

TRABAJO PRÁCTICO FINAL

PROCESAMIENTO DE

LENGUAJE NATURAL

EJERCICIO 1

Objetivo

El objetivo del proyecto es crear un chatbot capaz de responder de manera precisa y útil preguntas relacionadas con el Río Paraná, incluyendo consultas sobre alturas del río, distancias entre ciudades a lo largo del río, y otros aspectos relevantes. Para lograr esto, se utilizan diferentes fuentes de datos y técnicas de procesamiento de lenguaje natural.

Fuentes de información

Una vez elegido el tema sobre el cual trabajará el agente, se buscó información en la web para cumplir con los requerimientos pedidos por la cátedra.

En principio, se encontraron diversos artículos académicos que abordaban el tema desde diferentes puntos de vista y aportaban una buena cantidad de datos para generar vectores sobre el dominio a analizar.

Luego, en la página oficial de la provincia, se encontraron diversos archivos .csv que contenían registros sobre las alturas hidrométricas diarias de algunas ciudades sobre las cuales se desarrolla el río. Estos datos corresponden al período 2013-2021.

Para finalizar, restaba investigar sobre base de datos de grafos. Lo cual se convirtió en un inconveniente al sólo disponer de una fuente de datos (WikiData) que no satisfacía las exigencias buscadas. Por ello, se decidió crear un grafo personalizado con información agregada manualmente a un documento tabular.

Desarrollo

Instalaciones e importaciones necesarias

Con los temas vistos en la materia (los cuales teníamos a disposición en el campus) e investigación extra realizada de forma personal, se deciden las librerías necesarias para el desarrollo del agente. Se realiza una primera instalación y luego, con el avance del proyecto, se va adaptando el código para incorporar nuevos paquetes que surgen de la necesidad del mismo.

Esto incluye bibliotecas para:

- Procesamiento de texto: PyPDF, Langchain, NLTK, spaCy
- Manipulación de datos: Pandas
- Aprendizaje automático: scikit-learn, HuggingFace
- Graficación: Matplotlib
- Acceso a recursos externos: ChromaDB, NetworkX, SPARQLWrapper

Base de Datos Vectorial (ChromaDB)

En primera instancia, se va a utilizar la librería PyPDF para extraer la información almacenada en los archivos PDF y guardar la misma en una variable tipo string. Además, se realiza un proceso para omitir saltos de líneas no deseados. Para todo ello se crea la siguiente función:

```

def lectura_pdf(ruta):
    """
    Lee un archivo PDF desde la ruta especificada y devuelve su contenido como un string.

    Parámetros:
    ruta (str): La ruta del archivo PDF a leer.

    Retorna:
    str: El contenido del archivo PDF como un string.

    Ejemplo:
    >>> texto = lectura_pdf('documento.pdf')
    >>> print(texto)
    Contenido del archivo PDF...
    """

    # Abre el archivo en modo binario de lectura ('rb')
    with open(ruta, 'rb') as archivo:
        # Crea un objeto PdfFileReader
        lector = PyPDF2.PdfReader(archivo)

        # Inicializa una cadena vacía para almacenar el texto
        text = ''

        # Itera sobre todas las páginas del PDF
        for i in range(len(lector.pages)):
            # Obtiene la página y extrae el contenido
            contenido = lector.pages[i].extract_text()

            # Reemplaza saltos de línea simples con espacios y conserva solo dobles
            cleaned_text = "\n\n".join(para.strip() for para in contenido.split('\n\n'))

            # Agrega el texto limpio al texto acumulado
            text += cleaned_text

    return text

```

Luego, se accede a la carpeta almacenada en Google Drive donde se encuentran los documentos de texto y se crea un diccionario donde se utiliza el nombre de cada archivo como clave y el contenido de los mismos como valor. A continuación, se comprueba que el proceso se haya realizado correctamente imprimiendo uno de los documentos añadidos.

```

# Ruta de la carpeta en Google Drive donde se encuentran los archivos PDF
carpeta_drive = '/content/gdrive/MyDrive/tp_final_nlp/documentos'

# Listar todos los archivos en la carpeta
archivos_en_carpeta = os.listdir(carpeta_drive)

# Guardar textos en un diccionario
textos = {}
for archivo in archivos_en_carpeta:
    # Obtener el nombre del archivo sin la extensión
    nombre_archivo = os.path.splitext(archivo)[0]
    # Construir la ruta completa del archivo
    ruta_archivo = os.path.join(carpeta_drive, archivo)
    # Leer el texto del archivo PDF y guardarlo en el diccionario
    textos[nombre_archivo] = lectura_pdf(ruta_archivo)

# Se comprueba la carga correcta de los textos en el diccionario
print(textos['Caracterizacion_hidrodinamica_aguas_rio_Parana'])

```

Una vez que tenemos los textos a disposición, lo más conveniente es realizar una limpieza de los mismos (eliminación de caracteres no alfabéticos, conversión a minúsculas y eliminación de stopwords) para lograr un mejor resultado al momento de manipularlos con los modelos de lenguaje preentrenados. Por el mismo motivo, se recomienda fragmentar los textos con algún tamaño determinado y para ello se utiliza la librería Langchain.

Aquí se muestran las funciones creadas para ello y una prueba del correcto funcionamiento de las mismas:

```
# Descargar recursos necesarios para NLTK
nltk.download('stopwords')

def process_text(texto):
    """
    Limpia un texto aplicando varias técnicas de procesamiento.

    Parámetros:
    texto (str): El texto que se va a limpiar y procesar.

    Retorna:
    str: El texto procesado después de eliminar caracteres no alfabéticos, convertir a minúsculas
    y eliminar palabras vacías (stop words).

    Ejemplo:
    >>> texto = "Este es un ejemplo de texto que será procesado. ¡Hola Mundo!"
    >>> texto_procesado = process_text(texto)
    >>> print(texto_procesado)
    ejemplo texto procesado hola mundo
    """

    # Eliminar caracteres no alfabéticos y convertir a minúsculas
    texto_limpio = re.sub(r'[^a-zA-ZáéíóúñÑÁÉÍÓÚÜÑñ\s]', ' ', texto.lower())

    # Eliminación de stop words
    stop_words = set(stopwords.words('spanish'))
    palabras = [palabra for palabra in re.split(r'\s+|\n', texto_limpio) if palabra not in stop_words]

    # Reconstruir el texto preprocesado
    texto_preprocesado = ' '.join(palabras)

    return texto_preprocesado
```

```
def dividir_texto(texto):
    """
    Divide un texto en fragmentos utilizando RecursiveCharacterTextSplitter.

    Parámetros:
    texto (str): El texto que se va a dividir en fragmentos.

    Retorna:
    list: Una lista de strings, donde cada string representa un fragmento del texto.

    Ejemplo:
    >>> texto = "Este es un ejemplo de texto que será dividido en fragmentos."
    >>> fragmentos = dividir_texto(texto)
    >>> print(fragmentos)
    ['Este es un ejemplo', 'de texto que será', 'dividido en fragmentos.']
    """

    # Utilizar Langchain para dividir el texto en oraciones
    splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
    fragmentos = splitter.split_text(texto)
    return fragmentos
```

```
# Se carga en una variable uno de los valores del diccionario para la prueba
original_text = textos['caracterizacion_hidrodinamica_aguas_rio_Parana']
# Fragmentación del texto
chunks = dividir_texto(original_text)
# Limpieza del texto
cleaned_chunks = [process_text(chunk) for chunk in chunks]
# Resultado final
print(cleaned_chunks)
```

['ix congreso argentino ingenieria portuaria buenos aires 5 7 septiembre 2016 ix congreso argentino ingeniería portuaria paper 18 página 1 8 paper']

Para finalizar con el procesamiento de los documentos de textos obtenidos, se van a almacenar en una base de datos ChromaDB. La cual se encarga automáticamente de la realización de los embeddings para vectorizar el texto.

Para ello, se realiza la configuración básica especificando la ruta de almacenamiento, el nombre y el modelo de embedding a utilizar:

```
# Ruta donde se almacenarán los datos de ChromaDB
CHROMA_DATA_PATH = "chroma_data/"

# Modelo de incrustación utilizado para la extracción de características
EMBED_MODEL = "all-MiniLM-L6-v2"

# Nombre de la colección en ChromaDB
COLLECTION_NAME = "rio_parana"

# Crear un cliente persistente de ChromaDB apuntando a la ruta especificada
client = chromadb.PersistentClient(path=CHROMA_DATA_PATH)

# Función de incrustación que utiliza un modelo de Sentence Transformer
embedding_func = embedding_functions.SentenceTransformerEmbeddingFunction(model_name=EMBED_MODEL)

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret 'HF_TOKEN' does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
modules.json: 100% | 349/349 [00:00<00:00, 12.3kB/s]
config_sentence_transformers.json: 100% | 116/116 [00:00<00:00, 3.38kB/s]
README.md: 100% | 10.7k/10.7k [00:00<00:00, 424kB/s]
sentence_bert_config.json: 100% | 53.0/53.0 [00:00<00:00, 1.07kB/s]
config.json: 100% | 612/612 [00:00<00:00, 21.2kB/s]
pytorch_model.bin: 100% | 90.9M/90.9M [00:00<00:00, 106MB/s]
/usr/local/lib/python3.10/dist-packages/torch/_utils.py:831: UserWarning: TypedStorage is deprecated. It will be removed
  return self.fget._get__(instance, owner)()
tokenizer_config.json: 100% | 350/350 [00:00<00:00, 9.22kB/s]
vocab.txt: 100% | 232k/232k [00:00<00:00, 3.14MB/s]
```

Una vez hecha la configuración inicial, se crea la base de datos ChromaDB y se insertan los documentos previo procesamiento de limpieza y segmentación de los mismos:

```
# Creación de la base de datos ChromaDB
collection = client.create_collection(name=COLLECTION_NAME,
                                       embedding_function=embedding_func,
                                       metadata={"hnsw:space": "cosine"})

# Función para agregar documentos a la base de datos ChromaDB
def add_chroma_document(document, id, source):
    collection.add(documents=document, ids=id, metadatas=source)

# Iterar sobre el diccionario con los textos de los archivos
for titulo, documento in textos.items():
    # Dividir los documentos
    chunks = dividir_texto(documento)
    # Limpiar el texto
    cleaned_chunks = [process_text(chunk) for chunk in chunks]
    # Inicializar contadores para respetar los ids de la base de datos ChromaDB
    start_id = collection.count() + 1
    finish_id = start_id + len(cleaned_chunks)
    id_list = [str(i) for i in range(start_id, finish_id)]
    # Crear una lista de nombres de fuente diferentes para cada fragmento
    title_list = [titulo] * len(id_list)
    # Crear una lista de diccionarios con el nombre de fuente correspondiente para cada fragmento
    source_dict_list = [{"source": titulo} for _ in range(len(id_list))]
    # Agregar los fragmentos procesados a la base de datos ChromaDB
    add_chroma_document(cleaned_chunks, id_list, source_dict_list)
    print(f"Documento '{titulo}' ingresado en la colección!")

Documento 'tesis_Pereira, María Soledad' ingresado en la colección!
Documento 'MODELACIÓN HIDRODINÁMICA BIDIMENSIONAL DEL RÍO PARANÁ INFERIOR' ingresado en la colección!
Documento 'Caracterización_hidrodinamica_aguas_rio_Parana' ingresado en la colección!
Documento 'Delta del río Paraná y calentamiento global' ingresado en la colección!
Documento 'El_Río_Parana_Diversidad_Biológica_y_Con' ingresado en la colección!
Documento 'Agronegocio_y_crisis_hidráulica_en_la_cuenca_del_río_' ingresado en la colección!
Documento 'Humedales del río Paraná' ingresado en la colección!
Documento 'Particularidades hídricas y morfológicas zonales' ingresado en la colección!
Documento 'LA TRANSFORMACIÓN DE LA RIBERA BONAERENSE DEL PARANÁ INFERIOR. 1880-1930' ingresado e
```

Para comprobar la correcta carga en la base de datos, se muestra la cantidad de documentos y una vista previa de la misma. También, se realiza una consulta (“Información sobre el Río Paraná”) mostrando los 5 mejores resultados y se imprimen títulos, contenido y distancias:

```
# Cantidad de documentos de la colección
collection.count()

2919

# Vista previa de la colección
collection.peek()

# Se realiza una consulta a la colección
query_results = collection.query(query_texts=["Información sobre el Río Paraná"], n_results=5)

# Se imprimen los resultados obtenidos
print(query_results)
print(query_results["metadatas"])
print(query_results["documents"])
print(query_results["distances"])

{'ids': [[1763, 2666, 309, 905, 2051]], 'distances': [[0.20894229412078857, 0.23107624053955078
[[{'source': 'parana-medio-tomo1'}, {'source': 'parana-medio-tomo2'}, {'source': 'tesis_Pereira, María So
[['nacional vías navegables actualidad fuente aporrear información normalmente clave tratamiento tema ello
[[0.20894229412078857, 0.23107624053955078, 0.23280471563339233, 0.2401065230369568, 0.2525647282600403]]
```

Embeddings

Se emplean el modelo "all-MiniLM-L6-v2" de embeddings de SentenceTransformer para representar textos de manera vectorial. Esto se realiza para calcular, de forma posterior, embeddings de textos relacionados con alturas del río, distancias entre ciudades y otros temas relevantes al proyecto.

```
# Cargar el modelo en español
nlp = spacy.load("es_core_news_sm")

# Función que calcula e imprime la similaridad de coseno de dos vectores
def compute_cosine_similarity(u: np.ndarray, v: np.ndarray) -> float:
    return (u @ v) / (np.linalg.norm(u) * np.linalg.norm(v))

# Creación del modelo de embedding
model = SentenceTransformer("all-MiniLM-L6-v2")

# Transformación del texto procesado a vectores
text_embeddings = model.encode(cleaned_chunks)
```

Base de Datos Tabulares

Se recuperan los documentos .csv cargados en Google Drive para almacenarlos en un DataFrame de Pandas y así poder procesarlos correctamente. Se realizan distintos procesos para limpiar, combinar y filtrar los datos del dataframe. Entre ellos, se mantienen solamente las columnas correspondientes a ciudades por las que pasa el Río Paraná y se eliminan las filas donde todos los registros son nulos para así reducir el tiempo de procesamiento.

```
# Lista para almacenar DataFrames
dataframes = []

# Lee cada archivo CSV correspondiente a los años de interés y los agrega a la lista
for year in (2013, 2020):
    file_path = drive_path + f'alturas-cuenca-rio-parana-{year}.csv'
    df = pd.read_csv(file_path)
    dataframes.append(df)

# Combina los DataFrames
combined_df = pd.concat(dataframes, axis=0, ignore_index=True)

# Definir una función lambda para modificar los nombres de las columnas
modificar_nombre = lambda x: x.split(' - ')[-1]

# Renombrar las columnas utilizando el método rename y la función lambda
combined_df = combined_df.rename(columns=modificar_nombre)

# Caso especial por diferencia en nombre de columnas
file_path = drive_path + f'alturas-cuenca-rio-parana-2021.csv'
df = pd.read_csv(file_path)
df = df.rename(columns={'V Constitución': 'Villa Constitución'})

# Combina los DataFrames
df_alturas = pd.concat([combined_df, df])

# Filtrar columnas del Río Paraná
columnas_deseadas = ['Fecha', 'Corrientes', 'Bella Vista', 'Reconquista', 'San Javier',
                      'La Paz', 'Santa Fe', 'Diamante', 'Rosario', 'Villa Constitución']
df_alturas = df_alturas[columnas_deseadas]

# Modificar el formato de fecha para una mejor manipulación posterior
df_alturas['Fecha'] = pd.to_datetime(df_alturas['Fecha']).dt.strftime('%d/%m/%Y')

# Eliminar filas con NaN en todas las columnas excepto "Fecha"
df_alturas = df_alturas.dropna(subset=df_alturas.columns.difference(['Fecha']), how="all")
```

Resultado final del dataframe:

	Fecha	Corrientes	Bella Vista	Reconquista	San Javier	La Paz	Santa Fe	Diamante	Rosario	Villa Constitución
0	01/01/2013	3.08	NaN	2.83	NaN	NaN	3.05	NaN	2.97	NaN
1	02/01/2013	3.02	NaN	2.81	NaN	NaN	3.04	NaN	2.98	NaN
2	03/01/2013	3.0	NaN	2.84	NaN	NaN	2.99	NaN	2.91	NaN
3	04/01/2013	2.98	NaN	2.8	NaN	NaN	2.88	NaN	2.91	NaN
4	05/01/2013	2.86	NaN	2.8	NaN	NaN	2.86	NaN	2.86	NaN

Por último, para una posterior implementación en los modelos de lenguaje, se crea una función para pasar el dataframe creado a formato string:

```
def dataframe_to_string(df):
    """
    Convierte un DataFrame en una representación de texto legible.

    Parámetros:
    df (DataFrame): El DataFrame que se va a convertir en texto.

    Retorna:
    str: Una representación en formato de texto del DataFrame, donde cada fila se convierte
    en una línea de texto con el formato adecuado.

    Ejemplo:
    >>> df = pd.DataFrame({
    ...     "Fecha": ["2022-01-01", "2022-01-02"],
    ...     "Ciudad A": [2.5, None],
    ...     "Ciudad B": [3.1, 2.9]
    ... })
    >>> texto = dataframe_to_string(df)
    >>> print(texto)
    Altura hidrométrica del Río Paraná en la ciudad de Ciudad A el día 2022-01-01: 2.5 metros
    Altura hidrométrica del Río Paraná en la ciudad de Ciudad A el día 2022-01-02: No hay registro
    ...

    # Se inicializa una cadena vacía para almacenar el texto resultante
    result = ''
    # Se itera sobre cada fila y columna del DataFrame
    for index, row in df.iterrows():
        for column in df.columns:
            # Se verifica que la columna actual no sea la de las fechas
            if column != "Fecha":
                # Se obtiene el valor en la celda correspondiente a la fila y columna actuales
                value = row[column]
                # Se construye una cadena de texto con la altura hidrométrica del Río Paraná para la ciudad y fecha actuales,
                # incluyendo el valor si está disponible o indicando que no hay registro si es nulo
                result += (
                    f'Altura hidrométrica del Río Paraná en la ciudad de {column} el día {row["Fecha"]}: {value} metros\n'
                    if pd.notna(value) else
                    f'Altura hidrométrica del Río Paraná en la ciudad de {column} el día {row["Fecha"]}: No hay registro\n'
                )
        # Se agrega una línea en blanco al final de cada iteración de fila para separar las entradas
        result += "\n"
    return result
```

Resultado de la conversión:

```
# Convertir el dataframe de las alturas del río a string
alturas_str = dataframe_to_string(df_alturas)
print(alturas_str)

Altura hidrométrica del Río Paraná en la ciudad de Corrientes el día 26/09/2013: 3.23 metros
Altura hidrométrica del Río Paraná en la ciudad de Bella Vista el día 26/09/2013: 3.14 metros
Altura hidrométrica del Río Paraná en la ciudad de Reconquista el día 26/09/2013: 2.93 metros
Altura hidrométrica del Río Paraná en la ciudad de San Javier el día 26/09/2013: No hay registro
Altura hidrométrica del Río Paraná en la ciudad de La Paz el día 26/09/2013: No hay registro
Altura hidrométrica del Río Paraná en la ciudad de Santa Fe el día 26/09/2013: 2.6 metros
Altura hidrométrica del Río Paraná en la ciudad de Diamante el día 26/09/2013: No hay registro
Altura hidrométrica del Río Paraná en la ciudad de Rosario el día 26/09/2013: 2.58 metros
Altura hidrométrica del Río Paraná en la ciudad de Villa Constitución el día 26/09/2013: No hay registro
```

Para comprobar la correcta realización de los procesos realizados anteriormente, se calcula la similaridad de coseno entre el documento final y algunas palabras relacionadas y no relacionadas al mismo:

```

# Obtener embeddings para la base de datos tabular con el modelo preentrenado
embedding_alturas = model.encode(alturas_str)

# Las siguientes palabras se van a comparar con el documento de alturas
palabras = ['rio', 'crecida', 'barco', 'perro', 'agua', 'Rosario', 'metros']

for palabra in palabras:
    # Obtener el embedding de la palabra a analizar
    embedding_palabra = model.encode([palabra])
    # Asegurarse de que ambos embeddings tengan la misma forma
    documento = embedding_alturas.reshape(1, -1)
    embedding_palabra = embedding_palabra.reshape(1, -1)
    # Calcular la similitud coseno entre el documento y la palabra
    similitud = cosine_similarity(documento, embedding_palabra)[0][0]
    # Imprimir el resultado
    print(f"Similitud coseno entre '{palabra}' y documento alturas: {similitud}")

Similitud coseno entre 'rio' y documento alturas: 0.43361133337020874
Similitud coseno entre 'crecida' y documento alturas: 0.07483838498592377
Similitud coseno entre 'barco' y documento alturas: 0.15313675999641418
Similitud coseno entre 'perro' y documento alturas: 0.2011856585741043
Similitud coseno entre 'agua' y documento alturas: 0.18571601808071136
Similitud coseno entre 'Rosario' y documento alturas: 0.3001590967178345
Similitud coseno entre 'metros' y documento alturas: 0.1754770576953888

```

Base de Datos de Grafos

Se trae el archivo .csv con los datos de las distancias entre ciudades que se encuentran a lo largo del Río Paraná y se crea con ellos un DataFrame de Pandas.

```

# Ruta al archivo que contiene las distancias entre las ciudades
file_path= '/content/gdrive/MyDrive/tp_final_nlp/distancias_ciudades.csv'
# Creación del datafram
df_distancias = pd.read_csv(file_path)

df_distancias

      Ciudad_Inicio Ciudad_Fin Distancia_Ruta
0       Corrientes   Bella Vista        144
1       Corrientes  Reconquista        241
2       Corrientes     San Javier        411
3       Corrientes       La Paz        425
4       Corrientes      Santa Fe        566
...
67  Villa Constitución     San Javier        380
68  Villa Constitución       La Paz        416
69  Villa Constitución      Santa Fe        228
70  Villa Constitución     Diamante        205
71  Villa Constitución      Rosario         60

```

72 rows × 3 columns

Luego, se crea manualmente otro dataframe con información de cada ciudad: Población, Densidad Poblacional, Superficie y Altitud media.

```
# Creación de un nuevo dataframe con datos de las ciudades a trabajar
data = {
    'Ciudad': ['Corrientes', 'Bella Vista', 'Reconquista', 'San Javier',
               'La Paz', 'Santa Fe', 'Diamante', 'Rosario', 'Villa Constitución'],
    'Poblacion': [352646, 29071, 65956, 16449, 24716, 697054, 24094, 1342619, 47903],
    'Densidad_Poblacional': [3918, 17, 136, 6, 628, 1498, 184, 5726, 460],
    'Superficie': [90, 1706, 537, 2284, 119, 268, 104, 179, 103],
    'Altitud_Media': [62, 61, 21, 28, 55, 25, 14, 25, 42]
}

df_demografico = pd.DataFrame(data)

df_demografico
```

	Ciudad	Poblacion	Densidad_Poblacional	Superficie	Altitud_Media	
0	Corrientes	352646	3918	90	62	
1	Bella Vista	29071	17	1706	61	
2	Reconquista	65956	136	537	21	
3	San Javier	16449	6	2284	28	
4	La Paz	24716	628	119	55	
5	Santa Fe	697054	1498	268	25	
6	Diamante	24094	184	104	14	
7	Rosario	1342619	5726	179	25	
8	Villa Constitución	47903	460	103	42	

Estos dos dataframes se van a utilizar para crear una base de datos de grafos de NetworkX donde las ciudades se van a representar como nodos. Luego de la creación de dichos nodos, se establecen las relaciones (distancias entre ciudades) tomadas desde el dataframe “df_distancias”. Por último, se agrega la información del dataframe “df_demografico” a cada nodo.

```
# Crear un grafo dirigido de networkx
G = nx.DiGraph()

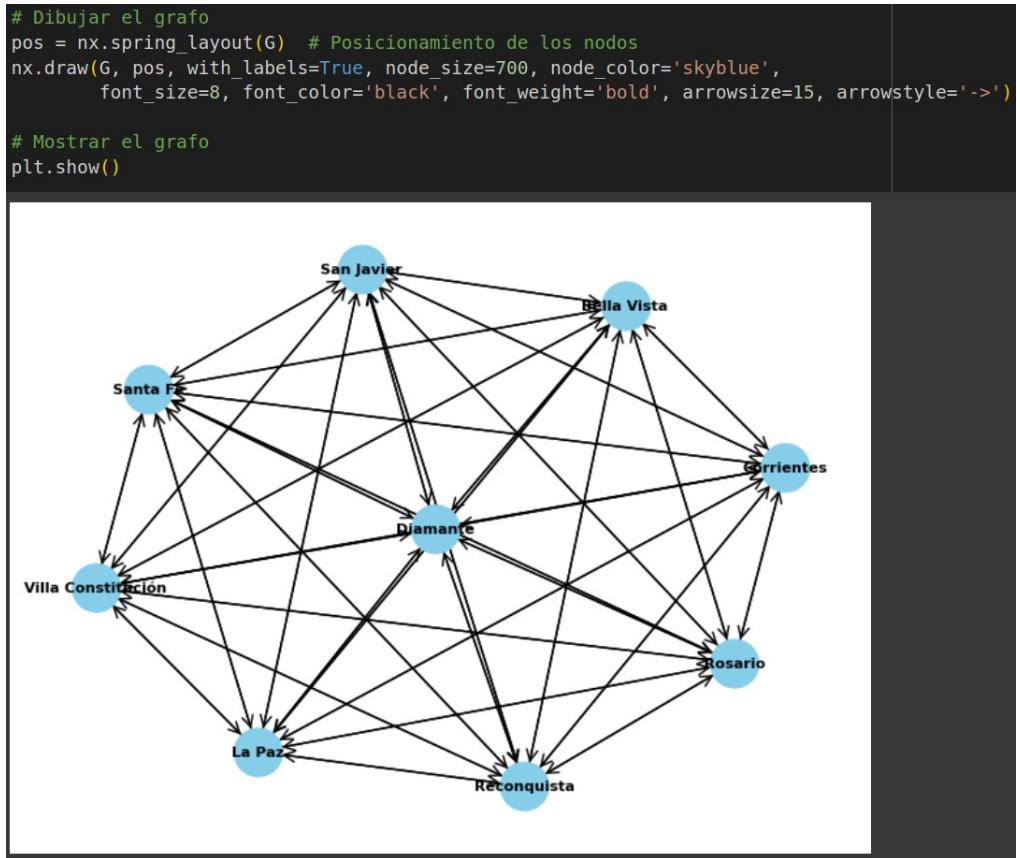
# Ciudades para crear nodos
ciudades = ['Corrientes', 'Bella Vista', 'Reconquista', 'San Javier', 'La Paz',
            'Santa Fe', 'Diamante', 'Rosario', 'Villa Constitución']

# Agregar los nodos al grafo
for ciudad in ciudades:
    G.add_node(ciudad)

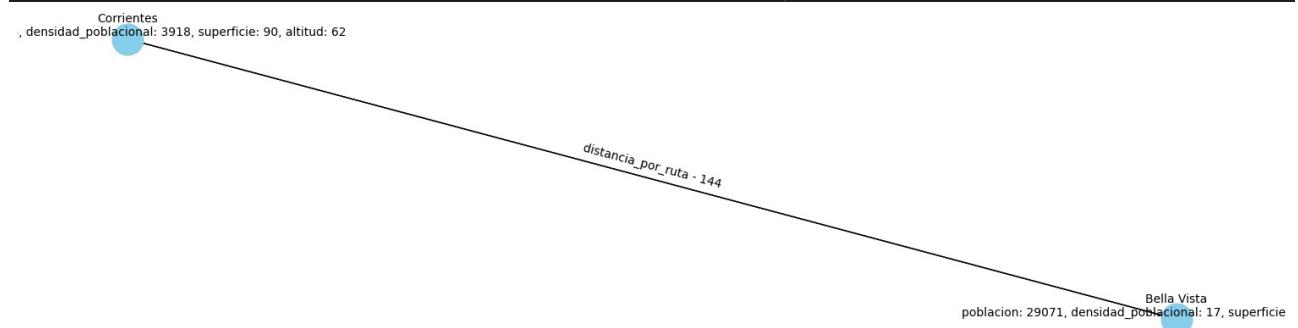
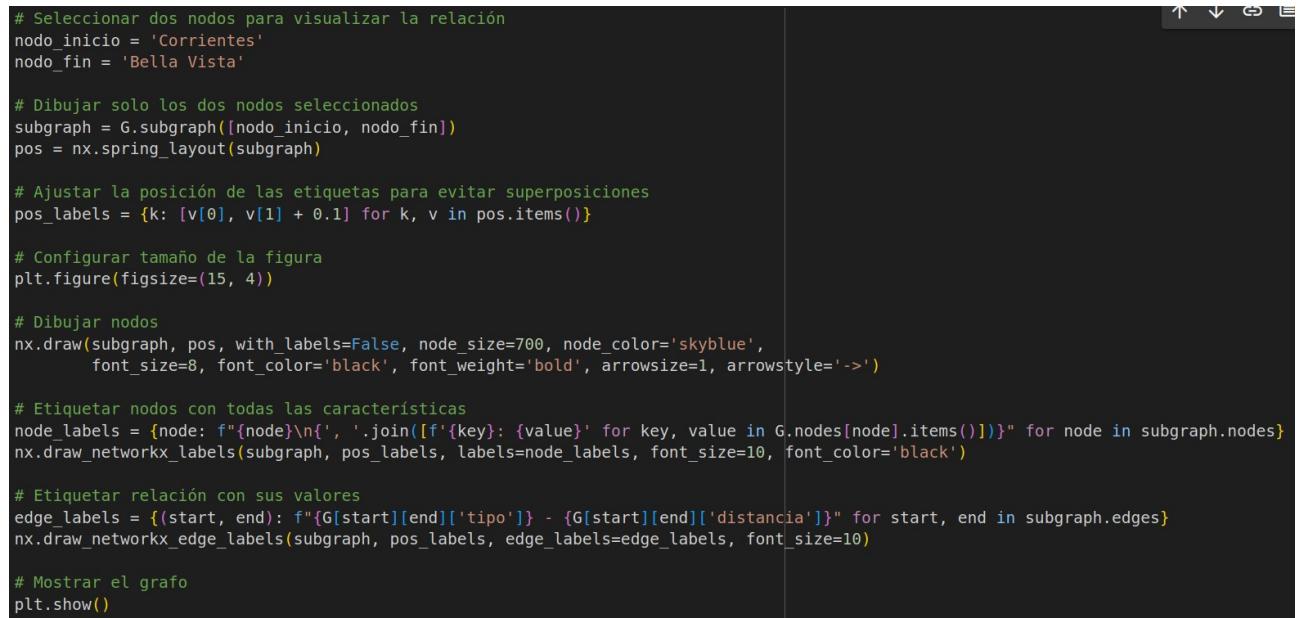
# Crear relaciones entre los nodos desde el DataFrame
for index, row in df_distancias.iterrows():
    # Establecer nodo inicial
    inicio = row['Ciudad_Inicio']
    # Establecer nodo final
    fin = row['Ciudad_Fin']
    # Tomar el valor de distancia que será la relación
    distancia_ruta = row['Distancia_Ruta']
    # Agregar la relación al grafo
    G.add_edge(inicio, fin, tipo='distancia_por_ruta', distancia=distancia_ruta)

# Agregar datos demográficos a cada nodos
for ciudad in ciudades:
    # Se agrega al nodo la cantidad de habitantes de la ciudad
    poblacion = df_demografico[df_demografico['Ciudad'] == ciudad]['Poblacion'].values[0]
    G.nodes[ciudad]['poblacion'] = poblacion
    # Se agrega al nodo la densidad poblacional de la ciudad
    densidad_poblacional = df_demografico[df_demografico['Ciudad'] == ciudad]['Densidad_Poblacional'].values[0]
    G.nodes[ciudad]['densidad_poblacional'] = densidad_poblacional
    # Se agrega al nodo la superficie de la ciudad
    superficie = df_demografico[df_demografico['Ciudad'] == ciudad]['Superficie'].values[0]
    G.nodes[ciudad]['superficie'] = superficie
    # Se agrega al nodo la altitud media de la ciudad
    altitud = df_demografico[df_demografico['Ciudad'] == ciudad]['Altitud_Media'].values[0]
    G.nodes[ciudad]['altitud'] = altitud
```

Se dibuja el grafo, para comprobar su correcta creación, estableciendo distintos parámetros de visualización como el tamaño y color de los nodos, el estilo y tamaño de las flechas de relación, etc.



También se grafica una relación entre nodos (con su valor correspondiente) y se muestra las propiedades de cada uno de ellos:



Por último, para una posterior implementación en los modelos de lenguaje, se crea una función para pasar el grafo creado a formato string:

```
def graph_data_to_string(graph):
    """
    Convierte la información de un grafo en formato de texto.

    Parámetros:
    graph (NetworkX Graph): El grafo del cual se va a extraer la información.

    Retorna:
    str: Una representación en formato de texto de la información del grafo.

    """

    # Inicializa una cadena vacía para almacenar el texto resultante
    result = ''

    # Itera sobre los nodos del grafo
    for node, data in graph.nodes(data=True):
        # Agrega el nombre del nodo (ciudad) al texto resultante
        result += f"Ciudad: {node}\n"
        # Itera sobre los datos asociados al nodo
        for key, value in data.items():
            # Agrega los datos asociados al nodo al texto resultante
            result += f"{key}: {value}\n"
        # Agrega una línea en blanco para separar la información de cada nodo
        result += "\n"

    # Itera sobre las relaciones (aristas) del grafo
    for edge in graph.edges(data=True):
        # Construye una cadena con la distancia por ruta entre las ciudades conectadas por la relación
        result += f"Distancia por ruta entre las ciudades de {edge[0]} y {edge[1]}: "
        # Verifica si hay información sobre la distancia en los datos asociados a la relación
        if 'distancia' in edge[2]:
            # Agrega la distancia al texto resultante
            result += f"{edge[2]['distancia']} kilómetros\n"
        else:
            # Indica que la distancia es desconocida en el texto resultante
            result += "Desconocido kilómetros\n"
        # Agrega una línea en blanco para separar la información de cada relación
        result += "\n"

    return result
```

Resultado de la conversión:

```
# Convertir el grafo de las ciudades a string
graph_str = graph_data_to_string(G)
print(graph_str)

Ciudad: Diamante
poblacion: 24094
densidad_poblacional: 184
superficie: 104
altitud: 14

Ciudad: Rosario
poblacion: 1342619
densidad_poblacional: 5726
superficie: 179
altitud: 25

Ciudad: Villa Constitución
poblacion: 47903
densidad_poblacional: 460
superficie: 103
altitud: 42

Distancia por ruta entre las ciudades de Corrientes y Bella Vista: 144 kilómetros
Distancia por ruta entre las ciudades de Corrientes y Reconquista: 241 kilómetros
Distancia por ruta entre las ciudades de Corrientes y San Javier: 411 kilómetros
Distancia por ruta entre las ciudades de Corrientes y La Paz: 425 kilómetros
Distancia por ruta entre las ciudades de Corrientes y Santa Fe: 566 kilómetros
Distancia por ruta entre las ciudades de Corrientes y Diamante: 632 kilómetros
```

Para comprobar la correcta realización de los procesos realizados anteriormente, se calcula la similaridad de coseno entre el documento final y algunas palabras relacionadas y no relacionadas al mismo:

```
# Obtener embeddings para la base de datos de grafos
embedding_grafo = model.encode(graph_str)

# Las siguientes palabras se van a comparar con el documento de grafo
palabras = ['rio', 'crecida', 'barco', 'perro', 'agua', 'Rosario', 'metros']

for palabra in palabras:
    # Obtener el embedding de la palabra a analizar
    embedding_palabra = model.encode([palabra])
    # Asegurarse de que ambos embeddings tengan la misma forma
    documento = embedding_grafo.reshape(1, -1)
    embedding_palabra = embedding_palabra.reshape(1, -1)
    # Calcular la similitud coseno entre el documento y la palabra
    similitud = cosine_similarity(documento, embedding_palabra)[0][0]
    # Imprimir el resultado
    print(f"Similitud coseno entre '{palabra}' y documento grafo: {similitud}")

Similitud coseno entre 'rio' y documento grafo: 0.3222016394138336
Similitud coseno entre 'crecida' y documento grafo: 0.10861679911613464
Similitud coseno entre 'barco' y documento grafo: 0.22090904414653778
Similitud coseno entre 'perro' y documento grafo: 0.18809348344802856
Similitud coseno entre 'agua' y documento grafo: 0.1820225715637207
Similitud coseno entre 'Rosario' y documento grafo: 0.3478882908821106
Similitud coseno entre 'metros' y documento grafo: 0.11916528642177582
```

Clasificador

Estableciendo una serie de datos de entrenamiento, se busca modelar un clasificador Support Vector Classifier que va a servir para predecir la categoría de las preguntas y seleccionar la fuente de datos más adecuada para proporcionar respuestas.

```
# Datos de entrenamiento
data = [
    {'text': '¿Cuál es la altura actual del Río Paraná en Corrientes?', 'category': 'Altura_Rio'},
    {'text': '¿Cuál fue la altura promedio del año 2022 del Río Paraná en la ciudad de Rosario?', 'category': 'Altura_Rio'},
    {'text': '¿Cuánta distancia por ruta hay entre las ciudades de Corrientes y Diamante?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cómo se monitorean las alturas del Río Paraná?', 'category': 'Otros_Rio'},
    {'text': 'Dime más sobre la navegación en el Río Paraná.', 'category': 'Otros_Rio'},
    {'text': '¿Cuál es la altura actual del Río Paraná en la ciudad de Santa Fe?', 'category': 'Altura_Rio'},
    {'text': '¿Cuánta distancia por ruta hay entre las ciudades de Rosario y Santa Fe?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la altura máxima registrada del Río Paraná en la ciudad de Rosario?', 'category': 'Altura_Rio'},
    {'text': '¿Cuánta distancia por ruta hay entre las ciudades de Bella Vista y Villa Constitución?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Goya y Rosario?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la altura mínima del Río Paraná en la ciudad de Corrientes?', 'category': 'Altura_Rio'},
    {'text': '¿Cuánta distancia por ruta hay entre las ciudades de Posadas y Corrientes?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Qué medidas de seguridad se toman durante las crecidas del río?', 'category': 'Otros_Rio'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Santa Fe y Paraná?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cómo se determinan las alertas de altura del Río Paraná?', 'category': 'Otros_Rio'},
    {'text': '¿Cuál es la altura media del Río Paraná en el mes de agosto?', 'category': 'Altura_Rio'},
    {'text': '¿Cuáles son los efectos económicos de la sedimentación en el Río Paraná?', 'category': 'Otros_Rio'},
    {'text': '¿Cuál es la altura actual del Río Paraná en la ciudad de Rosario?', 'category': 'Altura_Rio'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Rosario y Buenos Aires?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Paraná y Santa Fe?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cómo se determina la profundidad del Río Paraná en diferentes puntos?', 'category': 'Altura_Rio'},
    {'text': '¿Cuál es la altura actual del Río Paraná en el Puerto de Rosario?', 'category': 'Altura_Rio'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Rosario y Santa Fe?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Qué impacto tienen las alturas del Río Paraná en la economía local?', 'category': 'Otros_Rio'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Santa Fe y Paraná?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la altura actual del Río Paraná en la ciudad de Corrientes?', 'category': 'Altura_Rio'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Posadas y Corrientes?', 'category': 'Distancia_Ciudades'},
    {'text': '¿Cuál es la distancia por ruta entre las ciudades de Bella Vista y Villa Constitución?', 'category': 'Distancia_Ciudades'}
]
```

Estos datos cuentan con el texto propiamente que se utiliza como característica y la categoría establecida que se utiliza como etiqueta del modelo. En consecuencia, se realiza dicha división de datos y se crea un pipeline para vectorizar los documentos y clasificarlos.

```
# Dividir datos en características (X) y etiquetas (y)
X = [item['text'] for item in data]
y = [item['category'] for item in data]

# Crear un pipeline con un vectorizador TF-IDF y un clasificador Support Vector Classifier
classifier_pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('classifier', SVC(kernel='sigmoid'))
])

classifier_pipeline
```

```

Pipeline
Pipeline(steps=[('tfidf', TfidfVectorizer()), ('classifier', SVC(kernel='sigmoid'))])
    ↓
    TfidfVectorizer
    TfidfVectorizer()
        ↓
        SVC
        SVC(kernel='sigmoid')
    
```

Luego se entrena el modelo para comprobar su rendimiento. Para ello se divide el dataset en datos de entrenamiento y prueba, se realiza la predicción y se muestra la métrica accuracy obtenida.

```
# Dividir el conjunto de datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=15)

# Entrenar el clasificador
classifier_pipeline.fit(X_train, y_train)

# Predecir en el conjunto de prueba
y_pred = classifier_pipeline.predict(X_test)

# Evaluar el rendimiento
accuracy = accuracy_score(y_test, y_pred)
print("Exactitud del modelo:", accuracy)

Exactitud del modelo: 0.7777777777777778
```

Retriever

Se crea una función que busca la información más relevante, de acuerdo a una consulta, en una base

```
def chroma_similarity(query, collection: chromadb.api.models.Collection.Collection):
    """
    Realiza una búsqueda de similitud de documentos utilizando ChromaDB.

    Parámetros:
    query (str): La consulta de búsqueda para encontrar documentos similares.
    collection (ChromaDB Collection): La colección de ChromaDB en la que se realizará la búsqueda.

    Retorna:
    list: Una lista con los documentos de los 5 mejores resultados de la búsqueda.

    """

    # Realiza una consulta a la colección de ChromaDB utilizando la consulta proporcionada
    query_results = collection.query(query_texts=[query], n_results=5)

    # Obtiene la lista de documentos de los resultados de la consulta
    chroma_list = query_results['documents'][0]

    # Devuelve la lista de los 5 mejores resultados
    return chroma_list
```

De manera similar, se realiza la siguiente función que busca las similaridades en documentos de textos que se pasan como variable string. También se devuelven las mayores 5:

```
def context_similarity(query, context_str, top_n=5):
    """
    Calcula las mayores similitudes de coseno entre una consulta y una lista de strings.

    Parámetros:
    query (str): La consulta para comparar con los strings del contexto.
    context_str (str): El string que contiene los datos de contexto.
    top_n (int): El número de mayores similitudes a devolver (por defecto es 5).

    Retorna:
    list: Una lista con las n mayores similaridades entre la consulta y los strings del contexto.

    """

    # Dividir el texto ingresado con uno o dos saltos de linea
    respuestas = re.split(r'\n{1,2}', context_str)

    # Convertir la pregunta y los datos del DataFrame a un vector TF-IDF
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform([query] + respuestas)

    # Calcular la similaridad de coseno entre la pregunta y los strings
    cosine_similarities = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:]).flatten()

    # Ordenar los índices de los registros por similaridad descendente
    top_indices = cosine_similarities.argsort()[:-1][:-top_n]

    # Obtener las respuestas correspondientes a las mayores similaridades
    top_responses = [respuestas[i].strip() for i in top_indices]

    # Devolver las n mayores similitudes
    return top_responses
```

RAG (Retrieve and Generate)

Se utiliza un modelo de generación de texto preentrenado (Hugging Face) para generar respuestas basadas en el contexto recuperado y la pregunta formulada:

```
def generate_answer(prompt: str, max_new_tokens: int = 768) -> str:
    """
    Genera una respuesta utilizando un modelo de Hugging Face a partir de un prompt dado.

    Parámetros:
    prompt (str): El texto inicial o la pregunta que se proporciona como entrada al modelo.
    max_new_tokens (int): El número máximo de nuevos tokens que el modelo puede generar (por defecto es 768).

    Retorna:
    str: La respuesta generada por el modelo.

    """
    try:
        # URL de la API de Hugging Face para la generación de texto
        api_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-alpha"
        # Se utiliza el token de autorización proporcionado
        headers = {"Authorization": f"Bearer {api_key}"}
        # Datos para enviar en la solicitud POST
        data = {
            "inputs": prompt,
            "parameters": {
                "max_new_tokens": max_new_tokens,
                "temperature": 0.7,
                "top_k": 50,
                "top_p": 0.95
            }
        }
        # Realizamos la solicitud POST a la API de Hugging Face
        response = requests.post(api_url, headers=headers, json=data)
        # Extraer la respuesta generada por el modelo
        respuesta = response.json()[0]["generated_text"][len(prompt):] # Se recorta el prompt de la respuesta generada
        return respuesta # Se devuelve la respuesta generada por el modelo
    except Exception as e:
        # En caso de error, se imprime un mensaje de error junto con la excepción
        print(f"An error occurred: {e}")
        print("Response Text:")
        # Se imprime el texto de la respuesta para facilitar la depuración
        print(response.text)
```

Se crea una función que genera un prompt de entrada con mensajes estructurados con el formato Zephyr:

```

def zephyr_instruct_template(messages, add_generation_prompt=True):
    """
    Crea una plantilla para el prompt de entrada en el formato esperado por el modelo Zephyr.

    Parámetros:
    messages (list): Lista de mensajes, cada uno con un diccionario que contiene un 'role' (user, assistant, system, etc.)
                      y 'content' (el contenido del mensaje).
    add_generation_prompt (bool): Indica si se debe agregar un prompt de generación al final de la plantilla (por defecto es True).

    Retorna:
    str: La plantilla del prompt renderizada con los mensajes proporcionados.

    """

    # Definir la plantilla Jinja
    template_str = "{% for message in messages %}" # Inicia el bucle para cada mensaje
    template_str += "{% if message['role'] == 'user' %}" # Comprueba si el rol del mensaje es 'user'
    template_str += "{{ message['content'] }}</s>\n" # Agrega el contenido del mensaje
    template_str += "{% elif message['role'] == 'assistant' %}" # Comprueba si el rol del mensaje es 'assistant'
    template_str += "{{ message['content'] }}</s>\n" # Agrega el contenido del mensaje
    template_str += "{% elif message['role'] == 'system' %}" # Comprueba si el rol del mensaje es 'system'
    template_str += "{{ message['content'] }}</s>\n" # Agrega el contenido del mensaje
    template_str += "{% else %}" # Si no coincide con ningún rol conocido, asume 'user'
    template_str += "{{ message['content'] }}</s>\n" # Agrega el contenido del mensaje
    template_str += "{% endif %}" # Fin de la condición
    template_str += "{% endfor %}" # Fin del bucle

    # Agregar un prompt de generación si se especifica
    if add_generation_prompt:
        template_str += "\n"

    # Crear un objeto de plantilla con la cadena de plantilla
    template = Template(template_str)

    # Renderizar la plantilla con los mensajes proporcionados
    return template.render(messages=messages, add_generation_prompt=add_generation_prompt)

```

Se crea una función que prepara el prompt que se le va a pasar al modelo de generación de texto. El prompt se construye con los contenidos de texto recuperados del retriever y se aclara que no utilice información de contexto adicional.

```

def prepare_prompt(query_str: str, contexto: list):
    """
    Prepara el prompt para conversar con el agente.

    Parámetros:
    query_str (str): La pregunta que se le hará al agente.
    contexto (list): Una lista de strings que representan el contexto de la conversación.

    Retorna:
    str: El prompt final para la conversación con el agente.

    """

    # Template de la conversación con el agente
    TEXT_QA_PROMPT_TMPL = (
        "La información de contexto es la siguiente:\n"
        "-----\n"
        "{context_str}\n"
        "Dada la información de contexto anterior, y sin utilizar conocimiento previo, responde la siguiente pregunta.\n"
        "Pregunta: {query_str}\n"
        "Respuesta: ")

    # Acumular los contenidos de los strings en una lista
    accumulated_contexts = []
    for i in contexto:
        accumulated_contexts.append(i)

    # Construir el prompt final
    final_context_str = "\n".join(accumulated_contexts)
    final_prompt = zephyr_instruct_template([
        {"role": "system", "content": "Eres un asistente útil que siempre responde en idioma Español con respuestas veraces,"},
        {"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(context_str=final_context_str, query_str=query_str)}])

    return final_prompt

```

Por último, se crea una función que procesa la pregunta ingresada, aplica tanto el clasificador como el retriever y muestra la respuesta generada por el modelo junto con la categoría predicha:

```
def procesar_pregunta(b):
    """
    Procesa la pregunta ingresada por el usuario y muestra la respuesta.

    Parámetros:
    b: Un objeto que representa al botón que desencadena la función.

    """

    # Obtener la pregunta ingresada por el usuario desde el widget de entrada
    pregunta = input_pregunta.value

    # Utilizar el clasificador entrenado para predecir la categoría de la pregunta
    categoria_predicha = classifier_pipeline.predict([pregunta])

    if categoria_predicha == 'Altura_Rio':
        fuente_contexto = 'Base de datos tabular'
        # Calcular la similaridad utilizando la función context_similarity
        list_similarities = context_similarity(pregunta, alturas_str)

    elif categoria_predicha == 'Distancia_Ciudades':
        fuente_contexto = 'Base de datos de grafos'
        # Calcular la similaridad utilizando la función context_similarity
        list_similarities = context_similarity(pregunta, graph_str)

    else:
        fuente_contexto = 'Documentos'
        # Calcular la similaridad utilizando la función chroma_similarity
        list_similarities = chroma_similarity(pregunta, collection)

    # Preparar el prompt utilizando la función prepare_prompt
    final_prompt = prepare_prompt(pregunta, list_similarities)

    # Generar la respuesta utilizando la función generate_answer
    respuesta = generate_answer(final_prompt)

    # Mostrar la conversación en el widget de salida
    salida_conversacion.append_stdout(f"Usuario: {pregunta}\n\nChatbot: {respuesta}\n\n")

    # Mostrar el contexto correspondiente
    salida_conversacion.append_stdout(f"Contexto de la respuesta: {fuente_contexto}\n\n\n")

# Crear widget de entrada para la pregunta
input_pregunta = widgets.Text(description="Pregunta:", value="")

# Crear botón para enviar la pregunta
boton_enviar = widgets.Button(description="Enviar")
boton_enviar.on_click(procesar_pregunta)

# Crear widget de salida para mostrar la conversación
salida_conversacion = widgets.Output()

# Mostrar widgets
display(input_pregunta, boton_enviar, salida_conversacion)
```

Pregunta:

Enviar

Conclusiones

El chatbot del Río Paraná es una herramienta útil para obtener información rápida y precisa sobre diversos aspectos relacionados con el río. Su capacidad para recuperar y generar respuestas basadas en diferentes fuentes de datos lo hace versátil y efectivo en la atención de consultas relacionadas con el tema.

Futuras Mejoras

Para mejorar el chatbot del Río Paraná, se podrían considerar las siguientes mejoras:

- Ampliar y mejorar las fuentes de datos disponibles.
- Refinar los modelos de procesamiento de lenguaje natural para mejorar la precisión de las respuestas.
- Implementar un sistema de retroalimentación para mejorar continuamente la calidad de las respuestas.
- Incorporar capacidades de aprendizaje automático para adaptarse y mejorar con el tiempo.

Complicaciones encontradas en el proyecto

- En cuanto a la utilización de paquetes y librerías, se presentaron problemas con las versiones de los mismos. En diferentes ejecuciones del código, se intercalaban errores que fueron solucionados especificando las versiones estables de algunos librerías.
- Luego, al momento de procesar el texto extraído de los archivos .pdf, hubo que hacer una limpieza exhaustiva ya que se encontraron errores propios de la manipulación con la librería PyPDF. Para ello, se tuvo que eliminar caracteres no correspondientes y ordenar el texto en cuanto a saltos de líneas.
- Otro tema en el que hubo que prestar atención fue en la división del texto con Langchain, ya que fue necesario hacer diversas pruebas para ajustar el tamaño de los fragmentos que este realizaba para lograr un correcto procesamiento posterior en la base de datos ChromaDB.
- Para el requerimiento de utilizar base de datos de grafos se tuvo que investigar sobre el tema en la web, llegando a la conclusión que todavía no se cuenta con demasiadas fuentes de datos. Por lo tanto, se decidió (al tener el tema del proyecto ya establecido) crear manualmente una base de datos de grafos con información obtenida personalmente.
- Por último, para crear el clasificador hubo que establecer manualmente una serie de datos de entrenamiento ya que no se disponía de ningún modelo preentrenado para cumplir dicho proceso. También, se evaluaron distintos modelos clasificadores, cada uno con sus respectivas pruebas de parámetros, para elegir el que mejor rendimiento arroje.