Tecnicatura Universitaria en Inteligencia Artificial

Procesamiento del Lenguaje Natural (IA4.2)

Trabajo Práctico N° 2

Docentes:

- D'Alessandro, Ariel
- Geary, Alan
- Leon Cavallo, Andrea
- Manson, Juan Pablo

Alumno:

- Aguirre, Fabian A-4516/1

Año: 2023

ENUNCIADO

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

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.

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.

DESARROLLO

Ejercicio 1:

A continuación, se presenta la implementación de un chatbot experto utilizando la técnica RAG (Retrieval Augmented Generation). El chatbot se enfoca en temas relacionados con salud y alimentación, y utiliza diversas fuentes de conocimiento, como documentos de texto y datos numéricos en formato tabular, almacenados en una base de datos vectorial y una base de datos de grafos.

Los archivos utilizados son:

alimentacion-basada-en-plantas.pdf
documento-entornos-escolares-saludables.pdf
etica_y_trasplante.pdf
investigacion-sodio.pdf
recomendaciones-chagas.pdf
tasa_vih_jurid.csv

Descripción del código:

1. Descarga de Datos

Se utiliza la biblioteca gdown para descargar archivos desde Google Drive, y se organiza la estructura de carpetas. Se eliminan carpetas temporales después de la descarga. En la descarga se incluye el archivo .env que contiene el token de HuggingFace que se utilizará posteriormente.

```
url = 'https://drive.google.com/drive/folders/1xLQnsyMX8vCWplS3RXqkVcdRdn42GhSA?usp=sharing'

gdown.download_folder(url, quiet=True, output='tp2-nlp')

carpeta_destino = 'documentos'
if not os.path.exists(carpeta_destino):
    os.makedirs(carpeta_destino)

carpeta_origen = 'tp2-nlp/tp2-nlp-documentos'
for filename in os.listdir(carpeta_origen):
    ruta_origen = os.path.join(carpeta_origen, filename)
    ruta_destino = os.path.join(carpeta_destino, filename)
    shutil.move(ruta_origen, ruta_destino)

shutil.rmtree(carpeta_origen)

carpeta_origen = 'tp2-nlp'
carpeta_destino = '/content/'
for filename in os.listdir(carpeta_origen):
    ruta_origen = os.path.join(carpeta_origen, filename)
    ruta_destino = os.path.join(carpeta_destino, filename)
    shutil.move(ruta_origen, ruta_destino)

shutil.rmtree(carpeta_origen)

sys.path.append('/content/')

print("Archivos descargados con éxito.")
```

2. Procesamiento de Texto

Se utiliza la biblioteca fitz para extraer texto de archivos PDF. El texto se divide en fragmentos utilizando la clase RecursiveCharacterTextSplitter de langchain. Se realiza una limpieza básica del texto, convirtiéndolo a minúsculas y eliminando caracteres especiales.

3. Generación de Embeddings y Almacenamiento en ChromaDB

Se emplea el modelo de embeddings universal-sentence-encoder-multilingual de TensorFlow Hub para vectorizar el texto. Los embeddings junto con otros metadatos se almacenan en una base de datos ChromaDB.

```
#Creación de la base con CrhomaDB
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-multilingual/3")

client = chromadb.Client()
collection = client.get_or_create_collection("all-my-documents")

pdf_directory = "documentos"

pdf_files = [f for f in os.listdir(pdf_directory) if f.endswith(".pdf")]

textos = []
ids_textos = []
fuentes = []

def clean_text(text):
    cleaned_text = text.lower()
    cleaned_text = re.sub(r'[^\w\s.,]', '', cleaned_text)
    return cleaned_text

splitter = RecursiveCharacterTextSplitter(chunk_size=80, chunk_overlap=10)
```

```
for i, pdf_file in enumerate(pdf_files, start=1):
    pdf_path = os.path.join(pdf_directory, pdf_file)

    with fitz.open(pdf_path) as doc:
        text = ""
        for page_num in range(doc.page_count):
            page = doc[page_num]
            text += page.get_text()

    split_texts = splitter.split_text(text)

    clean_texts = [clean_text(sentence) for sentence in split_texts]

    textos.extend(clean_texts)
    ids_textos.extend([f"doc{i}_{j}" for j in range(1, len(clean_texts) + 1)])
    fuentes.extend([f"fuente{i}" for _ in range(len(clean_texts))])

embeddings = embed(textos).numpy().tolist()

collection.add(
    documents=textos,
    metadatas=[{"source": fuente} for fuente in fuentes],
    ids=ids_textos,
    embeddings=embeddings)
```

4. Creación de un Grafo RDF

Se crea un grafo RDF con los datos del pdf acerca de la prevención contra el chagas.

read_pdf(path):

- Esta función utiliza PyMuPDF (fitz) para leer un archivo PDF y extraer texto.
- Luego, divide el texto en fragmentos usando RecursiveCharacterTextSplitter.
- Realiza una limpieza simple del texto convirtiéndolo a minúsculas y eliminando caracteres no alfanuméricos.
- Devuelve una lista de fragmentos de texto limpios.

extract_entities_relations(doc):

- Utiliza spaCy para procesar un documento y extraer entidades y relaciones gramaticales.
- Identifica el sujeto, objeto y relación principal en una oración.
- Maneja preposiciones y sus objetos.
- Devuelve tuplas de entidades y relaciones.

Creación del Grafo RDF:

- Carga un modelo de spaCy para procesamiento del lenguaje natural.
- Utiliza RecursiveCharacterTextSplitter para dividir el texto del PDF en fragmentos.
- Limpia cada fragmento de texto y extrae entidades y relaciones utilizando extract_entities_relations.
- Crea un grafo RDF utilizando la librería rdflib.
- Asigna un espacio de nombres "http://chagas.org/" para los recursos del grafo.
- Agrega triples (sujeto, relación, objeto) al grafo para cada oración procesada.
- Serializa el grafo en formato XML y lo guarda en un archivo llamado "graph.rdf".

```python
#Creación del grafo
nlp = spacy.load("es_core_news_md")
spl = RecursiveCharacterTextSplitter(chunk_size=250, chunk_overlap=50)

def clean_text_pdf(text):
    cleaned_text = text.lower()
    cleaned_text = re.sub(r'[^\w\s]', '', cleaned_text)
    return cleaned_text

def read_pdf(path):
    with fitz.open(path) as doc:
        text = ""
        for page_num in range(doc.page_count):
            page = doc[page_num]
            text += page.get_text()
    split_texts = spl.split_text(text)
    clean_texts = [clean_text_pdf(sentence) for sentence in split_texts]
    return clean_texts

def extract_entities_relations(doc):
    subject = ""
    obj = ""
    relation = ""
    prep_obj = ""

    for token in doc:
        if "subj" in token.dep_:
            subject = token.text
        if "obj" in token.dep_:
```

```python
            obj = token.text
        if token.dep_ == "ROOT":
            relation = token.text
            for child in token.children:
                if child.dep_ == "nmod":
                    prep_obj = child.text
        if token.dep_ == "prep":
            for child in token.children:
                prep_obj = child.text
    if prep_obj:
        obj = prep_obj
    return (subject, obj), [relation]

# Crear un grafo RDF
g = Graph()
n = Namespace("http://chagas.org/")

chagas_pdf_path = "recomendaciones-chagas.pdf"
text_pdf_chagas = read_pdf(chagas_pdf_path)

for sentence in text_pdf_chagas:
    doc = nlp(sentence)
    entities, relations = extract_entities_relations(doc)
    print(entities, relations)
    if entities[0] and entities[1] and relations:
        subject_name, object_name = entities
        relation_name = relations[0]
        subject = URIRef(n + subject_name)
        predicate = URIRef(n + relation_name)
        obj = URIRef(n + object_name)
```

```python
        subject = URIRef(n + subject_name)
        predicate = URIRef(n + relation_name)
        obj = URIRef(n + object_name)
        g.add((subject, predicate, obj))

rdf_output = g.serialize(format='xml')

with open("graph.rdf", "w") as file:
    file.write(rdf_output)
```

5. Chatbot

Se implementa la lógica del chatbot utilizando varias funciones. El chatbot utiliza tanto la base de datos ChromaDB como el grafo RDF y datos en forma tabular para responder preguntas del usuario. Se utiliza un clasificador para determinar de que base de datos obtener el contexto.

Funciones Principales:

read_csv(path_csv):

- Lee un archivo CSV y construye un contexto CSV que contiene información sobre las tasas de VIH por jurisdicción.

read_graph(path_graph, query_user):

- Lee un grafo RDF desde un archivo y calcula la similitud de coseno entre las entidades extraídas de la pregunta del usuario y las entidades en el grafo RDF.

query_chromadb(query_embed):

- Realiza una consulta a una base de datos (ChromaDB) utilizando embeddings y devuelve los resultados.

classifier(query_str: str) -> str:

- Clasifica la pregunta del usuario para determinar si la respuesta proviene de ChromaDB, el grafo RDF o el contexto CSV.

- Utiliza embeddings y similitud de coseno para calcular la similitud entre la pregunta y los datos de cada fuente.

- Devuelve el contexto correspondiente con la mayor similitud.

zephyr_chat_template(messages, add_generation_prompt=True):

- Define un formato para los mensajes del chatbot.

generate_answer(prompt: str, max_new_tokens: int = 768) -> None:

- Genera una respuesta utilizando el modelo de lenguaje Zephyr de Hugging Face.

prepare_prompt(query_str, context_prompt):

- Prepara el prompt que se usará para generar la respuesta, incorporando el contexto proporcionado.

Bucle Principal:

- El chatbot entra en un bucle donde el usuario puede ingresar preguntas.
- Cada pregunta se clasifica utilizando la función classifier para determinar la fuente de la respuesta (CSV, ChromaDB, o grafo RDF).
- Se genera un prompt final que incluye la pregunta del usuario y el contexto relevante.
- La respuesta se genera utilizando el modelo Zephyr de Hugging Face.

```python
#CHATBOT
def read_csv(path_csv):
    csv_path = path_csv
    df = pd.read_csv(csv_path, encoding='utf-8-sig')
    context_csv = ""
    for _, row in df.iterrows():
        context_csv += f"{row['jurisdiccion']} tiene una tasa de VIH de {row['tasa_vih']}\n"
    return context_csv


def read_graph(path_graph, query_user):
    nlp = spacy.load("es_core_news_md")

    n = Namespace("http://chagas.org/")

    g = Graph()
    g.parse(path_graph, format="xml")

    def cosine_similarity_entities(query_embeddings, entity_embedding):
        similarity_score = cosine_similarity([query_embeddings], [entity_embedding])[0][0]
        return similarity_score

    def are_entities_in_graph(query_embedding_0, query_embedding_1):
        max_similarity_score = 0.0
        matching_entity = None

        for node in g.all_nodes():
            node_name = node.toPython()
```

```
            node_embeddings = embed([node_name]).numpy().tolist()[0]

            similarity_score_0 = cosine_similarity_entities(query_embedding_0, node_embeddings)
            similarity_score_1 = cosine_similarity_entities(query_embedding_1, node_embeddings)

            if max(similarity_score_0, similarity_score_1) > max_similarity_score:
                max_similarity_score = max(similarity_score_0, similarity_score_1)
                matching_entity = node_name

            return  max_similarity_score

    entities, _ = extract_entities_relations(nlp(query_user))

    query_embedding_0 = embed([n + entities[0]]).numpy().tolist()[0]
    query_embedding_1 = embed([n + entities[1]]).numpy().tolist()[0]

    similarity_graph = are_entities_in_graph(query_embedding_0, query_embedding_1)
    context_graph = ""
    for s, p, o in g:
        subject_name = s.split(n)[-1]
        predicate_name = p.split(n)[-1]
        object_name = o.split(n)[-1]
        context_graph += f"{subject_name} {predicate_name} {object_name}.\n"

    return similarity_graph, context_graph
```

```
def query_chromadb(query_embed):
    results = collection.query(
        query_embeddings=query_embed,
        n_results=20
    )
    return results


def classifier(query_str: str) -> str:
    query_embeddings = embed([query_str]).numpy().tolist()
    reference_text_csv = "VIH"
    reference_embedding_csv = embed([reference_text_csv]).numpy().tolist()
    max_cosine_similarity_csv = -1000

    for  text_segment_csv in query_str.split():
        segment_embedding_csv = embed([text_segment_csv]).numpy().tolist()
        similarity_score_csv = cosine_similarity(segment_embedding_csv, reference_embedding_csv)[0][0]

        if similarity_score_csv > max_cosine_similarity_csv:
           max_cosine_similarity_csv = similarity_score_csv

    path_csv_vih = "tasa_vih_jurid.csv"
    context_csv = read_csv(path_csv_vih)

    documents = query_chromadb(query_embeddings)
    diference_min_chromadb = min(abs(x - 1) for x in documents["distances"][0])
    context_chroma = "\n".join([f"{doc}" for documents_lista in documents["documents"] for doc in documents_lista])
    similarity_chroma = 1 - diference_min_chromadb
```

```
    path_graph_chagas = "graph.rdf"
    similarity_graph, context_graph = read_graph(path_graph_chagas,query_str)

    if similarity_chroma > max_cosine_similarity_csv and similarity_chroma > similarity_graph:
        return context_chroma
    elif similarity_graph > similarity_chroma and similarity_graph > max_cosine_similarity_csv:
        return context_graph
    else:
        return context_csv


def zephyr_chat_template(messages, add_generation_prompt=True):

    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>\n"
    template_str += "{% elif message['role'] == 'system' %}"
    template_str += "<|system|>{{ message['content'] }}</s>\n"
    template_str += "{% else %}"
    template_str += "<|unknown|>{{ message['content'] }}</s>\n"
    template_str += "{% endif %}"
    template_str += "{% endfor %}"
    template_str += "{% if add_generation_prompt %}"
    template_str += "<|assistant|>\n"
    template_str += "{% endif %}"
```

```
    template = Template(template_str)
    return template.render(messages=messages, add_generation_prompt=add_generation_prompt)



def generate_answer(prompt: str, max_new_tokens: int = 768) -> None:
    try:

        api_key = config('HUGGINGFACE_TOKEN')

        api_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-beta"

        headers = {"Authorization": f"Bearer {api_key}"}

        data = {
            "inputs": prompt,
            "parameters": {
                "max_new_tokens": max_new_tokens,
                "temperature": 0.7,
                "top_k": 50,
                "top_p": 0.95
            }
        }

        response = requests.post(api_url, headers=headers, json=data)
```

```
        respuesta = response.json()[0]["generated_text"][len(prompt):]

        return respuesta

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


def prepare_prompt(query_str, context_prompt):

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

    messages = [
        {
            "role": "system",
            "content": "Eres un asistente útil que siempre responde con respuestas veraces, útiles y basadas en hechos.",
        },
        {"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(context_prompt=context_prompt, query_str=query_str)},
    ]

    final_prompt = zephyr_chat_template(messages)
```

6. Interacción con el Usuario

Se proporciona un bucle de interacción donde el usuario puede ingresar preguntas y obtener respuestas del chatbot. El contexto se actualiza en cada iteración.

```
while True:
    user_query = input("Ingrese su consulta (o 'salir' para salir): ")
    if user_query.lower() == 'salir':
        break

    context_final = classifier(user_query)
    final_prompt = prepare_prompt(user_query, context_final)
    print('Respuesta:')
    print(generate_answer(final_prompt))
    print('------------------------------------------------------------')
```

7. Ejemplos de Preguntas

Se proporcionan ejemplos de preguntas que el chatbot puede manejar, relacionadas con tasas de VIH por provincia, procesos de trasplante de órganos, impactos del exceso de sodio en la alimentación, requerimientos para personas con alimentación vegetariana y recomendaciones contra el chagas.

```
#Ejemplos de preguntas
#¿Cuál es la tasa de VIH de Neuquen?
#¿Cómo es el procedimiento para una transplante de organos?
#¿Qué puede producir en el cuerpo el exceso de consumo de sodio en los alimentos?
#¿Cuáles son los requerimientos para personas con alimentación vegetariana?
#En pocas palabras ¿Cómo controlar el vector del chagas?
```

8. Temas Específicos del Chatbot

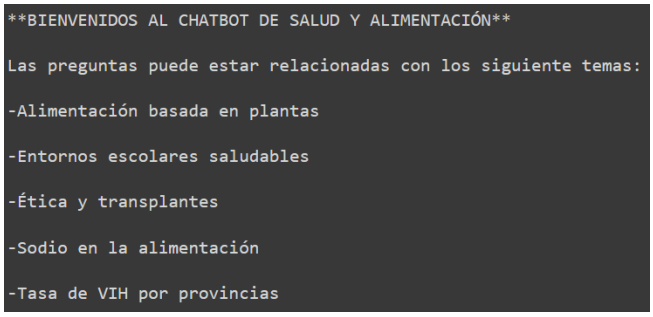
El chatbot se especializa en temas de salud y alimentación, como alimentación basada en plantas, entornos escolares saludables, ética y trasplantes, sodio en la alimentación, y tasas de VIH por provincias.

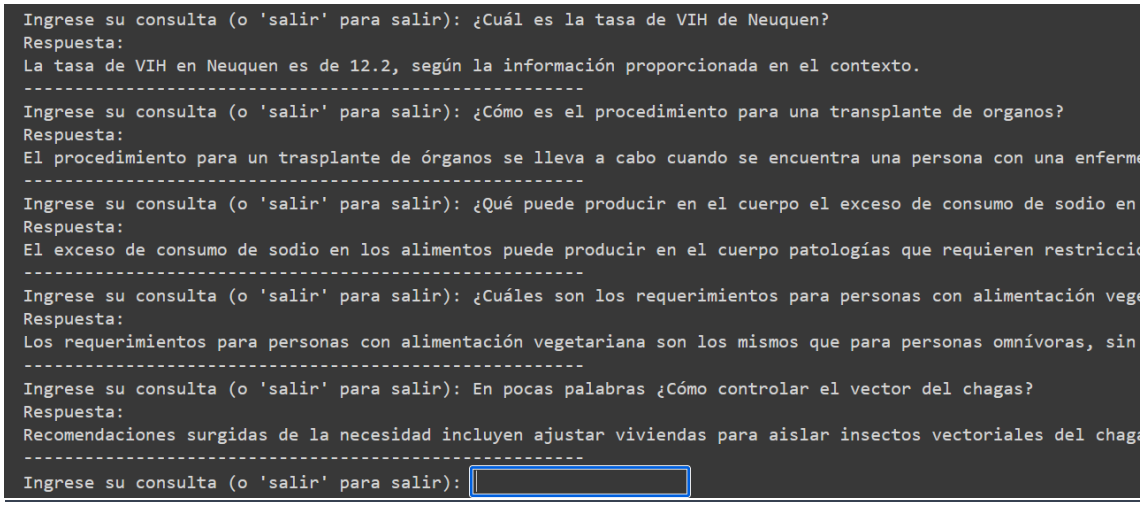
9. Uso de Modelos Externos

Se utiliza un modelo de lenguaje de Hugging Face (Zephyr) para la generación de respuestas.

10. Finalización del Proyecto

Se muestra un mensaje de bienvenida y se inicia el bucle de interacción con el usuario. El usuario puede ingresar "salir" para terminar la interacción.





Ejercicio 2:

Los Modelos de Lenguaje Grande (LLM) están desempeñando un papel importante en el desarrollo de futuras aplicaciones. Los LLM son muy buenos para comprender el lenguaje debido a la amplia capacitación previa que se ha realizado para los modelos básicos en billones de líneas de texto de dominio público. Métodos como el ajuste fino supervisado y el aprendizaje reforzado con retroalimentación humana (RLHF) hacen que estos LLM sean aún más eficientes para responder preguntas específicas y conversar con los usuarios.

Algunos ejemplos del uso de LLMs son:
Asistentes Virtuales: Aplicaciones como Siri, Google Assistant o Alexa utilizan agentes inteligentes basados en modelos de lenguaje para entender y responder preguntas habladas.

Chatbots en Servicio al Cliente: Empresas utilizan chatbots impulsados por modelos de lenguaje para responder a consultas de clientes en tiempo real.

Generación de Contenido Automático: Herramientas que utilizan modelos de lenguaje para generar automáticamente contenido escrito, como artículos de noticias, resúmenes o descripciones de productos.

Traducción Automática: Sistemas de traducción automática que emplean modelos de lenguaje para traducir texto entre diferentes idiomas.

Análisis de Sentimientos en Redes Sociales: Agentes inteligentes que utilizan modelos de lenguaje para analizar y comprender el sentimiento detrás de publicaciones en redes sociales.

Generación de Texto Creativo: Aplicaciones que generan poesía, historias cortas u otros tipos de contenido creativo mediante modelos de lenguaje.

Corrección Gramatical Automática: Herramientas que utilizan modelos de lenguaje para corregir errores gramaticales y ortográficos en texto.

Asistentes de Escritura: Plataformas que ofrecen sugerencias de escritura y mejoras en el contenido mediante agentes inteligentes basados en modelos de lenguaje.

Generación de Código Automática: Herramientas que asisten en la generación de código fuente a partir de descripciones en lenguaje natural.

Asistentes de Salud Virtual: Agentes inteligentes que proporcionan información de salud, responden preguntas y ofrecen recomendaciones basadas en modelos de lenguaje

Los agentes autónomos, en comparación con simples modelos de chat basados en LLMs (Modelos de Lenguaje con Aprendizaje Profundo), ofrecen mejoras significativas en términos de capacidad para realizar tareas más complejas y adaptarse a entornos dinámicos.

El uso de agentes autónomos se está popularizando en los últimos años, estos son algunos ejemplos:

Drones de Entrega Autónomos: Proyecto Wing (Alphabet, empresa matriz de Google)

Vehículos Autónomos: Waymo (también pertenece a Alphabet)

Tesla Autopilot: Robots de Almacén Autónomos: Kiva Systems (ahora propiedad de Amazon Robotics)

Sistemas de Navegación Autónoma: Sistemas de navegación de robots submarinos para exploración oceánica.

Sistemas de Monitorización Ambiental: Estaciones meteorológicas o de monitoreo de calidad del aire autónomas.

Sistemas de Seguridad Autónomos: Sistemas de vigilancia autónoma basados en IA.

Robots Agrícolas Autónomos: Robótica agrícola para tareas como la cosecha automatizada.

Robots de Exploración Espacial: Rover Curiosity de la NASA en Marte.

Algunos de los proyectos que utilizan LLMs son:

AutoGPT: AutoGPT es un agente autónomo que desarrolla un LLM subyacente para comprender el objetivo que se le ha asignado y trabajar para lograrlo. AutoGPT genera una lista de tareas que cree que necesita para cumplir con lo que le pediste, sin requerir más información o mensajes.

GPT-Engineer: Al igual que AutoGPT, GPT-Engineer es otro agente autónomo que utiliza LLM para comprender y trabajar hacia el objetivo asignado

BabyAGI: Este es otro proyecto que se centra en el desarrollo de agentes autónomos con habilidades de planificación a largo plazo y uso de memoria. BabyAGI busca crear agentes que puedan aprender y adaptarse a su entorno utilizando técnicas de aprendizaje por refuerzo.

También hay proyectos como CAMEL y Generative Agents se centran en la creación de entornos de simulación específicos para agentes autónomos.

Sistema Multiagente para Optimizar el Tráfico Urbano

Problemática: El tráfico urbano ineficiente afecta la calidad de vida de los habitantes, generando congestiones, contaminación ambiental y aumentando el tiempo de viaje. Además, vehículos de prioridad como ambulancias, bomberos o vehículos policiales se ven demorados por el caos del tránsito. La problemática central radica en la falta de coordinación entre los diferentes elementos del tráfico, como semáforos, señales y sistemas de transporte público.

Objetivo: Desarrollar un sistema multiagente que permita la gestión coordinada y eficiente del tráfico urbano, reduciendo la congestión, mejorando los tiempos de viaje y reduciendo los tiempos de vehículos con prioridad.

Desarrollo del Sistema Multiagente:

- **Agente de Central Coordinador:**
  - Coordina la operación de todo el sistema multi agente.
  - Se comunica con todos los agentes incluyendo la información obtenida por el agente usuario.

  Este agente se encarga de coordinar a todos los otros agentes para una respuesta óptima. Esta en constante comunicación con todo el sistema por lo que le permite tener una visión global.

- **Agente Usuario:**
    - Establece la comunicación con el usuario administrador del sistema

    Este agente establece una comunicación en lenguaje natural con el administrador para mostrar reportes, indicadores del estado general del tránsito. Además permite la intervención del usuario en todo momento de cualquier parte del sistema

- **Agente de Control de Semáforos:**
    - Utiliza datos en tiempo real para ajustar la duración de los semáforos.
    - Coordinación con los distintos semáforos para establecer "onda verde"
    - Ajusta los tiempos de los semáforos para favorecer los vehículos con prioridad

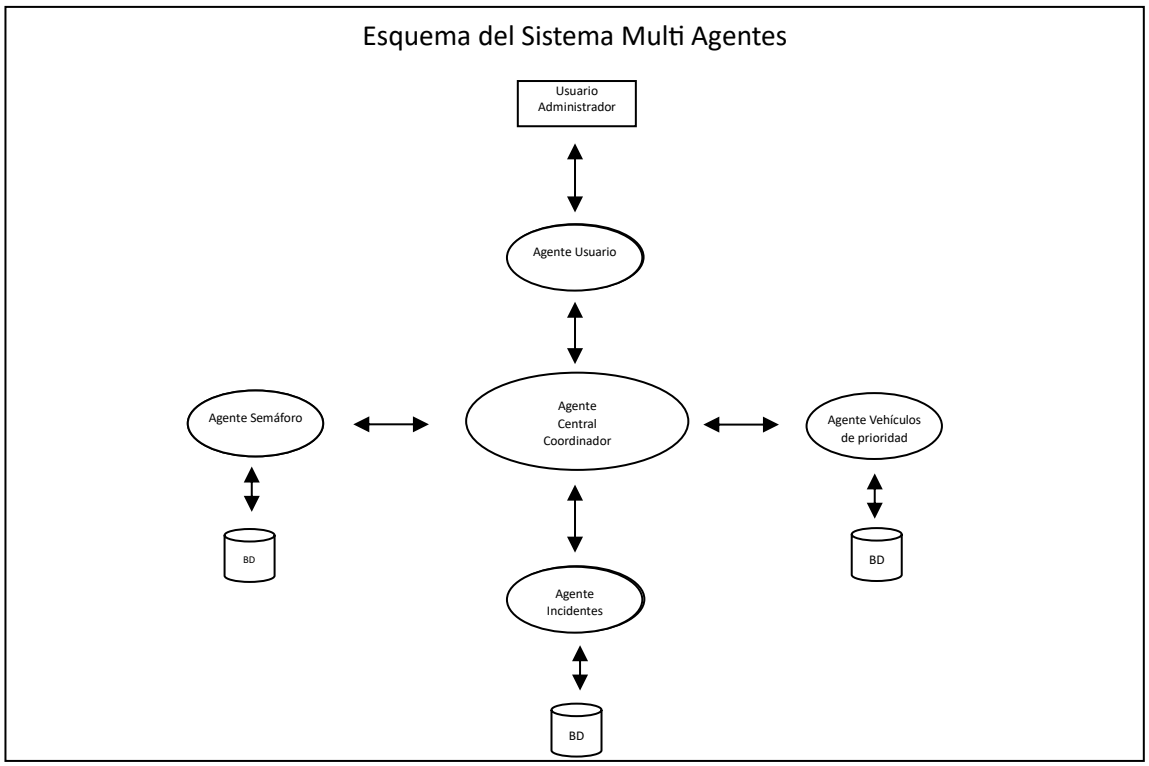
    Este agente se nutre de los datos almacenado en una base de datos (recopilados a través de múltiples sensores) de todos los datos referidos a los semáforos. Toma decisiones referidas a la operatoria todos los semáforos en general.

- **Agente de vehículos de prioridad:**
    - Optimiza las rutas para vehículos de prioridad

    Este agente se nutre de los datos almacenado en una base de datos (recopilados a través de múltiples sensores y datos de otras fuentes) de todos los datos referidos a los vehículos de prioridad. Toma decisiones referidas al aviso de los conductores en general y a los de los vehículos de prioridad.

- **Agente de Respuesta a Incidentes:**
    - Detecta eventos como accidentes, obras viales u otros imprevistos y ajusta la circulación.

    Este agente se nutre de los datos almacenado en una base de datos (recopilados a través de múltiples sensores y cámaras) de todos los datos referidos a cualquier accidente o imprevistos. Toma decisiones referidas al aviso de los conductores y los otros agentes para ajustar la circulación.



Implementación de Tecnologías: El sistema utilizará tecnologías emergentes como el Internet de las Cosas (IoT), análisis de big data y aprendizaje automático para recopilar y analizar datos en tiempo real.

Beneficios Esperados:

- Reducción de tiempos de viaje para los ciudadanos.
- Minimización de emisiones de gases contaminantes.
- Mejora en la eficiencia de los vehículos de prioridad.

Ejemplo de una conversación entre agentes

Inicio de la Conversación:

El agente de Central Coordinador inicia la conversación:

¿Cómo está el tráfico hoy?"

Coordinación de Semáforos:

El Agente control de Semáforos responde: "El tráfico está pesado en el centro. Necesitamos mejorar la circulación."

El Agente de Central Coordinador toma acción:

Mensaje al Agente de Control de Semáforos: "Ajusta los semáforos en el centro para reducir la congestión. Prioriza los tiempos para los vehículos de emergencia."

Optimización de Rutas para Vehículos de Prioridad:

El Agente de Control de Semáforos informa al Agente de vehículos de prioridad:

Mensaje al Agente de vehículos de prioridad: "Se ha ajustado la duración de los semáforos en el centro. Optimiza las rutas para los vehículos de emergencia."

Respuesta a Incidentes:

El Agente de Respuesta a Incidentes detecta un accidente:

Mensaje al Agente de Central Coordinador y al Agente de vehículos de prioridad: "Accidente en la avenida principal. Ajusten las rutas y notifiquen a los conductores."

Reportes y Estado General:

El Agente Usuario solicita un reporte al Agente de Central Coordinador:

Mensaje al Agente Central Coordinador: "¿Puedes proporcionar un informe general del tráfico y los tiempos de viaje?"

Cierre de Conversación:

El Agente Usuario decide intervenir manualmente:

Mensaje al Agente Central Coordinador: "Voy a ajustar manualmente algunos semáforos en el centro. Avísame si hay alguna objeción."

Conclusiones: La implementación de un sistema multiagente para la gestión del tráfico en una ciudad inteligente es crucial para abordar la problemática del tráfico urbano. La coordinación entre diferentes agentes permite una respuesta dinámica a situaciones cambiantes, mejorando la eficiencia y la sostenibilidad del sistema de transporte urbano. La importancia de administrar de una forma óptima las rutas de los vehículos mejora enormemente la fluidez del tránsito

Fuentes:

https://en.wikipedia.org/wiki/Wing_%28company%29

https://es.wikipedia.org/wiki/Waymo

https://www.xataka.com/automovil/waymo-sera-la-nueva-compania-independiente-de-alphabet-google-encargada-de-coches-autonomos

https://www.tesla.com/autopilot

https://en.wikipedia.org/wiki/Amazon_Robotics

https://invdes.com.mx/tecnologia/los-robots-marinos-que-exploran-aguas-profundas-del-golfo-de-mexico/

https://www.smn.gob.ar/noticias/nuevas-estaciones-autom%C3%A1ticas

https://www.soyseguridadprivada.com/inteligencia-artificial-en-la-seguridad-privada-de-los-proximos-10-anos/

https://www.edsrobotics.com/blog/agricultura-automatizada-y-robotica-agricola/

https://spaceplace.nasa.gov/mars-curiosity/sp/

https://autogpt.net/

https://babyagi.org/docs/README-es.html

https://github.com/AntonOsika/gpt-engineer

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

Unidad 6 - Chatbots y Sistemas de Diálogo (TUIA-NLP)

Unidad 7 - Agentes Autónomos y Sistemas Inteligentes (TUIA-NLP)