Facultad de
Ciencias Exactas,
Ingeniería y Agrimensura

Trabajo Practico Final

Procesamiento de Lenguaje Natural

Alumno:

- Garcia, Timoteo

Objetivo del trabajo

Crear un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation).

En este caso hice un ChatBot de conocimiento ‘general’, que posee 3 fuentes de datos diferentes, las mismas son:

- Base de datos de vectores: una base de datos vectorial alimentada de un archivo PDF, que el mismo corresponde con la información del DNU dictado por el presidente argentino Javier Milei.
- Base de datos de Grafos: utilice ‘Wiki data’ que es una base de datos de conocimiento libre y abierto que ofrece datos enlazados, descritos mediante RDF. Que esta fuente de datos nos brindara información para responder preguntas generales.
- Archivo CSV: y como fuente de datos tabulares elegí el ranking de Expectativa de vidas en los Países.

Resolución:

Para la resolución de este trabajo iniciamos cargando y procesando las distintas fuentes de datos, y a su vez cargando nuestro embedding de HuggingFace. Luego

creo las funciones para interactuar con la base de datos de grafos y obtener la información en base a el prompt del usuario final.

Hecho esto, pasamos a crear nuestro Bot, utilizando el modelo “zephyr-7b-beta” de HF. Creamos el generador de Prompts con un cierto template que le definimos, y el generador de la respuesta a dicho Prompt. Pero para el generador de respuesta debí crear la función mas importante quizás de este Bot, que es el clasificador de Prompts. La idea del clasificador de Prompts, es poder categorizar la instrucción del usuario con el fin de ‘redirigir’ la misma a la fuente de datos que corresponde.

Vayamos al código:

Cargamos nuestro Embedding:

En este caso utilizamos el modelo “sentence-transformers/paraphrase-multilingual-mpnet-base-v2” de Hugging Face.

```
embed_model = LangchainEmbedding(
    HuggingFaceEmbeddings(model_name='sentence-transformers/p
```

Cargamos y procesamos nuestros Documentos PDF

Cargamos nuestro documento PDF con la información del DNU, para luego armar nuestro retriever para realizar la búsqueda vectorial de información.

La opción `similarity_top_k=2` sugiere que al realizar una búsqueda, se devolverán los documentos más similares (hasta un límite de 2) basados en la similitud de vectores.

```
print('Indexando documentos...')
documents = SimpleDirectoryReader("Data").load_data()
index = VectorStoreIndex.from_documents(documents, show_progr
                                        service_context=Servi

# Construimos un retriever a partir del índice, para realizar
retriever_txt = index.as_retriever(similarity_top_k=2)
```

Cargamos y procesamos nuestros Documentos Tabulares

Hago al similar que con mi PDF, pero con mi archivo .CSV. Pero antes a el mismo archivo le hacemos una transformación para asi generar una columna en el mismo llamada ‘contexto’ que va a hacer que nuestro dataset sea mas nutritivo para el programa.

```
df = pd.read_csv("CSV/Expectativa.csv")
df['Context'] = df.apply(lambda row: f"Para el año {row['Year

csv_file_path = 'CSV/Expectativa.csvExpectativa.csv'

# Exporta el DataFrame a un archivo CSV
df['Context'].to_csv(csv_file_path, index=False)
```

```
documents = SimpleDirectoryReader("CSV").load_data()

index = VectorStoreIndex.from_documents(documents, show_progr
                                        service_context=Servi

# Construimos un retriever a partir del índice, para realizar
retriever_csv = index.as_retriever(similarity_top_k=2)
```

Creamos la función para obtener la información de la base de grafos 'Wikidata'

```
import json
import requests
import nltk
from textblob import TextBlob
from SPARQLWrapper import SPARQLWrapper, JSON
from qwikidata.sparql import return_sparql_query_results
nltk.download('brown')
nltk.download('punkt')


def run_query(id):
    sparql = SPARQLWrapper("https://query.wikidata.org/sparql
    query = """
    SELECT ?wdLabel ?ooLabel
    WHERE {{
     VALUES (?s) {{(wd:{0})}}
```

```
      ?s ?wdt ?o .
      ?wd wikibase:directClaim ?wdt .
      ?wd rdfs:label ?wdLabel .
      OPTIONAL {{
      ?o rdfs:label ?oLabel .
      FILTER (lang(?oLabel) = "en")
      }}
      FILTER (lang(?wdLabel) = "en")
      BIND (COALESCE(?oLabel, ?o) AS ?ooLabel)
      }} ORDER BY xsd:integer(STRAFTER(STR(?wd), "http://www.w
    """.format(id)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()
    return results

def get_related_wikidata(user_query):
    blob = TextBlob(user_query)
    wikidata_info = []

    for noun in blob.noun_phrases:
        params = {
            'action': 'wbsearchentities',
            'format': 'json',
            'language': 'en',
            'search': noun
        }

        API_ENDPOINT = "https://www.wikidata.org/w/api.php"
        response = requests.get(API_ENDPOINT, params=params)
        data = response.json()

        entities_id = []
        for entity in data['search']:
            entities_id.append(entity['id'])

        try:
            result = run_query(entities_id[0])  # Use the fir
```

```
            for r in result["results"]["bindings"]:
                wikidata_info.append({
                    'entity_label': r["wdLabel"]["value"].str
                    'property_label': r["ooLabel"]["value"]
                })
        except Exception as e:
            print(f'Error fetching Wikidata info for {noun}:

    return wikidata_info
```

Creamos los templates, la generación de respuestas, y la creación de prompt, de nuestro chatbot RAG:

Utilizando nuestro modelo de HF elegido y creamos el template para generar nuestros prompts y luego nuestra función generadora de respuesta.

```
def zephyr_instruct_template(messages, add_generation_prompt=
    # Definir la plantilla Jinja
    template_str = "{% for message in messages %}"
    template_str += "{% if message['role'] == 'user' %}"
    template_str += "<|user|>{{ message['content'] }}</s>\n"
    template_str += "{% elif message['role'] == 'assistant' %
    template_str += "<|assistant|>{{ message['content'] }}</s
    template_str += "{% elif message['role'] == 'system' %}"
    template_str += "<|system|>{{ message['content'] }}</s>\n
    template_str += "{% else %}"
    template_str += "<|unknown|>{{ message['content'] }}</s>\
    template_str += "{% endif %}"
    template_str += "{% endfor %}"
    template_str += "{% if add_generation_prompt %}"
    template_str += "<|assistant|>\n"
    template_str += "{% endif %}"

    # 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_
```

Llamamos al modelo

```
def generate_answer(prompt: str, max_new_tokens: int = 768) -
    try:
        api_key = config('HUGGINGFACE_TOKEN')

        api_url = "https://api-inference.huggingface.co/model

        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, js

        respuesta = response.json()[0]["generated_text"][len(
        return respuesta

    except Exception as e:
        print(f"An error occurred: {e}")
```

Creo una función que lo que hace es preprarar nuestro prompt en estilo QA

```
def prepare_prompt(query_str: str, nodes, category):
  TEXT_QA_PROMPT_TMPL = (
      "La información de contexto es la siguiente:\n"
      "---------------------\n"
      "{context_str}\n"
      "---------------------\n"
      "Dada la información de contexto anterior, y sin utiliz
      "Pregunta: {query_str}\n"
```

```
        "Respuesta: "
    )
    if category == '0':
      # Construimos el contexto de la pregunta
      context_str = ''
      for node in nodes:
          page_label = node.metadata["page_label"]
          file_path = node.metadata["file_path"]
          context_str += f"\npage_label: {page_label}\n"
          context_str += f"file_path: {file_path}\n\n"
          context_str += f"{node.text}\n"
    elif category == '1':
      context_str = ''
      if nodes:
          for wd_info in nodes:
              context_str += f'\t{wd_info["entity_label"]}: {wd
      else:
          context_str = "No se encontró información relevante e
    elif category == '2':
      context_str = ''
      for node in nodes:
          file_path = node.metadata["file_path"]
          context_str += f"file_path: {file_path}\n\n"
          context_str += f"{node.text}\n"


    # Construimos los mensajes para el modelo
    messages = [
        {
            "role": "system",
            "content": "Eres un asistente útil que siempre resp
        },
        {"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(
    ]

    final_prompt = zephyr_instruct_template(messages)
    return final_prompt
```

Creo el clasificador de Prompts

Como mencione anteriormente la idea de este clasificador de Prompts es otorgarle una categoría para saber que fuente de datos tiene que utilizar el modelo para generar la respuesta.

Y a su vez creamos la función final que genera la respuesta en base a la categoría.

```
def classify_query(payload):
    response = requests.post(API_URL, headers=headers, json=pa
    return response.json()

def classify_instruction(user_query):
    quest_to_model = f"""
    You are a query classifier that doesnt response queries,
    0 - regulations, changes and information about the dnu pr
    1 - general knowledge
    2 - Countries Life Expectancy

    +"¿Que dicen acerca de la Ley N° 9.643?"</s>

    0</s>

    +"Hablame acerca del DNU"</s>

    0</s>

    +"¿Que va a pasar con la Ley de Alquileres?"</s>

    0</s>

    +"¿Que va a pasar con YPF?"</s>

    0</s>

    +"Explicame quien es Lionel Messi"</s>

    1</s>

    +"¿Cual es la capital de Egipto?"</s>
```

```
    1</s>

    +"¿Como se juega a?"</s>

    1</s>

    +"¿Como se juega al Futbol?"</s>

    1</s>

    +"¿Quien es Barak Obama?"</s>

    1</s>

    +"Expectativa de vida"</s>

    2</s>

    +"Expectativa de vida de Spain en 2020"</s>

    2</s>

    +"¿ Cual fue el Pais con menor expectativa de vida en 201

    2</s>

    +"¿Que puesto ocupo Australia en el ranking de Expectativ

    2</s>

    +"{user_query}"</s>

    """
    return quest_to_model


def generate_answer_based_on_category(category, user_query):
```

```
    if category == '0':
      nodes = retriever_txt.retrieve(user_query)
      final_prompt = prepare_prompt(user_query, nodes, catego
      final_answer = generate_answer(final_prompt)
      return final_answer
    elif category == '1':

      print("Buscando en Wikidata...\n")
      wikidata_info = get_related_wikidata(user_query)
      prepared_prompt = prepare_prompt(user_query, wikidata_i
      final_answer = generate_answer(prepared_prompt)
      return final_answer

    elif category == "2":

      nodes = retriever_csv.retrieve(user_query)
      final_prompt = prepare_prompt(user_query, nodes, catego
      final_answer = generate_answer(final_prompt)
      return final_answer

    else:
        return "No se pudo determinar una categoría para la p
```

Probamos el modelo

```
print('Realizando llamada a HuggingFace para generar respuest

queries = ['Segun el DNU , ¿Que pasara con la ley de alquiler
           '¿Que va a pasar con YPF?',
           '¿Que resolución llevaran a cabo con el Impuesto a
           '¿Quien es Lionel Messi?',
           '¿Quien gano la copa libertadores de America en el
           '¿Que pais tuvo la mayor expectativa de vida en 20
           'Expectativa de vida de Spain en 2008']

for query_str in queries:
  output = classify_query({
      "inputs": classify_instruction(query_str),
```

```
        "parameters": {
            "do_sample": True,
            "max_new_tokens": 1,
            "return_full_text": False
        }
    })

    try:
        # Imprime la categoría asignada por el modelo de clasif
        print(f"Pregunta : {query_str}")

        category = output[0]['generated_text'].strip()
        # Genera la respuesta basada en la categoría
        response = generate_answer_based_on_category(category,
        print(f"Respuesta: {response}")

    except Exception as e:
        print(f"Error: {e}")
```

Y por ultimo arme un 'programita' para que se le puedan hacer las preguntas que quieran:

```
query = input("Ingrese su pregunta, o 'exit' para salir:")
while query != "exit":
  def respuesta(query):
    output = classify_query({
      "inputs": classify_instruction(query),
      "parameters": {
          "do_sample": True,
          "max_new_tokens": 1,
          "return_full_text": False
      }
  })
    categoria = output[0]['generated_text'].strip()


    # Genera la respuesta basada en la categoría
    response = generate_answer_based_on_category(categoria, q
```

```
    return categoria, response

categoria, response = respuesta(query)
print("Repuesta:")
print(response)
query = input("Ingrese su pregunta, o 'exit' para salir:")
```