Desarrollo de un Chatbot Especializado en Procesamiento del Lenguaje

Autor: Julián Francisco Britos

Facultad: Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Curso: Procesamiento del Lenguaje Natural (IA 4.2)

Profesores: Juan Pablo Manson y Alan Geary

Fecha de entrega: 21 de julio de 2024

Introducción

El objetivo es desarrollar un ChatBot especializado en el área de NLP (Natural Language Processing) basado en un modelo de lenguaje a gran escala (LLM) mediante la técnica RAG (Retrieval Augmented Generation) que capaz de comprender y responder a consultas de usuarios sobre temas relacionados con NLP, utilizando diversas fuentes de información. Para lograr esto se utilizaron diferentes técnicas para la obtención, extracción, procesamiento y vectorización del texto.

Desarrollo Detallado

1. **Obtención y Carga de Datos:** El código inicia descargando archivos necesarios desde Google Drive que luego son cargados, incluyendo PDFs con contenido teórico y un archivo CSV que es cargado como dataframe con ejercicios. Además, carga un grafo desde un archivo pickle, que presumiblemente representa relaciones entre conceptos.
    a. **Código:**

```
# Link con archivos sobre NLP
url = 'https://drive.google.com/drive/folders/1x_DXa3qQZtSaSGjXa3ntycesMABwn2eR?usp=sharing'
output_folder = 'Archivos_necesarios'

# Verifica si la carpeta ya existe
if os.path.exists(output_folder):
    print(f"La carpeta '{output_folder}' ya existe. No se descargará nuevamente.")
else:
    # Descarga la carpeta
    gdown.download_folder(url, quiet=True, output=output_folder)
    print(f"La carpeta '{output_folder}' fue descargada con éxito.")

# Carga de PDFs
unidad_1 = '/content/Archivos_necesarios/Unidad 1 - Extracción y Procesamiento de Texto.pdf'
unidad_2 = '/content/Archivos_necesarios/Unidad 2 - Representación Vectorial de Texto.pdf'
unidad_3 = '/content/Archivos_necesarios/Unidad 3 - Procesamiento del Lenguaje.pdf'
unidad_4 = '/content/Archivos_necesarios/Unidad 4 - Arquitecturas de Modelos de Lenguaje.pdf'
unidad_5 = '/content/Archivos_necesarios/Unidad 5 - Almacenamiento y Representación del
Conocimiento.pdf'
unidad_6 = '/content/Archivos_necesarios/Unidad 6 - Chatbots y Sistemas de Diálogo.pdf'
unidad_7 = '/content/Archivos_necesarios/Unidad 7 - Agentes Autónomos y Sistemas Inteligentes -
2024.pdf'
```

```
pdfs = [unidad_1, unidad_2, unidad_3, unidad_4, unidad_5, unidad_6, unidad_7]


# Carga del archivo CSV
ejercicios = pd.read_csv('/content/Archivos_necesarios/ejercicios_nlp_con_embeddings.csv')


# Carga del grafo
with open("/content/Archivos_necesarios/grafo_nlp.pickle", "rb") as f:
    G = pickle.load(f)
```

2. **Preprocesamiento de texto, extracción de PDF, eliminación de stopwords y split de datos con LangChain:** Se extrae el texto de los PDFs, se convierte a minúsculas, se eliminan acentos, caracteres especiales y stopwords (palabras comunes sin significado relevante). Luego, el texto se divide en fragmentos más pequeños utilizando la librería LangChain, lo que facilita su procesamiento posterior.

    a. **Código:**

```
# Descargar stopwords si no están ya descargadas
nltk.download('stopwords', quiet=True)


def eliminar_stopwords(texto):
    # Elimina las stopwords del texto
    stop_words = set(stopwords.words('spanish'))
    palabras = texto.split()
    return ' '.join([word for word in palabras if word.lower() not in stop_words])


def extract_text_from_pdf(pdf_path):
    # Extrae el texto de un archivo PDF
    text = ""
    with open(pdf_path, 'rb') as file:
```

```
        reader = PyPDF2.PdfReader(file)
        for page in reader.pages:
            text += page.extract_text()
    return text

def process_pdf_text(pdf_path):
    # Procesa el texto extraído del PDF
    text = extract_text_from_pdf(pdf_path)

    # Convierte todo el texto a minúsculas para uniformidad
    text = text.lower()

    # Elimina acentos y caracteres diacríticos
    text = unidecode(text)

    # Elimina todos los caracteres que no sean letras, números o espacios
    # Esto ayuda a estandarizar el texto y eliminar puntuación y símbolos
    return re.sub(r'[^a-z0-9\s]', '', text)

def split_text_with_langchain(text, chunk_size=500, chunk_overlap=50):
    # Divide el texto en fragmentos más pequeños
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        length_function=len,
    )
    return text_splitter.split_text(text)

# Crear un diccionario global para almacenar los fragmentos sin stopwords
global fragments_sin_stopwords_dict
fragments_sin_stopwords_dict = {}

# Dividir todos los PDFs y procesa los fragmentos para eliminar stopwords
for i, pdf_path in enumerate(pdfs, 1):
```

```
    processed_text = process_pdf_text(pdf_path)
    fragments = split_text_with_langchain(processed_text)

    fragments_sin_stopwords = [eliminar_stopwords(fragment) for fragment in fragments]

    # Almacenar los fragmentos sin stopwords en el diccionario global
    fragments_sin_stopwords_dict[f'unidad_{i}'] = fragments_sin_stopwords

    print(f"\nFragmentos de la Unidad {i}:")
    for j, fragment in enumerate(fragments_sin_stopwords, 1):
        # Muestra una vista previa de cada fragmento (primeros 50 caracteres)
        print(f"Fragmento {j}: {fragment[:50]}...")

    # Comparación de longitudes
    print(f"\nComparación de longitudes para la Unidad {i}:")
    for j, (frag_con, frag_sin) in enumerate(zip(fragments, fragments_sin_stopwords), 1):
        print(f"Fragmento {j}: Con stopwords: {len(frag_con)} caracteres, Sin stopwords: {len(frag_sin)}
caracteres")
```

b. Salida:

Fragmentos de la Unidad 1:

Fragmento 1: unidad 1 extraccion procesamiento texto 1 extracci...

.....

Fragmento 183: resultado sera…

Comparación de longitudes para la Unidad 1:

Fragmento 1: Con stopwords: 409 caracteres, Sin stopwords: 310 caracteres

.....

Fragmento 183: Con stopwords: 17 caracteres, Sin stopwords: 14 caracteres

Estas salidas se repiten para cada unidad a modo de muestra para una mayor entendibilidad de lo que está sucediendo.

3. **Embeddings:** Se emplea el modelo BERT en español para generar representaciones vectoriales (embeddings) de los fragmentos de texto. Estos embeddings capturan el significado semántico de las palabras y frases, permitiendo comparaciones y búsquedas basadas en similitud.

    a. **Código:**

```
from transformers import BertModel, BertTokenizer
import torch
import numpy as np

# Cargar el modelo BERT en español ya que todos nuestros datos estan en español
modelo_es = BertModel.from_pretrained('dccuchile/bert-base-spanish-wwm-cased')
tokenizador_es = BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-cased')

# Función que obtiene embeddings para cada texto
def obtener_embeddings(fragmentos):
    embeddings = []
    for fragmento in fragmentos:
        tokens = tokenizador_es(fragmento, truncation=True, padding=True, return_tensors='pt',
max_length=512)
        with torch.no_grad():
            outputs = modelo_es(**tokens)
        embedding_vector = outputs.last_hidden_state.mean(dim=1).squeeze()
        embeddings.append(embedding_vector.tolist())
    return embeddings

# Diccionario para almacenar los embeddings
embeddings_dict = {}

# Obtener embeddings para cada unidad en el diccionario fragments_sin_stopwords_dict
for unidad, fragmentos in fragments_sin_stopwords_dict.items():
    embeddings = obtener_embeddings(fragmentos)
```

```
    embeddings_dict[unidad] = embeddings


    print(f"\nEmbeddings de los fragmentos de {unidad}:")
    print(np.array(embeddings).shape)


# Acedemos a la los embeddings de la unidad 1 a modo de ejemplo para ver como queda
if 'unidad_1' in embeddings_dict:
    print("\nEjemplo de embeddings para unidad_1:")
    print(np.array(embeddings_dict['unidad_1']))
```

b. Salida:

```
Embeddings de los fragmentos de unidad_1:
(183, 768)

Embeddings de los fragmentos de unidad_2:
(168, 768)

Embeddings de los fragmentos de unidad_3:
(184, 768)

Embeddings de los fragmentos de unidad_4:
(146, 768)

Embeddings de los fragmentos de unidad_5:
(188, 768)

Embeddings de los fragmentos de unidad_6:
(186, 768)

Embeddings de los fragmentos de unidad_7:
(66, 768)

Ejemplo de embeddings para unidad_1:
```

```
[[-0.0613654  -0.01831055 -0.28445315 ... -0.33445156  0.37146971
  0.27996823]
 [-0.17318337  0.03546029 -0.35713169 ...  0.00659324  0.10719279
  0.34588438]
 [-0.31970716  0.09872929 -0.50264072 ...  0.19529545 -0.03763452
  0.30434799]
 ...
 [ 0.31846687 -0.02400654 -0.10420102 ... -0.24092144 -0.10237711
 -0.02467828]
 [-0.17425872 -0.06212582 -0.39995277 ... -0.17951284 -0.1263079
  0.19860029]
 [ 0.18321879  0.23901694 -0.24267174 ... -0.21993811  0.41398084
 -0.38186234]]
```

Primero vemos que dimensiones tienen los array de los embedding y luego vemos los embedding de la unidad 1 a modo de ejemplo.

4. **ChromaDB:** Los fragmentos de texto, sus embeddings y un número id se almacenan en una base de datos vectorial llamada ChromaDB. Esta base de datos permite realizar búsquedas eficientes de fragmentos similares en función de sus embeddings.

    a. **Código:**

```python
import chromadb
import numpy as np

# Crear el cliente y la colección de Chroma
chroma_client = chromadb.Client()
collection = chroma_client.create_collection(name="teoria")

# Unir todos los fragmentos en una sola lista
todos_los_fragmentos = []
todos_los_embeddings = []
ids = []

for unidad, fragmentos in fragments_sin_stopwords_dict.items():
    todos_los_fragmentos.extend(fragmentos)
    todos_los_embeddings.extend(embeddings_dict[unidad])
```

```
    ids.extend([f'{unidad}_id{i+1}' for i in range(len(fragmentos))])

# Comprobamos la cantidad total de fragmentos
print(f"Número total de fragmentos: {len(todos_los_fragmentos)}")

# Comprobamos que tenemos la misma cantidad de embeddings que de fragmentos
print(f"Número total de embeddings: {len(todos_los_embeddings)}")

# Comprobamos que tenemos la misma cantidad de IDs que de fragmentos
print(f"Número total de IDs: {len(ids)}")

# Convertimos los embeddings a una lista de listas
embeddings_lista = [embedding.tolist() if isinstance(embedding, np.ndarray) else embedding for
embedding in todos_los_embeddings]

# Agregamos los embeddings a chromaDB
collection.add(
    embeddings=embeddings_lista,
    documents=todos_los_fragmentos,
    ids=ids
)

print(f"Se han agregado {len(todos_los_fragmentos)} fragmentos a la colección de ChromaDB.")
```

b. Salida:

```
Número total de fragmentos: 1121
Número total de embeddings: 1121
Número total de IDs: 1121
Se han agregado 1121 fragmentos a la colección de ChromaDB.
```

5. **Modelo de clasificación (PDF, CSV o Grafo):** Se entrena un clasificador Naive Bayes para predecir la categoría (CSV, grafo o ChromaDB) a la que pertenece un prompt (entrada de texto del usuario). Este clasificador ayuda a determinar la fuente de información más adecuada para responder a la pregunta del usuario.

a. Código:

```
from sklearn.feature_extraction.text import CountVectorizer
```

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.model_selection import LeaveOneOut, cross_val_score
from sklearn.metrics import classification_report, make_scorer, accuracy_score
import joblib

# Preparación de datos
X = prompts
y = categorias

# Crear un pipeline
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(max_features=20)),  # Usar menos características
    ('clf', MultinomialNB())  # Naive Bayes, un modelo simple
])

# Configurar la validación cruzada dejando uno fuera
cv = LeaveOneOut()

# Realizar validación cruzada
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='accuracy')

print("Puntuaciones de validación cruzada:", scores)
print("Precisión media: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

# Función para obtener predicciones de validación cruzada
def get_cv_predictions(model, X, y, cv):
    y_pred = []
    for train_index, test_index in cv.split(X):
        X_train, X_test = [X[i] for i in train_index], [X[i] for i in test_index]
        y_train, y_test = [y[i] for i in train_index], [y[i] for i in test_index]
        model.fit(X_train, y_train)
        y_pred.extend(model.predict(X_test))
    return y_pred

# Obtener predicciones de validación cruzada
y_pred = get_cv_predictions(pipeline, X, y, cv)

# Imprimir informe de clasificación
print("\nInforme de clasificación:")
print(classification_report(y, y_pred))

# Entrenar el modelo final con todos los datos
pipeline.fit(X, y)

# Guardar el pipeline para uso futuro
joblib.dump(pipeline, 'pipeline_clasificacion.joblib')

# Función para predecir nuevos prompts
```

```
def predecir_categoria(prompt):
    return pipeline.predict([prompt])[0]

# Vemos un ejemplo para ver como queda
nuevo_prompt = "Dame un ejercicio de codificación de archivos"
categoria_predicha = predecir_categoria(nuevo_prompt)
print(f"\nPara el prompt: '{nuevo_prompt}'")
print(f"La categoría predicha es: {categoria_predicha}")
```

b. Salida:

```
Puntuaciones de validación cruzada: [1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 0. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 1. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 0. 1. 0.
 1. 0. 1. 1. 0. 0. 1. 0.]
Precisión media: 0.78 (+/- 0.84)

Informe de clasificación:
              precision    recall  f1-score   support

    chromadb       0.71      0.74      0.73        27
         csv       0.89      0.86      0.88        29
       grafo       0.71      0.71      0.71        24

    accuracy                           0.78        80
   macro avg       0.77      0.77      0.77        80
weighted avg       0.78      0.78      0.78        80


Para el prompt: 'Dame un ejercicio de codificación de archivos'
La categoría predicha es: csv
```

Los resultados de la validación cruzada indican que el modelo clasifica correctamente la mayoría de las muestras (representadas por 1), con algunos errores ocasionales (representados por 0).

La precisión media de 0.78 (+/- 0.84) refleja un rendimiento sólido, con un 78% de clasificaciones correctas en promedio. Sin embargo, el amplio intervalo de confianza (+/- 0.84) indica una variabilidad significativa en el rendimiento, posiblemente por la limitación de datos de entrenamiento.

El informe de clasificación detallado muestra para la categorías ‘csv’ un accuracy (proporción de predicciones correctas sobre el total de predicciones) de 0.89, mientras que para 'chromadb' y 'grafo' muestras 0.71 y 0.71 respectivamente.

La eficacia del modelo se demuestra al clasificar correctamente el prompt de ejemplo "Dame un ejercicio de codificación de archivos" en la categoría 'csv'.

6. **Clasificador basado en un modelo entrenado con ejemplos y embeddings (Unidad 3):** Con base en la categoría predicha por el clasificador, se busca la respuesta más relevante en la fuente correspondiente. Para encontrar la respuesta más similar, se utiliza la distancia de Levenshtein, que mide la similitud entre cadenas de texto.
    a. **Código:**

```
import numpy as np
from scipy.spatial.distance import cosine
from fuzzywuzzy import fuzz

# Cargar el modelo
pipeline = joblib.load('pipeline_clasificacion.joblib')

# Función para predecir la categoría
def predecir_categoria(prompt):
    return pipeline.predict([prompt])[0]

# Función para calcular la distancia de Levenshtein
```

```python
def levenshtein_distance(s1, s2):
    return 100 - fuzz.ratio(s1, s2)  # Convertimos la similitud a distancia

# Función para encontrar la respuesta más apropiada
def encontrar_respuesta(prompt, categoria):
    if categoria == 'csv':
        # Buscar en el dataframe 'ejercicios'
        distancias = ejercicios['Enunciado'].apply(lambda x: levenshtein_distance(prompt, x))
        indice_mas_cercano = distancias.idxmin()
        return ejercicios.loc[indice_mas_cercano, 'Enunciado']

    elif categoria == 'grafo':
        # Buscar en el grafo G
        distancias = {node: levenshtein_distance(prompt, node) for node in G.nodes()}
        nodo_mas_cercano = min(distancias, key=distancias.get)
        return nodo_mas_cercano

    elif categoria == 'chromadb':
        # Buscar en ChromaDB
        resultados = collection.query(
            query_texts=[prompt],
            n_results=1
        )
        if resultados['documents']:
            return resultados['documents'][0][0]
        else:
            return "No se encontró una respuesta apropiada en ChromaDB."

# Función principal para procesar el prompt del usuario
def procesar_prompt(prompt):
    categoria = predecir_categoria(prompt)
    respuesta = encontrar_respuesta(prompt, categoria)
    return categoria, respuesta
prompt_usuario = "Dame un ejercicio de codificación de archivos"
categoria, respuesta = procesar_prompt(prompt_usuario)

print(f"Prompt del usuario: '{prompt_usuario}'")
print(f"Categoría predicha: {categoria}")
print(f"Respuesta encontrada: {respuesta}")

prompt_usuario = "¿Como podria obtener texto de un archivo word?"
categoria, respuesta = procesar_prompt(prompt_usuario)

print(f"Prompt del usuario: '{prompt_usuario}'")
print(f"Categoría predicha: {categoria}")
print(f"Respuesta encontrada: {respuesta}")
```

b. Salida:

```
Prompt del usuario: 'Dame un ejercicio de codificación de archivos'
Categoría predicha: csv
Respuesta encontrada: Obtenga texto de un archivo word.
Prompt del usuario: '¿Como podria obtener texto de un archivo word?'
Categoría predicha: csv
Respuesta encontrada: Obtenga texto de un archivo word.
```

Podemos observar cómo para el primer prompt el modelo lo clasificó correctamente y además dio una respuesta apropiada, sin embargo, para el segundo prompt, aunque el modelo también lo clasificó como 'csv', esta clasificación no es la más adecuada, la más adecuada sería 'chromadb'. Además, la respuesta proporcionada es idéntica a la del primer prompt, lo cual no es apropiado ni útil para la consulta específica sobre cómo obtener texto de un archivo Word.

Esto sugiere que el modelo tiene limitaciones a la hora de predecir de dónde extraer la información y más importante tiene una gran limitación a la hora de proporcionar una respuesta adecuada, diversas y específicas.

7. **Clasificador basado en LLM (Unidad 6):** Con base en la categoría predicha por el clasificador inicial, este sistema utiliza un modelo de lenguaje de gran escala (LLM), específicamente el modelo Zephyr 7B Beta, para generar respuestas más elaboradas y contextualizadas. El proceso se desarrolla de la siguiente manera:
    a. **Selección de fuente de datos:** Dependiendo de la categoría predicha (CSV, grafo o ChromaDB), el sistema consulta la fuente de datos correspondiente:

i. Para CSV, se utiliza una consulta SQL en la base de datos de ejercicios.
ii. Para el grafo, se buscan nodos y relaciones relevantes.
iii. Para ChromaDB, se realiza una búsqueda semántica basada en los embeddings.

b. **Preparación del contexto:** La información relevante obtenida de la fuente de datos se utiliza para crear un contexto informativo que se incluirá en el prompt para el LLM.
c. **Generación del prompt:** Se utiliza una plantilla de prompt que incluye:
    i. Un mensaje de sistema que define el rol del asistente como experto en NLP.
    ii. El contexto extraído de la fuente de datos relevante.
    iii. La pregunta del usuario.
d. **Consulta al LLM:** El prompt preparado se envía al modelo Zephyr 7B Beta a través de la API de Hugging Face. Se configuran parámetros como la temperatura y el número máximo de tokens para controlar la generación de texto.

    Los parámetros utilizados en la consulta al LLM (Zephyr 7B Beta) son:
    i. max_new_tokens: 150 - Limita la longitud de la respuesta generada.
    ii. temperature: 0.7 - Controla la aleatoriedad de las respuestas. Un valor más alto (cerca de 1) produce respuestas más

creativas, mientras que un valor más bajo (cerca de 0) genera respuestas más deterministas.

iii. top_k: 50 - Limita la selección de las siguientes palabras a las 50 más probables.

iv. top_p: 0.95 - Selecciona las palabras más probables cuya probabilidad acumulada no supere 0.95.

e. **Generación de respuesta:** El LLM procesa el prompt y genera una respuesta basada en el contexto proporcionado y la pregunta del usuario. Esta respuesta combina la información específica del contexto con el conocimiento general del modelo, proporcionando una explicación más completa y coherente.

f. **Post-procesamiento:** La respuesta generada se extrae del texto completo devuelto por el LLM y se formatea para su presentación al usuario.

g. **Métricas de rendimiento:** Se calculan métricas como el tiempo de respuesta y la cantidad de tokens generados, lo que permite evaluar y optimizar el rendimiento del sistema.

h. Código:

```
import pandas as pd
import networkx as nx
import chromadb
import requests
from decouple import config
import sqlite3
from time import time
from google.colab import userdata

# Función para consultar el grafo
def query_graph(prompt):
    relevant_nodes = []
    for node in G.nodes():
```

```
        if any(keyword in node.lower() for keyword in prompt.lower().split()):
            relevant_nodes.append(node)
            relevant_nodes.extend(list(G.neighbors(node)))

    context = ""
    for node in set(relevant_nodes):
        context += f"Concepto: {node}\n"
        context += f"Definición: {G.nodes[node].get('definicion', 'No disponible')}\n"
        for neighbor in G.neighbors(node):
            edge_data = G.get_edge_data(node, neighbor)
            context += f"- {edge_data['relacion']} {neighbor}\n"
        context += "\n"

    return context

# Función para consultar la base de datos tabular
def query_tabular(prompt):
    conn = sqlite3.connect(':memory:')
    ejercicios.to_sql('ejercicios', conn, index=False)

    cursor = conn.cursor()
    query = f"""
    SELECT Unidad, Tema, Tipo_Ejercicio, Enunciado, Dificultad
    FROM ejercicios
    WHERE Enunciado LIKE '%{prompt}%'
    LIMIT 5
    """

    results = cursor.execute(query).fetchall()
    conn.close()

    context = ""
    for row in results:
        context += f"Unidad: {row[0]}, Tema: {row[1]}, Tipo: {row[2]}, Dificultad: {row[4]}\n"
        context += f"Enunciado: {row[3]}\n\n"

    return context

# Función para consultar ChromaDB
def query_chromadb(prompt):
    results = collection.query(
        query_texts=[prompt],
        n_results=3
    )

    context = ""
    for doc in results['documents'][0]:
        context += doc + "\n\n"
```

```
    return context

def zephyr_chat_template(messages):
    prompt = ""
    for message in messages:
        if message["role"] == "system":
            prompt += f"{message['content']}\n"
        elif message["role"] == "user":
            prompt += f"Usuario: {message['content']}\n"
        elif message["role"] == "assistant":
            prompt += f"Asistente: {message['content']}\n"
    prompt += "Asistente: "
    return prompt

def prepare_prompt(query_str: str, context_str: str = ""):
    text_qa_prompt_tmpl = (
        "Información de contexto:\n"
        "{context_str}\n"
        "Basándote en la información de contexto proporcionada, responde la siguiente pregunta en
español:\n"
        "Pregunta: {query_str}\n"
    )
    messages = [
        {"role": "system", "content": "Eres un asistente experto en Procesamiento del Lenguaje Natural.
Responde siempre en español de manera útil, veraz y basada en datos."},
        {"role": "user", "content": text_qa_prompt_tmpl.format(context_str=context_str,
query_str=query_str)}
    ]

    return zephyr_chat_template(messages)

def generate_response(prompt, max_new_tokens=150):
    api_key = config('HF_TOKEN', default=userdata.get('HF_TOKEN'))
    api_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-beta"
    headers = {"Authorization": f"Bearer {api_key}"}
    data = {
        "inputs": prompt,
        "parameters": {
            "max_new_tokens": max_new_tokens,
            "temperature": 0.7,
            "top_k": 50,
            "top_p": 0.95
        }
    }

    response = requests.post(api_url, headers=headers, json=data)
    return response.json()[0]['generated_text'].split("Asistente: ")[-1].strip()

def chatbot(user_prompt):
```

```
    categoria = predecir_categoria(user_prompt)

    if categoria == "csv":
        context = query_tabular(user_prompt)
    elif categoria == "grafo":
        context = query_graph(user_prompt)
    else:
        context = query_chromadb(user_prompt)

    full_prompt = prepare_prompt(user_prompt, context)
    response = generate_response(full_prompt)

    return f"Prompt del usuario: {user_prompt}\n\n" \
           f"Base de datos utilizada: {categoria}\n\n" \
           f"Respuesta: {response}"

def calculate_metrics(start_time, end_time, user_prompt, response):
    response_time = end_time - start_time
    token_count = len(response.split())

    return {
        "tiempo_respuesta": response_time,
        "cantidad_tokens": token_count,
    }

if __name__ == "__main__":
    queries = [
        'Dame un ejercicio de codificación de archivos',
        '¿Cómo podría obtener texto de un archivo Word utilizando Python?',
        '¿Podrías explicarme el concepto de RAG (Retrieval Augmented Generation)?',
        '¿Que es NLP?'
    ]

    for query in queries:
        start_time = time()
        result = chatbot(query)
        end_time = time()

        print(result)
        print("\nMétricas:")
        metrics = calculate_metrics(start_time, end_time, query, result)
        for key, value in metrics.items():
            print(f"{key}: {value}")
        print("\n" + "="*50 + "\n")
```

i. Salida:

```
Prompt del usuario: Dame un ejercicio de codificación de archivos
```

Base de datos utilizada: csv

Respuesta: ¡Por supuesto! Aquí tienes un ejemplo de cómo códigar un archivo en Python utilizando la librería built-in `zipfile`:

```python
import zipfile
# Crear un objeto ZipFile y abrir el archivo destino en escritura
archivo_zip = zipfile.ZipFile('archivo.zip', 'w')
# Añadir los archivos que deseas comprimir al archivo ZIP
archivo_zip.write('archivo1.txt')
archivo_zip.write('archivo2.txt')

# Cerrar el
```

Métricas:

tiempo_respuesta: 1.905933141708374

cantidad_tokens: 68

==========

Prompt del usuario: ¿Cómo podría obtener texto de un archivo Word utilizando Python?

Base de datos utilizada: csv

Respuesta: Para obtener el texto de un archivo Word utilizando Python, puedes utilizar la librería PyWin32 o pypiwin32. Primero, debes instalar la librería en tu sistema utilizando pip:

```
pip install PyWin32
```

O en caso de utilizar pypiwin32:

```
pip install pypiwin32
```

Después, puedes utilizar el siguiente código Python:

```python
import win32com.client
# Crea un objeto Word
word = win3
```

Métricas:

tiempo_respuesta: 1.5155980587005615

cantidad_tokens: 80

Podemos observar cómo para el primer prompt el modelo lo clasificó correctamente y además dio una respuesta más apropiada y detallada que la dada por el otro modelo ya que esta cuenta con la presencia de código.

Para el segundo prompt, aunque el modelo lo clasificó como 'csv' (lo cual no es la categoría más adecuada para esta pregunta), la respuesta proporcionada es sorprendentemente apropiada y útil. La respuesta ofrece instrucciones detalladas sobre cómo obtener texto de un archivo Word utilizando Python, incluyendo la instalación de la librería necesaria (PyWin32 o pypiwin32) y un fragmento de código para realizar la tarea.

Este resultado demuestra que, a pesar de la clasificación incorrecta, el modelo basado en LLM (Unidad 6) es capaz de generar respuestas relevantes y específicas para cada pregunta. Esto sugiere que el sistema tiene una buena capacidad para entender el contexto de las preguntas y proporcionar información útil, incluso cuando la categorización inicial no es precisa.

8. **Rerank:** El Rerank es una técnica fundamental en los sistemas RAG (Retrieval Augmented Generation), ya que permite mejorar la calidad y relevancia de los resultados finales. Esta técnica se encarga de reordenar los resultados obtenidos en la primera búsqueda en la base de datos, colocando los más relevantes en las primeras posiciones.

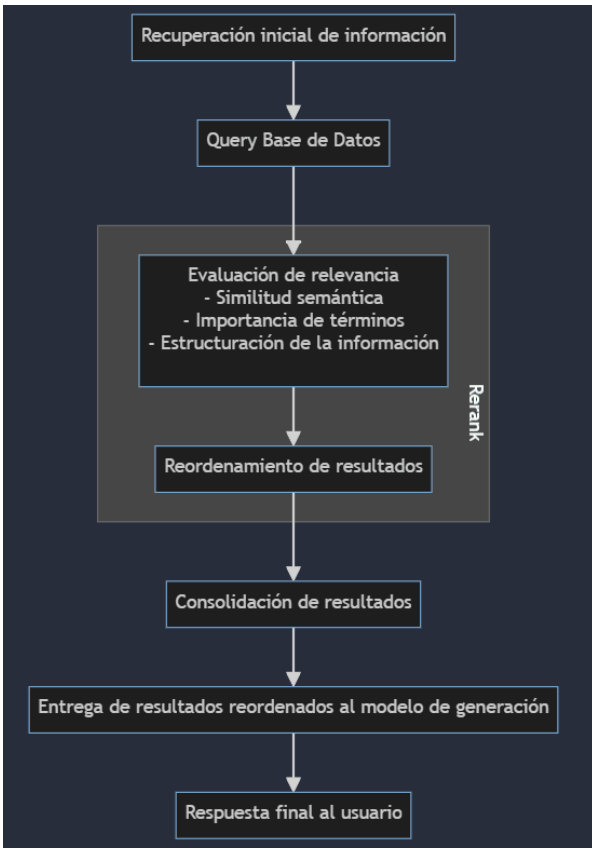
    a. **Beneficios de Rerank**

        i. Mayor precisión: Al reordenar los resultados recuperados, Rerank asegura que la información más relevante se utilice para generar la respuesta, mejorando así la precisión y la calidad de la misma.

        ii. Respuestas más contextualizadas: Al considerar la relevancia de los resultados en relación con la consulta específica, Rerank permite generar respuestas que se ajustan mejor al contexto de la pregunta del usuario.

        iii. Experiencia de usuario mejorada: Las respuestas más precisas y contextualizadas contribuyen a una experiencia de usuario más satisfactoria, ya que las consultas se abordan de manera más efectiva y claras.

b. Diagrama



Recuperación inicial de información: prompt del usuario y su correspondiente vectorización

Query Base de Datos: la consulta del usuario se utiliza para recuperar información relevante de la base de datos.

Evaluación de relevancia: Aquí se realiza una evaluación de la relevancia de los resultados obtenidos. Esta evaluación se basa en diversos factores como:

i. Similitud semántica: Se evalúa qué tan cercanos semánticamente son los resultados a la consulta del usuario.

ii. Importancia de términos: Se analiza la importancia de los términos presentes en los resultados.
iii. Estructuración de la información: Se revisa la forma en que la información está organizada en los resultados.

Estos tres aspectos no son obligatorios, sino que pueden ser utilizados de forma flexible, dependiendo de los requisitos y el enfoque del sistema de pregunta-respuesta. Lo importante es que la evaluación de relevancia logre identificar y priorizar las respuestas más relevantes y útiles para el usuario.

Reordenamiento de resultados: Con base en la evaluación de relevancia, se reordena la lista de resultados para presentar los más relevantes primero.

Consolidación de resultados: En esta etapa, los resultados reordenados se combinan y se preparan para ser entregados al modelo de generación.

Entrega de resultados reordenados al modelo de generación: Los resultados consolidados se envían al modelo de generación, que se encargará de producir la respuesta final para el usuario.

Respuesta final al usuario: Finalmente, la respuesta generada se entrega al usuario como resultado de su consulta.

c. Implementación.

En mi código, la técnica de Rerank se implementaría después de la fase de recuperación inicial de información, pero antes de suministrar los resultados al modelo de generación. En concreto, esta optimización se aplicaría sobre los datos obtenidos de cualquiera de mis tres funciones de consulta principales: query_chromadb, query_graph y query_tabular, refinando así la información que estas funciones proporcionan antes de su utilización en la generación de la respuesta final.

Conclusión

El enfoque basado en LLM de la Unidad 6 mostró una mejora significativa sobre el modelo basado en la Unidad 3, generando respuestas más detalladas, apropiadas e incluso cuando la categorización inicial no es precisa.

Rerank es un componente esencial en la arquitectura RAG, ya que refina el proceso de recuperación de información y garantiza que los resultados más relevantes se utilicen para generar respuestas de alta calidad. Al integrar Rerank en las funciones de consulta, se mejora la precisión, la contextualización y la experiencia general del usuario.

Bibliografía

- spacy: https://spacy.io/
- PyPDF2: https://pythonhosted.org/PyPDF2/
- serpapi: https://serpapi.com/
- chromadb: https://www.trychroma.com/
- networkx: https://networkx.org/
- unidecode: https://pypi.org/project/Unidecode/
- langchain: https://www.langchain.com/
- fuzzywuzzy: https://github.com/seatgeek/fuzzywuzzy
- python-docx: https://python-docx.readthedocs.io/
- python-Levenshtein: https://github.com/ztane/python-Levenshtein/
- langchain-community: https://github.com/langchain-ai/langchain
- sentence-transformers: https://www.sbert.net/
- llama-index-embeddings-huggingface: https://github.com/run-llama/llama_index
- pypdf: https://pypdf.readthedocs.io/
- python-decouple: https://github.com/henriquebastos/python-decouple
- llm-templates: https://github.com/simonw/llm-templates
- llama-index-readers-file: https://github.com/run-llama/llama_index
- Fuente de datos: https://campusv.fceia.unr.edu.ar/course/view.php?id=503
- Retrieval Augmented Generation (RAG) | Prompt Engineering Guide: https://www.promptingguide.ai/techniques/rag
- Retrieval Augmented Generation (RAG) for LLMs | Prompt Engineering Guide: https://www.promptingguide.ai/research/rag.en#generation
- Rerank | Langchain: https://js.langchain.com/v0.2/docs/integrations/document_compressors/cohere_rerank

- Rerank - Optimize Your Search With One Line of Code | Cohere: https://cohere.com/rerank
- A Hands-on Guide to Enhance RAG with Re-Ranking: https://adasci.org/a-hands-on-guide-to-enhance-rag-with-re-ranking/