TECNICATURA UNIVERSITARIA EN INTELIGENCIA ARTIFICIAL (TUIA)

PROCESAMIENTO DEL LENGUAJE NATURAL (NLP)

Trabajo Práctico 2 (TP2)

Estudiante:

Martín L. Perrone Leone

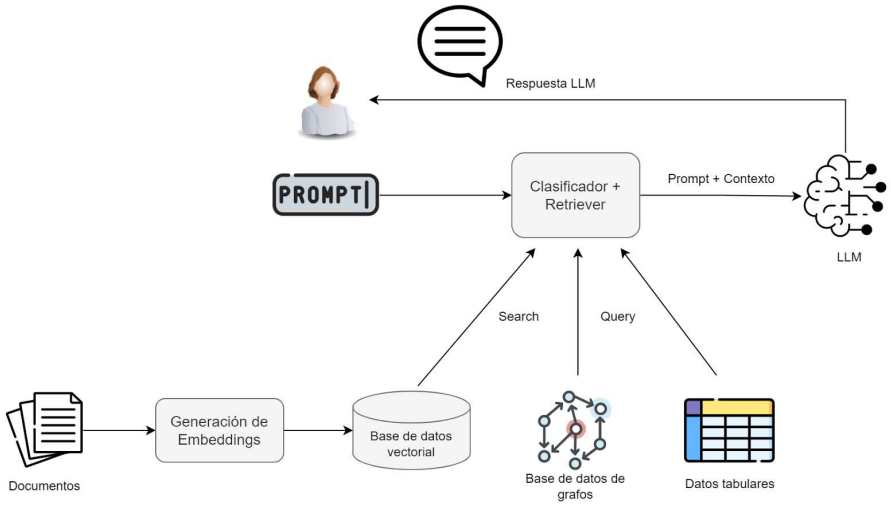
Legajo: P-5197/7

AÑO 2023 (2do CUATRIMESTRE), Rosario.

Ejercicio 1 - RAG

Crear un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation). Como fuentes de conocimiento se utilizarán al menos las siguientes fuentes:

- Documentos de texto
- Datos numéricos en formato tabular (por ej., Dataframes, CSV, sqlite, etc.)
- Base de datos de grafos (Online o local)



El sistema debe poder llevar a cabo una conversación en lenguaje español. El usuario podrá hacer preguntas, que el chatbot intentará responder a partir de datos de algunas de sus fuentes. El asistente debe poder clasificar las preguntas, para saber qué fuentes de datos utilizar como contexto para generar una respuesta.

Requerimientos generales

- Realizar todo el proyecto en un entorno Google Colab
- El conjunto de datos debe tener al menos 100 páginas de texto y un mínimo de 3 documentos.
- Realizar split de textos usando Langchain (RecursiveTextSearch, u otros métodos disponibles). Limpiar el texto según sea conveniente.
- Realizar los embeddings que permitan vectorizar el texto y almacenarlo en una base de datos ChromaDB
- Los modelos de embeddings y LLM para generación de texto son a elección
- Para el desarrollo del “Clasificador” es posible utilizar diversas técnicas aprendidas en la materia, por ejemplo en Unidad 3 y Unidad 6

Ejercicio 2 - Agentes

Realice una investigación respecto al estado del arte de las aplicaciones actuales de agentes inteligentes usando modelos LLM libres.

Plantee una problemática a solucionar con un sistema multiagente. Defina cada uno de los agentes involucrados en la tarea. Es importante destacar con ejemplos de conversación, la interacción entre los agentes.

Realice un informe con los resultados de la investigación y con el esquema del sistema multiagente, no olvide incluir fuentes de información.

Opcional: Resolución con código de dicho escenario.

Informe Ejercicio 1

Introducción:

Intenté resolver el ejercicio con el modelo **Zephyr-7B-β**[1] disponible en HuggingFace, pero debido a que su funcionamiento no es del todo bueno y me hacía utilizar mucho tiempo en pruebas decidí probar con otros. Buscando otro modelo de código abierto encontré el ***Mistral 7b***[2]***,*** *t*ambién en HuggingFace, que está mucho mejor ranqueado[3] pero que no está disponible para utilizarlo por medio de la API de la página. Esto me obligaba a bajarlo, lo que hacía que su ejecución sea muy lenta y complicaba la tarea.
Estos inconvenientes hicieron decidirme por usar el modelo de OpenAI ***gpt-3.5-turbo***[4], que si bien es pago, me resultó de ayuda para la resolución del ejercicio.

1 https://huggingface.co/HuggingFaceH4/zephyr-7b-beta

2 https://huggingface.co/mistralai/Mistral-7B-v0.1

3 https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard

4 https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates

Informe Ejercicio 1 - RAG:

El tema a elección para realizar el chatbot es sobre fútbol. En particular sobre el reglamento, información sobre estadísticas en mundiales y otros datos que puedan ser obtenidos por el chatbot en wikidata.

Los distintos temas se dividieron en cada una de las bases de datos solicitadas:

1. **Base de datos vectorial:** contiene información sobre el reglamento del fútbol en formato pdf en 144 páginas.

2. **Base de datos tabular:** contiene información sobre estadísticas de las selecciones en los mundiales de fútbol. Es una base creada por mí con datos obtenidos en internet, de wikipedia.

3. **Base de datos de grafos:** El chatbot realiza consultas a la base de datos online **Wikidata**.

Según la consulta del usuario el chatbot busca el contexto en alguna de las **tres categorías** definidas en: r*eglamento (1),* mundiales de fútbol *(2), otros (3).*

Código de Bases de Datos:

1. Base de datos vectorial:

Luego de instalar las librerías necesarias comienza la preparación de los datos para la base de datos vectorial. En primer lugar se carga el pdf[5] de nombre ***"Futbol_ fifa.pdf"*** que contiene el reglamento del fútbol, se hace el split[6] del texto y se generan los Embeddings utilizando el modelo ***Paraphrase-multilingual-mpnet-base-v2*** de ***Hugging Face***[7], para vectorizarlo y almacenarlo en una base de datos[8]. El pdf está bastante limpio y en las pruebas que hice no tomaba los número de páginas y otras cosas que puedan ensuciar los datos, por lo que no necesita limpieza.

```
[5]  # Cargo el pdf sobre reglas de fútbol
     loader_futbol = PyPDFLoader("Futbol_ fifa.pdf")

     # Divido el contenido del PDF en páginas
     pages_futbol = loader_futbol.load_and_split()

[6]  # Concatena las páginas de pages_futbol
     text_futbol = ''.join([page.page_content for page in pages_futbol])

     # Crea un objeto para dividir el texto en fragmentos
     text_splitter_basquet = CharacterTextSplitter(separator ="\n",chunk_size=800, chunk_overlap=10)

     # Crea documentos a partir de los fragmentos de texto
     documents = text_splitter_basquet.create_documents([text_futbol])

     # Genero una base de datos
     db = Chroma.from_documents(documents, embed_model)
```

Una vez que el clasificador da como resultado que la consulta realizada por el usuario es de la categoría "reglamento", se realiza una búsqueda de similitud en la base de datos utilizando la consulta. La función ***similarity_search*** de la base de datos toma la pregunta como entrada y devuelve documentos similares. Se devuelve un string con todos los documentos como contexto para la respuesta.

---

5 PyPDFLoader de Langchain

6 CharacterTextSplitter de Langchain

7 **https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2**
Es un es un modelo de sentence-transformers que mapea oraciones y párrafos a un espacio vectorial de 768 dimensiones y puede utilizarse para tareas como clustering o búsqueda semántica

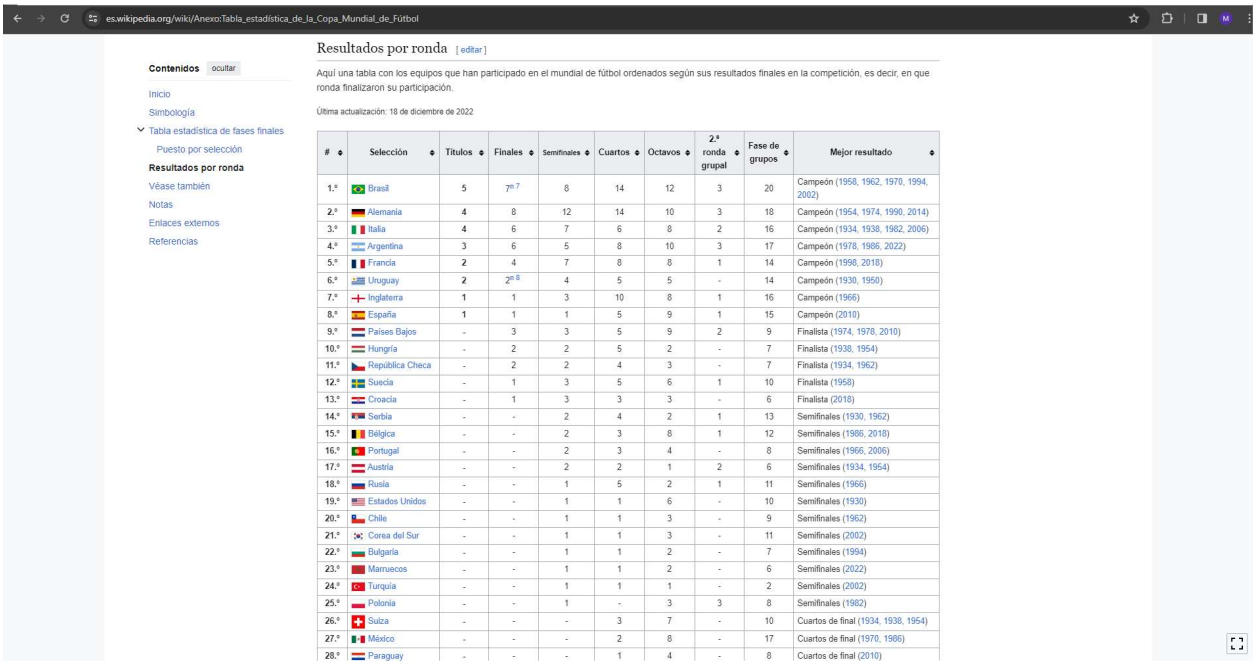
8 Chroma de Langchain

```python
def search_in_vector_store(question: str):

    docs = db.similarity_search(question)
    context_str = ''
    for node in docs:
        context_str += f"{node.page_content}\n"
    return context_str
```

2. Base de datos tabular:

Para la base de datos tabular generé un archivo sql ***"mundiales_stats.sql"*** con las consultas para generar una base de datos relacional con la información sobre mundiales que obtuve de ***wikipedia***[9].



es.wikipedia.org/wiki/Anexo:Tabla_estadística_de_la_Copa_Mundial_de_Fútbol

Contenidos ocultar

- Inicio
- Simbología
- Tabla estadística de fases finales
  - Puesto por selección
  - **Resultados por ronda**
- Véase también
- Notas
- Enlaces externos
- Referencias

Resultados por ronda [editar]

Aquí una tabla con los equipos que han participado en el mundial de fútbol ordenados según sus resultados finales en la competición, es decir, en que ronda finalizaron su participación.

Última actualización: 18 de diciembre de 2022

#	Selección	Títulos	Finales	Semifinales	Cuartos	Octavos	2.ª ronda grupal	Fase de grupos	Mejor resultado
1.º	Brasil	5	7[n 7]	8	14	12	3	20	Campeón (1958, 1962, 1970, 1994, 2002)
2.º	Alemania	4	8	12	14	10	3	18	Campeón (1954, 1974, 1990, 2014)
3.º	Italia	4	6	7	6	8	2	16	Campeón (1934, 1938, 1982, 2006)
4.º	Argentina	3	6	5	8	10	3	17	Campeón (1978, 1986, 2022)
5.º	Francia	2	4	7	8	8	1	14	Campeón (1998, 2018)
6.º	Uruguay	2	2[n 8]	4	5	5	-	14	Campeón (1930, 1950)
7.º	Inglaterra	1	1	3	10	8	1	16	Campeón (1966)
8.º	España	1	1	1	5	9	1	15	Campeón (2010)
9.º	Países Bajos	-	3	3	5	9	2	9	Finalista (1974, 1978, 2010)
10.º	Hungría	-	2	2	5	2	-	7	Finalista (1938, 1954)
11.º	República Checa	-	2	2	4	3	-	7	Finalista (1934, 1962)
12.º	Suecia	-	1	3	5	6	1	10	Finalista (1958)
13.º	Croacia	-	1	3	3	3	-	6	Finalista (2018)
14.º	Serbia	-	-	2	4	2	1	13	Semifinales (1930, 1962)
15.º	Bélgica	-	-	2	3	8	1	12	Semifinales (1986, 2018)
16.º	Portugal	-	-	2	3	4	-	8	Semifinales (1966, 2006)
17.º	Austria	-	-	2	2	1	2	6	Semifinales (1934, 1954)
18.º	Rusia	-	-	1	5	2	1	11	Semifinales (1966)
19.º	Estados Unidos	-	-	1	1	6	-	10	Semifinales (1930)
20.º	Chile	-	-	1	1	3	-	9	Semifinales (1962)
21.º	Corea del Sur	-	-	1	1	3	-	11	Semifinales (2002)
22.º	Bulgaria	-	-	1	1	2	-	7	Semifinales (1994)
23.º	Marruecos	-	-	1	1	2	-	6	Semifinales (2022)
24.º	Turquía	-	-	1	1	1	-	2	Semifinales (2002)
25.º	Polonia	-	-	1	-	3	3	8	Semifinales (1982)
26.º	Suiza	-	-	-	3	7	-	10	Cuartos de final (1934, 1938, 1954)
27.º	México	-	-	-	2	8	-	17	Cuartos de final (1970, 1986)
28.º	Paraguay	-	-	-	1	4	-	8	Cuartos de final (2010)

9 https://es.wikipedia.org/wiki/Anexo:Tabla_estadística_de_la_Copa_Mundial_de_Fútbol

Usando ***sqlite*** se genera la tabla, se levanta el archivo sql y se ejecutan las consultas del mismo. De esta manera se genera la base de datos.

```python
import sqlite3
con = sqlite3.connect("mundiales.db")

# Crea un cursor para ejecutar comandos SQL
cursor = con.cursor()

# Genera la tabla si no existe

cursor.execute("CREATE TABLE IF NOT EXISTS mundiales_stats (seleccion VARCHAR(100), titulos INTEGER,\
                finales INTEGER, semifinales INTEGER, cuartos INTEGER, octavos INTEGER, segundaronda INTEGER, \
                fasegrupos INTEGER, mejorresultado VARCHAR(255))")


# Abre el archivo "mundiales_stats.sql" que contiene instrucciones SQL
with open('mundiales_stats.sql', encoding="utf8") as f:
  sql_insterts=f.read()

# Divide las instrucciones SQL en una lista
sql_list = sql_insterts.split(";")
print(len(sql_list))

# Ejecuta cada instrucción SQL en la lista
for sql in sql_list:
    cursor.execute(sql)

# Confirma los cambios en la base de datos
con.commit()

# Cierra la conexión a la base de datos
con.close()
```

Si el clasificador da como resultado la categoría “mundiales de fútbol”, se utiliza el modelo para que genere una query sql para la base de datos generada, dicha query se ejecuta con ***sqlite*** para obtener la respuesta, que se utiliza como contexto para responder la consulta del usuario. Para ayudar a mejorar la información de contexto se le agregan las palabras claves de las preguntas.

```python
def search_in_sql(question: str):
    # Crea una plantilla de prompt para solicitar una consulta SQL
    prompt_template = PromptTemplate.from_template(
        """Sos un esperto en SQLLite. Tu tarea es crear un sql para SQLLite sintáticamente correcto listo para ejecutar \
    para contestar la pregunta dada a continuación y conociendo el esquema de la tabla. Sólo consultar las columnas \
    necesarias para responder la pregunta y alguna adicional que sirva para clarificar la respuesta.

    Prestá \
    atención y usá sólo los nombres de columna que puedes ver en la tabla que se indica a continuación. Tenés especial \
    cuidado de no consultar columnas que no existan. Prestá atención y usá la función CURRENT_DATE para obtener la fecha actual, si
    la pregunta tiene que ver con el "hoy".

    La salida sólo debe ser el sql listo para ejecutar.

    Esquema de tabla a consultar: ```mundiales_stats (seleccion VARCHAR(100), titulos INTEGER, finales INTEGER, semifinales INTEGER, cuartos INTEGER, \
    octavos INTEGER, segundaronda INTEGER, fasegrupos INTEGER, mejorresultado VARCHAR(255))```

    Pregunta: ```{question}```
    """)

    prompt_val = prompt_template.format(question=question)

    # Crea una lista de mensajes para interactuar con el modelo
    messages = [
        SystemMessage(content="Sos un esperto en SQLLite"),
        HumanMessage(content=prompt_val),
    ]

    response = chat.invoke(messages)
    sql_query = response.content
    sql_query = sql_query.replace("sql", "").replace("```", "")

    # Conecta a la base de datos
    con = sqlite3.connect("mundiales.db")
    cursor = con.cursor()
    try:
      # Ejecuta la consulta
      cursor.execute(sql_query)
      resultado = cursor.fetchall()

      # Obtiene los nombres de las columnas
      columnas = [descripcion[0] for descripcion in cursor.description]

      # Imprime los resultados en formato de texto
      contexto = ""
      for fila in resultado:
        for i in range(len(columnas)):
              contexto = contexto + str(columnas[i]) + ": " + str(fila[i]) + " "
        contexto =  contexto + "\n"
    except Exception as e:
       print(f"An error occurred: {e}")
       contexto = ""
    # Cierra la conexión
    con.close()

    # Procesa la pregunta para identificar palabras clave
    question_modf = ''.join(c for c in question if c.isalnum() or c.isspace())
    palabras_question = question_modf.split()
    sql_query_modif = ''.join(c for c in sql_query if c.isalnum() or c.isspace())
    #print(palabras_question)

    # Identifica palabras clave en la consulta SQL
    inf_extra_contexto = ""
    for i in palabras_question:
      if i in sql_query_modif:
        inf_extra_contexto += " " + i + " "

    return inf_extra_contexto + contexto
```

3. Base de datos de grafos:

La información de la base de datos de grafos se obtienen con consultas ***SPARQL*** a ***Wikidata***.[10]
Si el clasificador da como resultado la categoría “otros”, con el modelo se genera la query a partir de la pregunta del usuario. Una vez obtenidas las respuestas se toman de las mismas nombres y descripciones. En las pruebas que hice, las query que funcionaron se armaron en base a frases como “Equipos de fútbol” o “Jugadores de fútbol”, devolviendo el modelo una lista con esa información.

Realmente me resultó muy difícil hacer que las consultas a ***Wikidata*** me den respuestas, incluso intentando desde ***https://query.wikidata.org/***, por lo que lo pude resolver de esa manera.

```python
from SPARQLWrapper import SPARQLWrapper, JSON

# Configura el punto de acceso SPARQL de Wikidata
sparql = SPARQLWrapper("https://query.wikidata.org/sparql")

def get_wikidata_entity_name(wikidata_id):

    # URL del punto de acceso SPARQL de Wikidata
    url = 'https://query.wikidata.org/sparql'

    # Consulta SPARQL para obtener el nombre de la entidad
    query = f'''
    SELECT ?label WHERE
    {{
        SERVICE wikibase:label {{
            bd:serviceParam wikibase:language "en" .
            wd:{wikidata_id} rdfs:label ?label .
        }}
    }}
    '''
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    data = sparql.query().convert()
    return {"label": data["results"]["bindings"][0]["label"]["value"]}
```

10 https://www.wikidata.org/

```python
def get_wikidata_description(wikidata_id):
    url = 'https://query.wikidata.org/sparql'
    query = f'''
    SELECT ?description WHERE
    {{
        SERVICE wikibase:label {{
            bd:serviceParam wikibase:language "en" .
            wd:{wikidata_id} schema:description ?description .
        }}
    }}
    '''
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    data = sparql.query().convert()
    # Extrae la descripción de la entidad de los resultados
    return {"description": data["results"]["bindings"][0]["description"]["value"]}

def search_in_wikidata(question: str):
    # Crea una plantilla de prompt para solicitar una consulta SPARQL
    prompt_template = PromptTemplate.from_template(
        """Tu tarea es crear un consulta SPARQL para consultar datos de Wikidata \
        para contestar la pregunta dada a continuación. Es importante que la consulta sea sintáticamente correcto y esté lista para ejecutar.
        Armar la consulta para que el AUTO_LANGUAGE sea español.

        La salida sólo debe ser la consulta SPARQL lista para ejecutar sin ninguna explicación adicional. Siempre limita a máximo 10 registros

        Pregunta: ```{question}```"""
        )
    prompt_val = prompt_template.format(question=question)

    # Crea una lista de mensajes para interactuar con el modelo
    messages = [
        SystemMessage(content="Sos un asistente experto en SPARQL y Wikidata"),
        HumanMessage(content=prompt_val),
    ]

    response = chat.invoke(messages)
    wiki_query = response.content.replace('\n', '').replace("'","").replace("sparql", "").replace("```", "")
    print(wiki_query)

    # Realiza la consulta SPARQL a través de la URL del punto de acceso de Wikidata
    try:
      sparql.setQuery(wiki_query)
      sparql.setReturnFormat(JSON)
      results = sparql.query().convert()
      print(results)

      # Extrae los resultados de la consulta
      head = results["head"]["vars"][0]

      context = ""

      for data in results["results"]["bindings"]:
          for head in  results["head"]["vars"]:
            if "WHERE" in head:
              continue
            valor = data[head]["value"]
            print("...data: "+ valor + " id? "+str(("http://www.wikidata.org/entity/" in valor)))
            if "http://www.wikidata.org/entity/" in valor:
              id = valor.replace("http://www.wikidata.org/entity/", "")
              try:
                descripcion =  str(get_wikidata_description(id)["description"])
              except:
                descripcion = ""
              context = context + head +": " + str(get_wikidata_entity_name(id)["label"]) + " - " + descripcion + "\n"
            else:
              context = context + head +": " + valor +  "\n"
    except Exception as e:
       print(f"An error occurred: {e}")
       context = ""

    return  context
```

Código del clasificador:

Para el clasificador se utiliza el modelo, indicándole mediante el prompt que su función es la de clasificar la pregunta entre las tres categorías: ***reglamento, mundiales de fútbol y otros***. Según la respuesta que de el clasificador, se pasa la pregunta a la base de datos correspondiente para que genere el contexto.

```
def clasificador(question: str):

    prompt_template = PromptTemplate.from_template(
        """Tu tarea es clasificar la pregunta que se te indica a continuación entre las siguientes tres categorias [reglamento, mundiales de fútbol, otros]

    La respuesta sólo debe ser una de las tres categorias sin otro texto.

    Pregunta: ```{question}```
    """)

    prompt_val = prompt_template.format(question=question)
    messages = [
        SystemMessage(content="Sos un experto en temas de deporte y ayudas a un asistente virtual a encontrar el contexto adecuado para contestar preguntas."),
        HumanMessage(content=prompt_val),
    ]

    response = chat.invoke(messages)

    return response.content
```

Código del chatbot:

Al modelo se le pasan la consulta del usuario y el contexto y se le indica mediante el prompt que debe responder a la consulta que se le hace solamente utilizando este último. Si no tiene información suficiente, lo debe aclarar en la respuesta.

```
def generar_respuesta(query: str, context: str):
  prompt_template = PromptTemplate.from_template(
    """La información de contexto es la siguiente:\n"
      "```{context_str}```"
      "\nUtilizando únicamente la información de contexto anterior y sin mencionarla en la respuesta, responde la siguiente pregunta. Si no podés contestar con información del contexto \
      respondé 'No tengo información suficiente.'.\n"
      "Pregunta: ```{query_str}```"""
  )
  prompt_val = prompt_template.format(query_str=query, context_str=context)
  print(prompt_val)
  messages = [
      SystemMessage(content="Eres un asistente útil que siempre responde con respuestas veraces, útiles y basadas en hechos."),
      HumanMessage(content=prompt_val),
  ]

  response = chat.invoke(messages)
  return response.content
```

Interfaz del chatbot:

Por último se utiliza una interfaz de ***ChatInterface de Gradio***[11] para la presentación del chatbot.

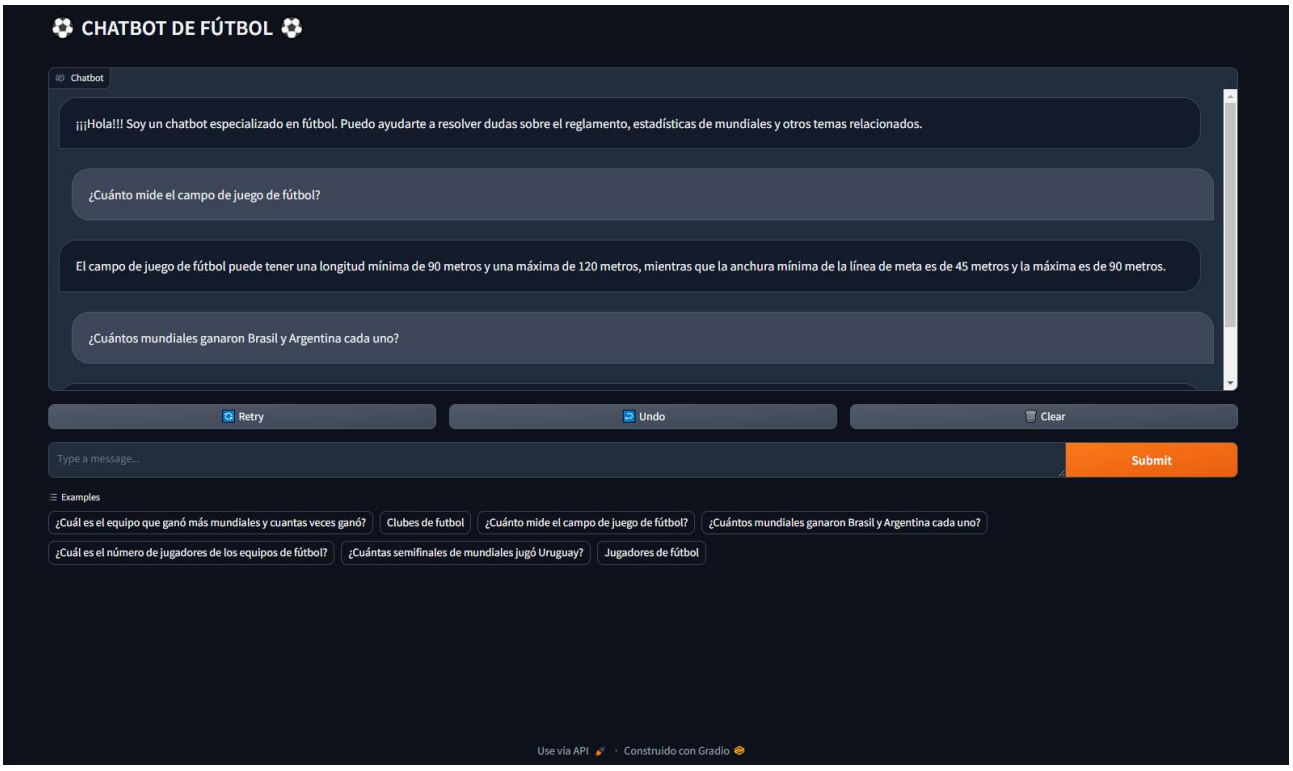
```python
# Usa una interfaz de ChatInterface para el chat
def chat_response(query, history):
    # Usa la función clasificador para definir la categoría de la pregunta
    qtype = clasificador(query)
    print(qtype)
    context = ""
    if qtype == "reglamento":
      context = search_in_vector_store(query)
    elif qtype == "mundiales de fútbol":
      context = search_in_sql(query)
    else:
      context = search_in_wikidata(query)

    response = generar_respuesta(query, context)

    return  response

demo = gr.ChatInterface(fn = chat_response,examples=["¿Cuál es el equipo que ganó más mundiales y cuantas veces ganó?",
                                                     "Clubes de futbol",
                                                     "¿Cuánto mide el campo de juego de fútbol?",
                                                     "¿Cuántos mundiales ganaron Brasil y Argentina cada uno?",
                                                     "¿Cuál es el número de jugadores de los equipos de fútbol?",
                                                     "¿Cuántas semifinales de mundiales jugó Uruguay?",
                                                     "Jugadores de fútbol"] ,title = '⚽ CHATBOT DE FÚTBOL ⚽'
                                                     , chatbot=gr.Chatbot(value=[(None, "¡¡¡Hola!!! Soy un chatbot especializado en fútbol. Puedo ayudarte a resolver

#demo.launch(debug=True)
demo.launch(debug=True)
```



11 https://www.gradio.app/

Conclusión:

El chatbot funciona relativamente bien cuando usa la base de dato vectorial y la base de datos tabular, aunque se podría mejorar bastante sobre todo en el tamaño de la última. Para la base de datos de grafos, al hacer las consultas directamente a ***Wikidata***, el chatbot responde algunas veces. En este caso se podría generar una base de datos propia, también grande, y de esta manera se lograría mejorar mucho su funcionamiento.
Por otro lado, en todos los casos teniendo más conocimiento sobre el uso de los prompt, seguramente se podría optimizar mucho el chatbot.

Informe Ejercicio 2

Qué es un Agente Inteligente? Características, tipos y cómo funciona

En el desarrollo constante de la inteligencia artificial, los agentes inteligentes cobran cada vez más relevancia en el ámbito tecnológico, y es que, su extrema utilidad en los numerosos aspectos del mundo real, hacen de esta **entidad autónoma** un sistema que se aplicará cada vez más, y seremos testigos de sus grandes ventajas cuando experimentemos los beneficios que aportan a la humanidad.

Definición de agente inteligente

En inteligencia artificial, un agente inteligente es un **sistema perceptivo** capaz de interpretar y procesar la información que recibe de su entorno, actuando en consecuencia de acuerdo a los datos que recoge y procesa. La forma de actuar de esta entidad es lógica y racional basándose en las reacciones del comportamiento normal de un sistema en concreto. Utiliza **sensores** para recibir información y **actuadores** para ejercer sus funciones.

Características de los agentes inteligentes

- El agente tiene capacidad de raciocinio.
- El agente aprende por sí mismo en función de la información que recibe y guarda.
- Puede interactuar con el entorno que le rodea.
- Tiene la capacidad de cooperar con otros agentes para cumplir una labor en concreto.
- Un agente puede tomar decisiones propias y obrar según su criterio.
- El comportamiento de un agente es determinado de acuerdo a la información percibida del entorno.
- Si el agente tiene una secuencia de actuaciones, es debido a la captación continua de información de la cual es sensible.

Tipos de agentes inteligentes

Los agentes inteligentes se clasifican en 6 tipos diferentes, donde cada uno distingue su utilidad y características que lo definen:

1. **Agente de reactivo simple:** Cuando una percepción en concreto coincide con una regla programada, el agente responde según la forma en que fue predispuesto. Este accionar se conoce como condición-acción. Toman decisiones basadas únicamente en la percepción actual, respondiendo a señales ambientales inmediatas sin ninguna memoria interna de eventos pasados.

   Ejemplo: Un termostato que enciende el aire acondicionado cuando la temperatura actual supera un determinado umbral es un simple agente reflejo.

2. **Agente reactivo basado en modelo:** Este tipo de agente permite simular su acción de respuesta y sus diversas interacciones en un entorno preparado. De esta forma se estudia su comportamiento y sus efectos en el espacio de actuación. Utilizan sensores para recabar información y tienen en cuenta el historial de percepciones, lo que permite tomar decisiones más sofisticadas.
   Ejemplo: Una IA que juega al ajedrez y que tiene en cuenta el historial de jugadas y el estado actual del tablero para decidir la siguiente jugada es un agente basado en modelos.

3. **Agente basado en metas:** Combina características del agente reactivo simple y agente reactivo basado en modelo. En este caso, este tipo de agente tiene un objetivo en concreto, por lo tanto, está programado para buscar la vía más óptima y planificar un conjunto de acciones para cumplir dicho propósito.

4. **Agente basado en utilidad:** Este agente tiene varios propósitos como sistema inteligente, además, posee una herramienta para medir el valor de su comportamiento en el cumplimiento de sus metas establecidas. Sus estándares de conducta garantizan alta calidad en sus acciones.

5. **Agente que aprende:** Es un tipo de agente que busca aprender de sus acciones mientras se encuentra en funcionamiento. Es un sistema altamente complejo, ya que está programado para interactuar con el mundo real, además de tener preestablecidas varias metas a alcanzar. Posee en su interior un elemento que indica el éxito de la entidad, y tiene la capacidad de interactuar en entornos que no conoce.
   Ejemplo: Un filtro de spam que aprende a identificar nuevos tipos de correos basura basándose en los comentarios de los usuarios es un agente de aprendizaje.

6. **Agente de consulta:** Se dedica a responder consultas por parte de las personas que interactúan con este sistema. Tiene la peculiaridad de **crear varios agentes**, y dividir la pregunta del usuario en varias tareas para su respectiva solución. Además, en caso de que los agentes asignados no sean capaces de responder con exactitud la incógnita enviada, se crearán más agentes y se buscarán en más bases de datos para ofrecer una **resolución completa de la problemática**.

¿Cómo funciona un agente inteligente?

Un agente inteligente funciona mediante la **obtención constante de datos percibidos del entorno** en el que se encuentra. De acuerdo a los datos obtenidos, la entidad inteligente tiene un comportamiento u otro. Además, debido a que posee autonomía propia, puede asignarse nuevas reglas para continuar su crecimiento como sistema intelectual.

Agentes LLM

Los modelos de lenguaje grandes (LLM) están revolucionando el campo de la inteligencia artificial. Estos modelos, como GPT-4 y PaLM2, han abierto nuevas posibilidades para aplicaciones de agentes inteligentes.

1. Planificación: Para abordar problemas complejos, un agente debe:

- Reflexionar sobre sus observaciones y pensamientos.
- Ser crítico en su proceso de razonamiento.
- Poder concatenar su razonamiento en una serie de estados intermedios.
- Dividir los objetivos en unidades más pequeñas.

Estos problemas se puede abordar mediante dos tipos de técnicas:

Técnicas de Descomposición de Tareas:

•**Input-Output Prompting (IO):** Este enfoque va directamente de la entrada a la salida. Es útil para preguntas sencillas, como cuando le preguntamos a Chat-GPT la fecha de nacimiento de algún famoso. Sin embargo, para preguntas más complejas, puede generar respuestas incoherentes.

•**Chain of Thought Prompting (CoT):** En CoT, se parte del input y se divide el razonamiento en pasos intermedios. Esto facilita que el modelo llegue a conclusiones razonables.

•**Self Consistency with CoT (CoT-SC):** En lugar de un único CoT, se utilizan múltiples cadenas. Un voto mayoritario asegura que al menos una de esas cadenas tenga el razonamiento correcto. Estas técnicas refinan el plan de acción del agente.

Técnicas de Reflexión:

•Estas técnicas combinan aspectos del razonamiento (como CoT) con la actuación en un entorno externo (como los agentes que utilizan Reinforcement Learning).

•ReAct (Reason+Act): ReAct trabaja con tres conceptos: el "pensamiento inicial" del agente (Thought), la actuación en el entorno (Act) y la observación (Obs) posterior a la acción. Con ReAct, podríamos crear un agente inteligente que, a partir de un prompt de entrada y las herramientas adecuadas, resuelva problemas por sí mismo.

2. Memoria:

•Memoria a corto plazo: Almacena acciones y pensamientos temporales mientras el agente responde a una sola pregunta del usuario.

•Memoria a largo plazo: Contiene acciones y pensamientos relacionados con eventos pasados y un historial de conversaciones.

3. Herramientas:

•Son flujos de trabajo ejecutables que los agentes utilizan para realizar tareas.

•Ejemplos incluyen RAG para respuestas contextuales, intérpretes de código para programación y APIs para buscar información en internet o datos meteorológicos.

La combinación de estos elementos define a un agente LLM.

Investigación respecto al estado del arte de las aplicaciones actuales de agentes inteligentes usando modelos LLM libres:

AutoGPT

AutoGPT cambia drásticamente la relación entre la inteligencia artificial y el usuario final (es decir, tú). Mientras que ChatGPT se basa en un intercambio entre la IA y el usuario, donde tú le das una solicitud y la IA devuelve un resultado, Auto-GPT solo necesita un mensaje inicial de tu parte. A partir de ahí, el agente de IA generará una lista de tareas que cree que necesita para cumplir con lo que le pediste, sin requerir más información o mensajes. Esencialmente, encadena "pensamientos" de modelos LLM según el desarrollador Toran Bruce Richards.

Auto-GPT es un sistema complejo que depende de múltiples componentes. Se conecta a internet para obtener información y datos específicos (algo que la versión gratuita de ChatGPT no puede hacer), cuenta con gestión de memoria a corto y largo plazo, utiliza GPT-4 para la generación de texto más avanzada de OpenAI y GPT-3.5 para almacenamiento y resumen de archivos.

Ejecutar AutoGPT es bastante simple, solo debes seguir los pasos descritos en el repositorio de ***Github: https://github.com/Significant-Gravitas/Auto-GPT*** Una vez que lo ejecutas te pedirá el nombre, el rol, y sus objetivos (goals). Ejemplo: el nombre es AutoGPT-Demo, el rol es "Una inteligencia artificial diseñada para enseñarme sobre Auto gpt" y sus objetivos son "buscar auto gpt, buscar el repositorio en github y averiguar de que se trata el proyecto, explicar que es autogpt en un archivo autogpt.txt, finalizar".

De esta forma podemos ver como el agente se pone a trabajar y crea una AI que tiene un rol y objetivos específicos, y hará todo lo necesario para cumplir con sus metas.

BabyAGI

BabyAGI es otro proyecto que se centra en el desarrollo de agentes autónomos con habilidades de planificación a largo plazo y uso de memoria. Al igual que AutoGPT, BabyAGI busca crear agentes que puedan aprender y adaptarse a su entorno utilizando técnicas de aprendizaje por refuerzo. BabyAGI es una versión simplificada que demuestra cómo la inteligencia artificial puede mejorar la eficiencia en la gestión de tareas.

¿Cómo funciona BabyAGI?

El script de BabyAGI opera en un bucle infinito que realiza los siguientes pasos: Extrae la primera tarea de la lista de tareas. Envía la tarea al agente de ejecución, que utiliza la API de OpenAI para completarla en

función del contexto. Enriquece el resultado y lo almacena en Pinecone. Crea nuevas tareas y reorganiza la lista de tareas según el objetivo y el resultado de la tarea anterior.

Integración con OpenAI y Pinecone

BabyAGI aprovecha las capacidades de procesamiento del lenguaje natural (NLP) de OpenAI para crear nuevas tareas basadas en el objetivo predefinido. Pinecone, por otro lado, se utiliza para almacenar y recuperar los resultados de las tareas, proporcionando contexto para futuras tareas.

Simulaciones de Agentes: CAMEL y Generative Agents

Las simulaciones de agentes son entornos en los que se pueden probar y evaluar agentes autónomos. Los proyectos CAMEL y Generative Agents se centran en la creación de entornos de simulación específicos para agentes autónomos.

CAMEL

CAMEL es un proyecto que se centra en hacer que dos agentes con personalidades distintas interactúen entre sí de manera colaborativa en un entorno de simulación específico.

Los agentes tienen memoria a corto y largo plazo y utilizan pasos de reflexión para asignar puntuaciones de importancia a las observaciones y generalizar lo que han aprendido. La novedad de CAMEL radica en la interacción entre dos agentes y la colaboración en un entorno de simulación.

Generative Agents

Generative Agents es otro proyecto que se centra en la creación de entornos de simulación para agentes autónomos. Estos agentes se basan en modelos generativos para imitar comportamientos individuales y grupales que son similares a los humanos, y se fundamentan en sus “identidades, entorno y experiencias cambiantes”. Como parte de un estudio, investigadores de Stanford y Google crearon un entorno interactivo inspirado en el popular videojuego de simulación The Sims.

Este entorno de “sandbox” permitió a los investigadores estudiar los comportamientos individuales y sociales de los agentes de inteligencia artificial. Algunos de los agentes demostraron preferencias y objetivos similares a los humanos, como el caso de la Agente Alice, que es pintora.

¿Cómo funciona Generative Agents?

Los Generative Agents siguen una arquitectura que consta de componentes como observación, reflexión y planificación. La arquitectura también puede almacenar registros de las experiencias de un agente mediante el uso de lenguaje natural. Según el artículo de investigación, los usuarios pudieron interactuar con los agentes a través del lenguaje natural. Además, los agentes generativos también tenían la capacidad de entablar conversaciones. En el experimento, se descubrió que los agentes generativos pueden realizar diversas tareas, como preparar desayunos, ir al trabajo, pintar, formar relaciones, compartir opiniones e iniciar conversaciones. Son sorprendentemente similares a los humanos, ya que pueden recordar, recuperar y reflexionar sobre el pasado y el presente. Los Generative Agents son un concepto innovador que combina lo mejor de la inteligencia artificial y la simulación interactiva. Tienen la capacidad de crear simulaciones realistas del comportamiento humano que los usuarios pueden explorar. Sin embargo, al igual que cualquier otra tecnología, los agentes generativos también requieren una consideración cuidadosa en cuanto a sus implicaciones éticas, ya que la idea de que los humanos coexistan con seres virtuales conlleva una serie de desafíos.

LangChain

LangChain es un marco para desarrollar aplicaciones impulsadas por modelos de lenguaje. Permite aplicaciones que: Son conscientes del contexto: conectan un modelo de lenguaje a fuentes de contexto (instrucciones de comandos, ejemplos breves, contenido para fundamentar sus respuestas, etc.). Razonan: dependen de un modelo de lenguaje para razonar (sobre cómo responder basado en el contexto proporcionado, qué acciones tomar, etc.). Este marco consta de varias partes. Bibliotecas de LangChain: Las bibliotecas de Python y JavaScript. Contienen interfaces e integraciones para una multitud de componentes, un tiempo de ejecución básico para combinar estos componentes en cadenas y agentes, e implementaciones listas para usar de cadenas y agentes. Plantillas de LangChain: Una colección de arquitecturas de referencia fácilmente desplegables para una amplia variedad de tareas. LangServe: Una biblioteca para implementar cadenas de LangChain como una API REST. LangSmith: Una plataforma para desarrolladores que te permite depurar, probar, evaluar y monitorear cadenas construidas en cualquier marco de modelos de lenguaje grande (LLM) e integra perfectamente con LangChain. Las bibliotecas de LangChain en sí mismas se componen de varios paquetes diferentes.

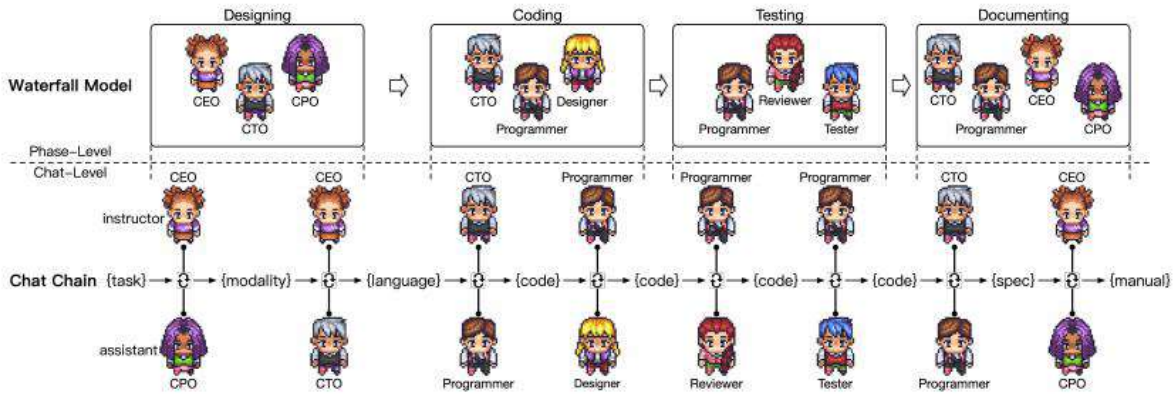
ChatDev

ChatDev

ChatDev es una empresa de software virtual que opera a través de varios agentes inteligentes que desempeñan diferentes roles, incluyendo CEO, CPO, CTO, programador, revisor, probador y diseñador. Estos agentes forman una estructura organizativa de múltiples agentes y están unidos por una misión para "revolucionar el mundo digital a través de la programación". ChatDev IDE es una extensión de navegador que permite conectar sin problemas las conversaciones entre varios agentes dentro de un navegador web. Consta de tres partes: Modo de Juego, Modo de Chat y Prompt IDE2. Puedes personalizar estos NPCs, personalizar el indicador de ubicación y construir tus GPTs con el editor de indicadores visuales. Además, ChatDev se lanzó como una plataforma SaaS que permite a los desarrolladores de software e innovadores empresarios construir software de manera eficiente a un costo muy bajo y con una barrera de entrada muy baja.

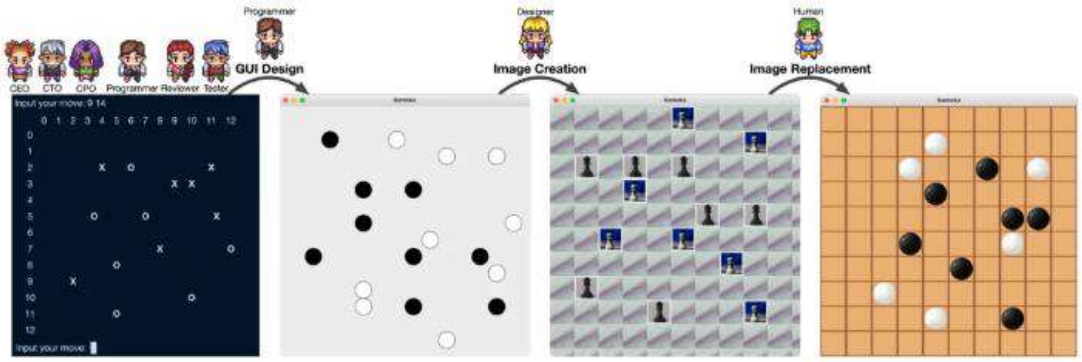
Los científicos están explorando el uso de marcos basados en aprendizaje profundo para cumplir tareas dentro del proceso de desarrollo de software y así mejorar la efectividad. ChatDev es una simulación de una empresa virtual especializada en desarrollo de software que usa un enfoque basado en agentes LLM y busca la necesidad de eliminar modelos especializados en cada fase del desarrollo. Opera a través de varios agentes inteligentes que desempeñan diferentes roles, incluyendo al Director Ejecutivo, Director de Producto, Director Tecnológico, Programador, Revisor, Tester y Diseñador de arte. Utiliza LLMs y su comunicación en lenguaje natural unifica y optimiza procesos claves.

Demostró ser efectivo y rentable para completar el proceso de desarrollo de software Además identifica vulnerabilidades y rectifica posibles alucinaciones de código (causadas por la falta de especificación de tareas (falta del pensamiento guiado al que conducen analizar requerimientos de usuario o seleccionar un lenguaje de programación entre otros) y falta de contrainterrogatorio). Adopta el modelo en cascada dividendo el proceso en 4 fases: Diseño, Codificación, Pruebas y Documentación. En cada etapa del proceso utiliza un equipo de agentes virtuales que colaboran entre sí mediante diálogos y resultan en un flujo de trabajo fluido. Utiliza una cadena de chat que divide cada etapa del proceso en subtareas donde participan dos roles que hacen propuestas y validan soluciones a través de la comunicación.



Al final de todo el proceso devuelve un software con el código fuente, manuales de usuario, y especificaciones de entorno de dependencia.

Ejemplo de lo que se puede lograr: Juego de damas:



La figura más a la izquierda demuestra el software básico creado por el marco sin utilizar ninguna GUI. La aplicación sin GUI ofrece interactividad limitada y los usuarios pueden jugar a este juego sólo a través del terminal de comandos. La siguiente figura muestra un juego visualmente más atractivo creado con el uso de GUI, que ofrece una mejor experiencia de usuario y una interactividad mejorada para un entorno de juego atractivo que los usuarios pueden disfrutar mucho más. Luego, el agente diseñador crea gráficos adicionales para mejorar aún más la usabilidad y la estética del juego sin afectar ninguna funcionalidad. Sin embargo, si los usuarios humanos no están satisfechos con la imagen generada por el diseñador, pueden reemplazar las imágenes después de que se haya completado el software. Si bien ChatDev utiliza ChatGPT-turbo-16 para simular el desarrollo de software multiagente resulta un caso al menos curioso de aplicaciones de agentes LLM. Los anteriores son solo algunos frameworks y aplicaciones de agentes LLM. En internet se mencionan varios más pero de seguir el informe se haría demasiado largo.

Problemática: Gestión eficiente de un sistema de transporte público en una ciudad grande.

Un sistema multiagente podría ser útil para gestionar y optimizar un sistema de transporte público en una ciudad grande. Los agentes podrían trabajar juntos para minimizar los tiempos de viaje, equilibrar la demanda y la oferta de vehículos, y mejorar la satisfacción general de los pasajeros.

Agentes involucrados en esta tarea:

1. Agente de Rutas: Agente responsable de gestionar las rutas de los vehículos de transporte público. Utiliza datos en tiempo real sobre el tráfico y la demanda de pasajeros para optimizar las rutas y minimizar los tiempos de viaje.

2. Agente de Horarios: Agente se encarga de crear y ajustar los horarios de los vehículos de transporte público. Trabaja en conjunto con el Agente de Rutas para asegurar que los vehículos estén disponibles cuando y donde se necesiten.

3. Agente de Mantenimiento: Este agente programa el mantenimiento regular de los vehículos para asegurar su funcionamiento óptimo. También maneja cualquier problema o avería imprevista.

4. Agente de Información al Pasajero: Este agente proporciona información en tiempo real a los pasajeros sobre los horarios de los vehículos, las rutas y cualquier cambio o retraso. Esto podría hacerse a través de una aplicación móvil
.

5. Agente de Control de Tráfico: Este agente monitorea el tráfico en la ciudad y proporciona esta información a los otros agentes. Esto permitiría al Agente de Rutas ajustar las rutas en tiempo real para evitar el tráfico pesado y al Agente de Horarios ajustar los horarios en consecuencia.

Bibliografía:

https://medium.com/@aydinKerem/what-is-an-llm-agent-and-how-does-it-work-1d4d9e4381ca

https://developer.nvidia.com/blog/introduction-to-llm-agents/

https://huggingface.co/blog/open-source-llms-as-agents
https://python.langchain.com/docs/modules/agents/

https://chatdev.ai/

https://medium.com/latinxinai/agentes-aut%C3%B3nomos-y-simulaciones-en-llm-un-vistazo-a-autogpt-babyagi-camel-y-generative-agents-bdb0bbfcddac

https://aplicaciones.ai/baby-agi/