs(documents)
print(f"Se generaron {len(docs)} fragmentos.")
```

Este bloque toma los documentos cargados (que pueden ser largos textos de archivos PDF, por ejemplo), y los **divide en fragmentos más pequeños** para facilitar el procesamiento por modelos de IA. Esto es útil porque los modelos de lenguaje tienen un límite en la cantidad de texto que pueden manejar a la vez.

`from langchain.text_splitter import RecursiveCharacterTextSplitter`: Importa una clase llamada `RecursiveCharacterTextSplitter` desde `LangChain`. Esta clase sirve para dividir textos largos en partes más pequeñas (fragmentos o *chunks*) de forma eficiente y controlada.

Seguido de esto, se realiza una función llamada `Split_docs`, la cual recibe:

- `documents`: una lista de objetos tipo `Document`, cada uno con texto y metadatos.
- `chunk_size`: el número máximo de caracteres que tendrá cada fragmento (por defecto: 1000).
- `chunk_overlap`: la cantidad de caracteres que se repiten entre fragmentos consecutivos (por defecto: 100).

Luego, se crea el objeto fragmentador:

- `text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=chunk_overlap)`: Donde se crea una instancia del `RecursiveCharacterTextSplitter` con los parámetros definidos. Este objeto usará una estrategia recursiva para dividir el texto, respetando límites de tamaño y solapamiento.

Se aplica el fragmentador a los documentos:

- `docs = text_splitter.split_documents(documents)`: Se aplica el método `.split_documents()` al listado de documentos. Devuelve una nueva lista de objetos `Document`, pero ahora cada uno representa **un fragmento del texto original**.

Con `return docs`, devuelve la lista de fragmentos generados.

Finalmente, se ejecuta la función y se muestra el resultado:

- `docs = split_docs(documents)`: Aquí se ejecuta la función pasando los documentos originales.
- `print(f"Se generaron {len(docs)} fragmentos.")`: Se imprime cuántos fragmentos se generaron.

## 5. Inicializar y crear el índice en Pinecone

```
# =====  
# 4. INICIALIZAR Y CREAR EL ÍNDICE EN PINECONE  
# =====  
# Configura la clave de Pinecone (asegúrate de que esté definida en tu entorno)  
os.environ["PINECONE_API_KEY"] = "pcsk_3vyEYS_3g3zbwHHeidVgmsNF76brK5QRvhJ66hZC14E5L6hmVbrTngDzZgpFTXZ6ChDuhP"  
  
from pinecone import Pinecone, ServerlessSpec  
from langchain_openai import OpenAIEmbeddings  
  
pc = Pinecone(api_key=os.environ.get("PINECONE_API_KEY"))  
index_name = "integrational"
```

**Pinecone** es una base de datos vectorial. Sirve para almacenar y buscar "embeddings", es decir, representaciones numéricas de textos, imágenes u otros datos, que nos permiten encontrar similitudes entre ellos. Es muy útil cuando se trabaja con inteligencia artificial, como chatbots o motores de búsqueda inteligentes.

Lo primero que hay que hacer es definir la clave de API de pinecone en las variables de entorno:

```
os.environ["PINECONE_API_KEY"] =  
"pcsk_3vyEYS_3g3zbwHHeidVgmsNF76brK5QRvhJ66hZC14E5L6hmVbrTngDzZgpFTXZ6Ch  
DuhP"
```

- `os.environ` es un diccionario de Python que permite leer o definir variables del sistema operativo.
- Aquí se asigna la clave privada de Pinecone a una variable del entorno llamada `"PINECONE_API_KEY"`.

- Esto es importante porque la clave no debe escribirse directamente muchas veces en el código por seguridad.

Después, se importan las librerías necesarias:

- `from pinecone import Pinecone, ServerlessSpec`
- `from langchain_openai import OpenAIEmbeddings`

Pinecone: Es la clase principal para conectarse y trabajar con tu cuenta e índices de Pinecone.

ServerlessSpec: Sirve para configurar el entorno serverless si lo necesitas (aunque aquí no se usa explícitamente).

OpenAIEmbeddings: Se usa después para convertir los textos a vectores con modelos de OpenAI.

Finalmente, se realiza la inicialización con la conexión a Pinecone, y se define el índice:

- `pc = Pinecone(api_key=os.environ.get("PINECONE_API_KEY"))` : Se crea una instancia del cliente de Pinecone usando la API key guardada, y con `os.environ.get` extrae esa clave del entorno. Esto permite que el código pueda interactuar con tu cuenta de Pinecone: crear índices, enviar datos, buscar, etc.
- `index_name = "integrationai"`: Este es el nombre que se le da al índice de vectores que se creará (o al que se conectará). Un índice en Pinecone es como una base de datos donde se guardan los vectores. Es importante tener en cuenta que el nombre debe ser único dentro de tu cuenta de Pinecone.

## 6. Inicializar el modelo de embeddings (OpenAI)

```
# =====
# 5. INICIALIZAR EL MODELO DE EMBEDDINGS (OpenAI)
# =====
openai_api_key = os.environ.get('sk-proj-4fkcyhYJV26jOz0T0gujFz-X3nMWr10IkKMT1FrzsSovgTym8z6wIN9m15J2IRIT6wPp2T1rVL73B1bkFJY5-fzEfIHxNzXe52ajZnkrwUo2vI_xM9jktCY2aoK78Ca8taalbnA13gvSE20G-k-wDLoB'
| | | | | 'sk-proj-XC40Hocot6r3t6FdYw8ZPggaKXRdHAQez_0_03ksF4Ed35D8qInvantdy8fpli_roHEzKfmVT3B1bkF35DVvhEzsd1E5MfQoVE1UK3q33kjIZm69FZKk7Y7Wm1x989o572fr59UA4EF0V26tUmBcnz25QA'
model_name = 'text-embedding-ada-002'

# Inicializar el modelo de embeddings
embed = OpenAIEmbeddings(
    model=model_name,
    openai_api_key=openai_api_key
)
```

- `openai_api_key = os.environ.get('sk-proj-...')` or `'sk-proj-...'` : Se intenta obtener la clave API de OpenAI desde las variables de entorno (`os.environ.get(...)`).
- `model_name = 'text-embedding-ada-002'` : Este es el modelo de OpenAI que se usará, es un modelo optimizado para generar *embeddings* eficientes y precisos a bajo costo.

- Finalmente, con embed se inicializa un objeto OpenAIEmbeddings, que luego usará el modelo indicado para convertir texto en vectores.

## 7. Enviar los vectores a Pinecone

```
# =====
# 6. ENVIAR LOS VECTORES A PINECONE
# =====
from langchain_pinecone import PineconeVectorStore

vectorstore_from_docs = PineconeVectorStore.from_documents(
    docs,
    index_name=index_name,
    embedding=embed
)

print("Proceso completado: vectores enviados a Pinecone.")
```

Pinecone es un servicio de **almacenamiento vectorial**. Te permite guardar vectores (como los embeddings del paso anterior) y hacer **búsquedas semánticas** sobre ellos.

- `from langchain_pinecone import PineconeVectorStore`: Importa la clase `PineconeVectorStore`, que permite interactuar con Pinecone usando LangChain.
- Finalmente, `vectorstore_from_docs` es la parte mas importante de este bloque. Lo que hace es:
  - `docs`: Es la lista de documentos fragmentados (del paso 3), cada uno con `page_content` y metadatos como el nombre del archivo.
  - `index_name=index_name`: Es el nombre del índice de Pinecone en el que se van a guardar los vectores. Ya debe estar creado en tu cuenta de Pinecone.
  - `embedding=embed`: Es el objeto del paso anterior que permite convertir los fragmentos de texto en vectores.
- Internamente lo que se puede observar es que cada documento en `docs` es procesado con `embed` para obtener su `embedding`. Esos vectores se almacenan en el índice `integrationai` de Pinecone, lo que permite que más adelante puedas **consultar** ese índice con una pregunta y obtener los fragmentos más relevantes según el significado.