Trabajo practico final

TUIA NLP 2024
Brisa Moresco

Introducción

En este ejercicio, se implementa un sistema que utiliza técnicas de recuperación y generación de información para responder consultas de manera efectiva. Este proceso incluye la descarga y procesamiento de archivos PDF, la segmentación del texto en fragmentos manejables y la limpieza del texto para su posterior análisis.

Descarga de Recursos de NLTK

```
# Descargar recursos de NLTK
nltk.download('punkt')
nltk.download('stopwords')
```

Para procesar texto en el lenguaje natural, se utilizan dos recursos principales de la biblioteca NLTK (Natural Language Toolkit).

Descarga de Archivos PDF desde URL

```
# Función para descargar archivos PDF desde URL
def download_pdf_from_url(url, save_path):
    response = requests.get(url)
    with open(save_path, 'wb') as f:
        f.write(response.content)
```

Esta función facilita la obtención de documentos PDF para su posterior procesamiento.

División del Texto en Fragmentos Usando LangChain

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Función para dividir el texto en fragmentos usando LangChain
def split_text_with_langchain(text, chunk_size=512, chunk_overlap=50):
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_overlap=chunk
    return text_splitter.split_text(text)
```

Se descargaron archivos PDF de diversas URLs para proporcionar una base de datos textual para el sistema. Estos documentos se procesaron para extraer texto que será utilizado en la generación de respuestas del chatbot.

Justificaciones

- **NLTK**: Se utiliza para preprocesar el texto mediante la tokenización y la eliminación de palabras comunes. Esto prepara el texto para análisis más profundos como la generación de embeddings y la consulta.
- **Descarga de PDF**: Permite la obtención de documentos relevantes desde la web, esencial para el análisis basado en documentos reales.
- **División del Texto**: La segmentación en fragmentos facilita la búsqueda y recuperación de información específica sin procesar documentos completos a la vez, optimizando el rendimiento del sistema.

Implementación del Sistema de Recuperación y Generación Aumentada (RAG)

En esta sección, se aborda la limpieza de texto y la descarga de archivos PDF. La limpieza del texto es crucial para asegurar que el análisis posterior sea preciso y eficiente, y la descarga de archivos PDF permite obtener datos

relevantes desde la web

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

def clean_text(text):
    # Eliminar caracteres especiales y números
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Tokenizar el texto
    tokens = word_tokenize(text)
    # Eliminar stopwords
    stop_words = set(stopwords.words('spanish'))
    tokens = [word for word in tokens if word.lower() not in stop_words]
    # Reconstruir el texto limpio
    clean_text = ' '.join(tokens)
    return clean_text.strip()
```

Esta función asegura que el texto se depure de información irrelevante y esté listo para análisis posteriores.

Descarga y Procesamiento de Archivos PDF

El siguiente código gestiona la descarga de archivos PDF desde URLs especificadas, su procesamiento para extraer texto, y la eliminación de archivos temporales:

```
pdf_urls = [
    "https://aranedasombra.com/public_html/wp-content/uploads/2017/11/Pan-Pizzas-y-Empanadas-
    "http://www.ladyweb.com/HotPizzaRecipes.pdf",
    "https://www.juntadeandalucia.es/export/drupaljda/salud_5af9586a41e8b_mis_recetas_sin_glu
]
# Descargar archivos PDF localmente y procesarlos
texts = []
for i, url in enumerate(pdf_urls):
    pdf_filename = f"documento_{i+1}.pdf"
    download_pdf_from_url(url, pdf_filename)

    # Procesar el PDF
    try:
        doc = fitz.open(pdf_filename)
        text = ""
        for page in doc:
            text += page.get_text()
        texts.append(text)
    finally:
        # Cerrar el documento y eliminar archivo temporal
        doc.close()
        os.remove(pdf_filename)
```

Justificación:

- **Limpieza del Texto**: La limpieza y tokenización del texto son necesarias para eliminar ruido y mejorar la calidad de la información para análisis. Las stopwords son eliminadas para enfocarse en términos relevantes.
- **Descarga y Procesamiento de PDF**: La descarga de PDFs desde URLs específicas permite obtener documentos reales para análisis. La extracción y limpieza del texto aseguran que solo la información relevante se conserve para los siguientes pasos del procesamiento.

Implementación del Sistema de Recuperación y Generación Aumentada (RAG)

En esta sección, se aborda la división de textos en fragmentos manejables y la generación de embeddings para cada fragmento. Estos pasos son esenciales para la eficiencia del sistema de recuperación de información y la mejora de la precisión en la generación de respuestas.

División de Textos en Fragmentos

El siguiente código divide los textos procesados en fragmentos más pequeños utilizando la biblioteca LangChain:

```
# Dividir los textos en fragmentos usando LangChain
chunks = [split_text_with_langchain(text) for text in texts]
clean_chunks = [clean_text(chunk) for sublist in chunks for chunk in sublist]
```

Generación de Embeddings

Para representar los fragmentos de texto de manera numérica, se generan embeddings utilizando el modelo `SentenceTransformer`:

```
from sentence_transformers import SentenceTransformer

# Cargar el modelo preentrenado de Sentence Transformers
model = SentenceTransformer('paraphrase-MiniLM-L6-v2')

# Generar embeddings para los fragmentos de texto
embeddings = model.encode(clean_chunks)
```

Justificación:

- **División de Textos**: Dividir los textos en fragmentos manejables permite que el sistema procese grandes volúmenes de datos de manera eficiente, mejorando la precisión de las consultas y la generación de respuestas.
- **Limpieza de Fragmentos**: La limpieza asegura que solo el contenido relevante sea considerado, reduciendo el ruido en los embeddings generados.
- **Generación de Embeddings**: Los embeddings son fundamentales para la comparación de textos y la recuperación de información. Utilizar un modelo preentrenado asegura que los embeddings capturen la semántica de manera efectiva.

Configuración y Uso de ChromaDB

En esta parte, se configura y utiliza ChromaDB para almacenar los embeddings generados para los fragmentos de texto. ChromaDB es una base de datos orientada a la recuperación de embeddings, lo cual es crucial para realizar búsquedas eficientes y precisas en grandes volúmenes de datos.

Configuración de ChromaDB

```
import chromadb

# Configurar el cliente de ChromaDB
client = chromadb.Client()
collection = client.get_or_create_collection("my-collection")
```

Función para Dividir en Lotes

```
# Función para dividir en lotes
def batch(iterable, n=1):
    l = len(iterable)
    for ndx in range(0, l, n):
        yield iterable[ndx:min(ndx + n, l)]
```

Almacenamiento de Embeddings en Lotes

```
# Guardar los embeddings en lotes más pequeños
batch_size = 100  # Puedes ajustar el tamaño del lote según sea necesario
for document_batch, embedding_batch, id_batch in zip(
        batch(clean_chunks, batch_size),
        batch(embeddings, batch_size),
        batch([str(i) for i in range(len(clean_chunks))], batch_size)):

    collection.add(
        documents=document_batch,
        embeddings=[embedding.tolist() for embedding in embedding_batch],
        ids=id_batch
    )
```

Verificación de Documentos Añadidos

```
# Verificar que los documentos se han añadido correctamente
print("Documentos añadidos correctamente:")
for i, chunk in enumerate(clean_chunks):
    # Utilizamos el índice (i) como el ID del documento
    print(f"Documento ID: {i}, Texto: {chunk}, Embedding: {embeddings[i].tolist()}")
```

Justificaciones

- **Configuración de ChromaDB**: La configuración permite almacenar y gestionar los embeddings de manera eficiente. La colección en ChromaDB facilita la organización y recuperación de los documentos.
- **División en Lotes**: Dividir el proceso en lotes mejora el rendimiento y evita problemas de memoria al manejar grandes volúmenes de datos.
- **Verificación**: Verificar los documentos añadidos asegura que la información se haya almacenado correctamente y es accesible para consultas futuras.

Carga y Procesamiento de Datos desde CSV

En esta sección, se describe el proceso de carga de datos desde un archivo CSV. Se intentó cargar el archivo desde una URL, pero debido a problemas técnicos, se optó por cargar el archivo localmente en su lugar.

```
import pandas as pd

# Especifica la ruta del archivo cargado en Colab
csv_filename = 'recipes (5).csv'

# Cargar el archivo CSV en un DataFrame de Pandas
df = pd.read_csv(csv_filename, delimiter=',', quotechar='"', encoding='latin-1', dtype=str)
```

Clasificación con LLM (Modelos de Lenguaje de Gran Tamaño)

Para la clasificación de texto utilizando un modelo preentrenado de LLM, se utilizó el pipeline `zero-shot-classification` de la biblioteca `transformers`. Este enfoque permite clasificar texto en categorías sin necesidad de entrenamiento previo específico para esas categorías.

```
from transformers import pipeline

# Cargar el modelo preentrenado de clasificación de texto
llm_classifier = pipeline('zero-shot-classification', model="facebook/bart-large-mnli")

# Función para clasificar texto usando LLM
def classify_with_llm(text, labels):
    result = llm_classifier(text, candidate_labels=labels)
    return result['labels'][0]

# Ejemplo de uso
text_to_classify = "Este es un ejemplo de texto para clasificar"
labels = ["positivo", "negativo"]

llm_result = classify_with_llm(text_to_classify, labels)
print("Clasificación con LLM:", llm_result)
```

Clasificación con SVM (Máquinas de Vectores de Soporte) con Embeddings

En este enfoque, se utiliza un clasificador SVM entrenado con embeddings de texto generados por un modelo de `Sentence Transformers`. Se entrenó un clasificador SVM para distinguir entre clases de texto basado en estos embeddings.

```

# Datos de entrenamiento (ajusta estos textos y etiquetas según sea necesario)
train_texts = ["Texto de entrenamiento positivo", "Texto de entrenamiento negativo"]
train_embeddings = model.encode(train_texts)
train_labels = [1, 0]  # 1 para positivo, 0 para negativo

# Crear y entrenar el clasificador SVM
svm_classifier = SVC()
svm_classifier.fit(train_embeddings, train_labels)

# Función para clasificar texto usando SVM con embeddings
def classify_with_svm(text):
    test_embedding = model.encode([text])
    prediction = svm_classifier.predict(test_embedding)
    return prediction[0]
```

`classify_with_svm(text)`: Función que clasifica el texto generando su embedding y realizando la predicción con el clasificador SVM.

Justificaciones

- **Clasificación con LLM**: Permite la clasificación sin necesidad de entrenamiento específico para cada tarea. Es útil para clasificaciones rápidas y sin requerimientos de datos de entrenamiento extensivos.
- **Clasificación con SVM y Embeddings**: Proporciona un enfoque entrenado y personalizado que puede ser más preciso si se dispone de un conjunto de datos de entrenamiento representativo. Utiliza embeddings para capturar

la semántica del texto.

Consultas Dinámicas y Filtración de Datos

Se implementaron consultas SPARQL para obtener datos de DBpedia y se utilizó la filtración dinámica de datos en un DataFrame de Pandas. Se configuró una conexión al endpoint SPARQL de DBpedia para ejecutar consultas dinámicas que recuperan datos relacionados con un término específico.

```
from SPARQLWrapper import SPARQLWrapper, JSON

# Configurar la conexión al endpoint SPARQL
sparql = SPARQLWrapper("http://dbpedia.org/sparql")

# Definir la consulta SPARQL dinámica
prompt = "Buscar elementos relacionados"
query = f"""
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    SELECT ?s ?o
    WHERE {{
        ?s ?p ?o .
        FILTER(CONTAINS(STR(?o), "{prompt}"))
    }}
"""

# Establecer el tipo de salida y ejecutar la consulta
sparql.setQuery(query)
sparql.setReturnFormat(JSON)
results = sparql.query().convert()

# Procesar los resultados
for result in results["results"]["bindings"]:
    print(result["s"]["value"], result["o"]["value"])
```

Filtración Dinámica de Datos en DataFrame

Busca y filtra elementos relevantes en un DataFrame de Pandas basado en un prompt específico. Utilice el promt vacio para abarcar todas las filas de ambas columnas.

```
def buscar_en_tablas(message):
    # Crear una serie combinando las columnas 'name' y 'description'
    combined_series = df[['name', 'description']].apply(lambda row: ' '.join(row.values.astyp

    # Prompt para abarcar todo el contenido
    prompt = ""

    # Filtrar el DataFrame basado en la combinación de las columnas 'name' y 'description'
    filtered_df = df[combined_series.str.contains(message, case=False, na=False)]

    if not filtered_df.empty:
        return filtered_df.head().to_dict(orient='records')
    return None
```

Justificaciones

- **Consultas SPARQL**: Permiten obtener datos estructurados de bases de datos semánticas como DBpedia de manera dinámica, lo cual es útil para enriquecer la información disponible en el sistema.

- **Filtración Dinámica de Datos**: Facilita la obtención de datos específicos de un conjunto más grande, mejorando la eficiencia en la búsqueda de información relevante.

Consultas Dinámicas y Filtración de Datos

Se ha trabajado en la generación de tríadas a partir de un archivo pdf para estructurar datos de recetas de pizzas.

- Utilizamos `requests` para descargar el PDF desde la URL y guardarlo localmente.
- Utilizamos `fitz` para abrir el archivo PDF y extraer el texto de cada página.
- Asumimos que el texto extraído tiene un formato en el que cada línea contiene datos separados por comas. Esto puede variar dependiendo de cómo esté estructurado el texto en el PDF.
- Dividimos el texto en líneas y luego en partes. Cada parte corresponde a un atributo de la tríada.
- Ajustamos la forma en que se divide y se extrae la información según el formato específico del PDF.

```
import fitz  # PyMuPDF
import re

# URL del PDF
pdf_url = "https://clubniva.com/media/kunena/attachments/16/pizza.pdf"

# Descargar el archivo PDF localmente
pdf_filename = 'pizza.pdf'
import requests

response = requests.get(pdf_url)
with open(pdf_filename, 'wb') as file:
    file.write(response.content)

# Función para extraer texto del PDF
def extract_text_from_pdf(pdf_filename):
    text = ""
    doc = fitz.open(pdf_filename)
    for page in doc:
        text += page.get_text()
    doc.close()
    return text

# Extraer el texto del PDF
pdf_text = extract_text_from_pdf(pdf_filename)

# Función para procesar el texto extraído
def generate_triples_from_text(text):
    triples = []

    # Aquí asumimos que el texto sigue un formato específico
    # Vamos a dividir el texto en líneas y luego en partes
    lines = text.split('\n')
    for line in lines:
        # Dividir la línea en partes (ajustar el delimitador si es necesario)
        parts = line.split(',')
        if len(parts) >= 3:  # Asegúrate de que haya suficientes partes para procesar
            category = parts[0].strip()
            description = parts[1].strip()
            nutrient_data = {
                'Nutrient Data Bank Number': parts[2].strip(),
                'Data.Alpha Carotene': parts[3].strip() if len(parts) > 3 else '',
```

```
                'Data.Ash': parts[4].strip() if len(parts) > 4 else '',
                'Data.Beta Carotene': parts[5].strip() if len(parts) > 5 else '',
                # Agregar más atributos según sea necesario
            }
            triples.append((category, description, nutrient_data))

    return triples

# Generar tríadas a partir del texto extraído
triples = generate_triples_from_text(pdf_text)

# Imprimir las tríadas generadas
for triple in triples:
    print(triple)
```

```
import nltk
from nltk import pos_tag
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')

# Ejemplo de prompt
prompt = "Buscar recetas de pizza"

# Tokenización y POS tagging
tokens = word_tokenize(prompt)
pos_tags = pos_tag(tokens)

# Lematización de sustantivos (NN)
lemmatizer = WordNetLemmatizer()
filtered_tokens = [(lemmatizer.lemmatize(word), pos) for word, pos in pos_tags if pos.startsw

# Construir consulta dinámica basada en sustantivos lematizados
query_terms = [word for word, _ in filtered_tokens]
query = f"SELECT * FROM alimentos WHERE contenido_proteinas > 10 AND {' OR '.join(query_terms

print("Consulta generada:", query)
```

Análisis de Texto Flexible

Este código demuestra cómo utilizar NLTK para procesar un prompt, identificar sustantivos relevantes mediante lematización, y construir una consulta dinámica en SQL.

Es útil para la construcción automatizada de consultas basadas en lenguaje natural, facilitando la búsqueda de información específica en bases de datos que almacenan datos relacionados con recetas de pizzas y empanadas.

Primero, descargamos los recursos necesarios de NLTK.

```
import nltk
from nltk import pos_tag
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
```

```
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
```

Tokenizamos el prompt, realizamos el POS tagging y lematizamos los sustantivos.

```
# Ejemplo de prompt
prompt = "Buscar recetas de pizza"

# Tokenización y POS tagging
tokens = word_tokenize(prompt)
pos_tags = pos_tag(tokens)

# Lematización de sustantivos (NN)
lemmatizer = WordNetLemmatizer()
filtered_tokens = [(lemmatizer.lemmatize(word), pos) for word, pos in pos_tags if pos.star
tswith('NN')]

# Construir consulta dinámica basada en sustantivos lematizados
query_terms = [word for word, _ in filtered_tokens]
query = f"SELECT * FROM alimentos WHERE contenido_proteinas > 10 AND {' OR '.join(query_te
rms)}"

print("Consulta generada:", query)
```

Justificaciones

La tokenización y el etiquetado POS son esenciales para descomponer el texto en sus componentes básicos y asignar etiquetas gramaticales a cada componente. Este proceso permite identificar las partes del discurso, como sustantivos, verbos y adjetivos, lo cual es fundamental para la posterior lematización y construcción de consultas.

Implementación de un Chatbot Experto Utilizando RAG

El sistema está diseñado para manejar consultas en español, utilizando múltiples fuentes de conocimiento, como documentos de texto, datos tabulares y bases de datos de grafos, para proporcionar respuestas precisas y contextualizadas.

Cargar y Preparar Modelos y Datos

```
# Cargar el modelo de spaCy para NLP
nlp = spacy.load('es_core_news_md')


# Configurar la conexión al endpoint SPARQL
sparql = SPARQLWrapper("http://dbpedia.org/sparql")
```

Implementación de Funciones de Búsqueda en Diversas Fuentes

Dividir el proceso de búsqueda en funciones específicas para cada tipo de fuente (texto, tablas y grafos) permite modularizar el código, haciéndolo más legible y fácil de mantener. Además, asegura que el chatbot pueda acceder a la información de manera optimizada según el tipo de consulta.

Utiliza embeddings para encontrar el fragmento de texto más relevante.

```
def buscar_en_textos(message):
    # Utilizar embeddings para encontrar el fragmento de texto más relevante
    query_embedding = model.encode([message])[0].tolist()  # Convertir el embedding a lista

    # Consultar ChromaDB
    results = collection.query(query_embeddings=[query_embedding])

    # Procesar los resultados para encontrar el fragmento más relevante
    if results:
        # Asumimos que results es un diccionario con las claves 'documents' y 'scores'
        documents = results.get('documents', [])
        if documents:
            return documents[0]  # Devuelve el primer documento (ajustar según sea necesario)

    return None
```

Realiza consultas SPARQL para recuperar información relevante.

```
def consultar_grafos(message):
    # Definir la consulta SPARQL dinámica
    query = f"""
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        SELECT ?s ?o
        WHERE {{
            ?s ?p ?o .
            FILTER(CONTAINS(STR(?o), "{message}"))
        }}
    """
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()

    if results["results"]["bindings"]:
        return results["results"]["bindings"]
    return None
```

Procesamiento del Mensaje del Usuario

Procesar el mensaje del usuario permite que el chatbot determine la fuente de datos adecuada para proporcionar la respuesta más precisa posible. Este enfoque garantiza que las consultas sean dinámicas y personalizadas.

```
def process_message(message):
    # Buscar en textos
    resultado_textos = buscar_en_textos(message)
    if resultado_textos:
        return f"Encontré la siguiente información: {resultado_textos}"

    # Buscar en datos tabulares
    resultado_tablas = buscar_en_tablas(message)
    if resultado_tablas:
        return f"Encontré estos datos en las tablas: {resultado_tablas}"

    # Consultar en la base de datos de grafos
    resultado_grafos = consultar_grafos(message)
    if resultado_grafos:
        return f"Encontré esta información en la base de datos de grafos: {resultado_grafos}"
```

```
    return "No encontré información relevante. ¿Puedes especificar un poco más?"
```

Diálogo Interactivo con el Chatbot

Permitir una interacción continua con el usuario asegura que el chatbot pueda manejar múltiples consultas y proporcionar asistencia en tiempo real.

```
# Diálogo interactivo con el bot
print("¡Hola! Soy el chatbot experto. ¿Cómo puedo ayudarte hoy?")
while True:
    user_input = input("Tú: ")
    if user_input.lower() == 'salir':
        print("Chatbot: ¡Gracias por tu consulta! ¡Adiós!")
        break

    bot_response = process_message(user_input)
    print("Chatbot:", bot_response)
```

Justificación:

- `process_message` coordina la búsqueda en las tres fuentes de datos y selecciona la más relevante. Esto garantiza que el chatbot pueda proporcionar respuestas basadas en una amplia variedad de fuentes de información.
- El bucle de diálogo interactivo permite al usuario interactuar con el chatbot de manera continua, ofreciendo respuestas hasta que el usuario decida finalizar la conversación.

Conclusiones

En este trabajo práctico, se desarrolló un chatbot experto que integra múltiples fuentes de datos para proporcionar respuestas precisas y relevantes. La combinación de modelos de embeddings, técnicas de NLP, clasificación LLM y consultas SPARQL permite al chatbot manejar una variedad de consultas en lenguaje natural y ofrecer respuestas basadas en textos, datos tabulares y grafos.

La implementación de la solución demuestra la capacidad de los modelos de lenguaje y las bases de datos modernas para trabajar juntos en un sistema coherente y efectivo. El enfoque flexible y adaptable adoptado asegura que el chatbot pueda adaptarse a diferentes tipos de consultas y proporcionar información precisa y útil.

Enlaces a Modelos y Librerías Utilizados

- **Sentence Transformers**: https://www.sbert.net/
- **LangChain**: https://github.com/langchain/langchain
- **ChromaDB**: https://docs.trychroma.com/
- **Transformers (Hugging Face)**: https://huggingface.co/transformers/
- **spaCy**: https://spacy.io/
- https://github.com/pantulis/recetario/blob/master/db/recipes.csv
- "https://www.recetasmexicanas.org/productos/El-Recetario-Diana-Baker.pdf"
- "https://www.consaboramexico.mx/recetario.pdf"
- "https://www.hospitalitaliano.org.ar/multimedia/archivos/repositorio/3/recursos/1947_Volvamos%20a%20las%20rec

Ejercicio 2

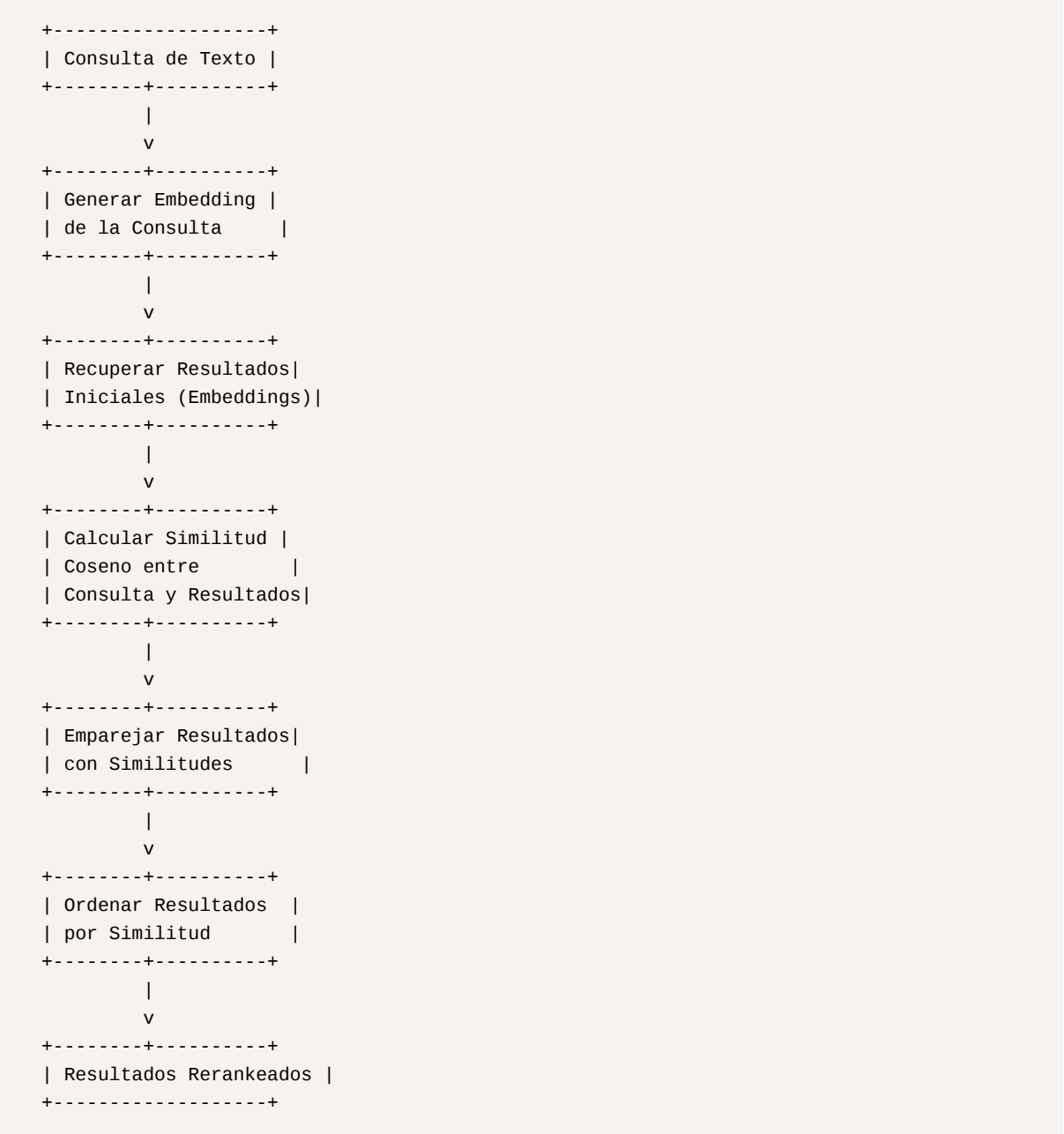
Explicación del Concepto de Rerank

Rerank es una técnica utilizada en sistemas de recuperación de información para mejorar la relevancia de los resultados obtenidos. Después de realizar una búsqueda inicial, los resultados se vuelven a clasificar basándose en

criterios adicionales, como la similitud semántica, la relevancia contextual o el feedback del usuario. Esta técnica puede mejorar significativamente la precisión y la calidad de los resultados presentados al usuario.

Impacto en el Desempeño de la Aplicación: Al aplicar Rerank, la aplicación puede proporcionar resultados más precisos y relevantes, lo que mejora la experiencia del usuario y la eficiencia del sistema de búsqueda. Aunque puede aumentar el tiempo de procesamiento, los beneficios en términos de relevancia y precisión suelen justificar el costo adicional.

Diagrama: Reranking Usando Similitud Coseno Basada en Embeddings



Descripción del Diagrama:

1. **Consulta de Texto**: El proceso comienza con una consulta de texto del usuario.
2. **Generar Embedding de la Consulta**: La consulta de texto se convierte en un embedding vectorial utilizando un modelo de embeddings, como `SentenceTransformer`.

3. **Recuperar Resultados Iniciales (Embeddings)**: Se recuperan resultados iniciales de una base de datos, donde cada resultado también está representado como un embedding vectorial.
4. **Calcular Similitud Coseno entre Consulta y Resultados**: Se calcula la similitud coseno entre el embedding de la consulta y los embeddings de los resultados iniciales. La similitud coseno mide la similitud entre dos vectores en el espacio vectorial.
5. **Emparejar Resultados con Similitudes**: Cada resultado se empareja con su puntaje de similitud coseno.
6. **Ordenar Resultados por Similitud**: Los resultados se ordenan en función de su similitud con la consulta, de mayor a menor.
7. **Resultados Rerankeados**: Finalmente, se devuelve la lista de resultados rerankeados según la similitud con la consulta.

Este diagrama proporciona una visión clara de cómo la similitud coseno basada en embeddings se utiliza para mejorar la relevancia de los resultados en un sistema de recuperación de información.

Aplicación en el Código

- **Explicación:**

  Aplicaría la técnica de Rerank después de obtener los resultados iniciales de una consulta o búsqueda. Por ejemplo, después de recuperar documentos relevantes de ChromaDB, los resultados se pueden volver a clasificar utilizando un modelo de similitud semántica.

- **Código de Ejemplo para Rerank:**

```
from sklearn.metrics.pairwise import cosine_similarity

def rerank_results(query_embedding, result_embeddings, results):
    # Calcular similitud coseno entre la consulta y los resultados
    similarities = cosine_similarity([query_embedding], result_embeddings)[0]
    # Emparejar cada resultado con su similitud
    result_with_scores = list(zip(results, similarities))
    # Ordenar resultados por similitud en orden descendente
    reranked_results = sorted(result_with_scores, key=lambda x: x[1], reverse=True)
    return [result for result, _ in reranked_results]
```