TUIA

PROCESAMIENTO DEL LENGUAJE NATURAL

TRABAJO PRÁCTICO FINAL

Alumno:
Eugenio M. López

Febrero 2024

INTRODUCCIÓN

El siguiente proyecto se basa en dos ejercicios:

En el primero se debe crear un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation).

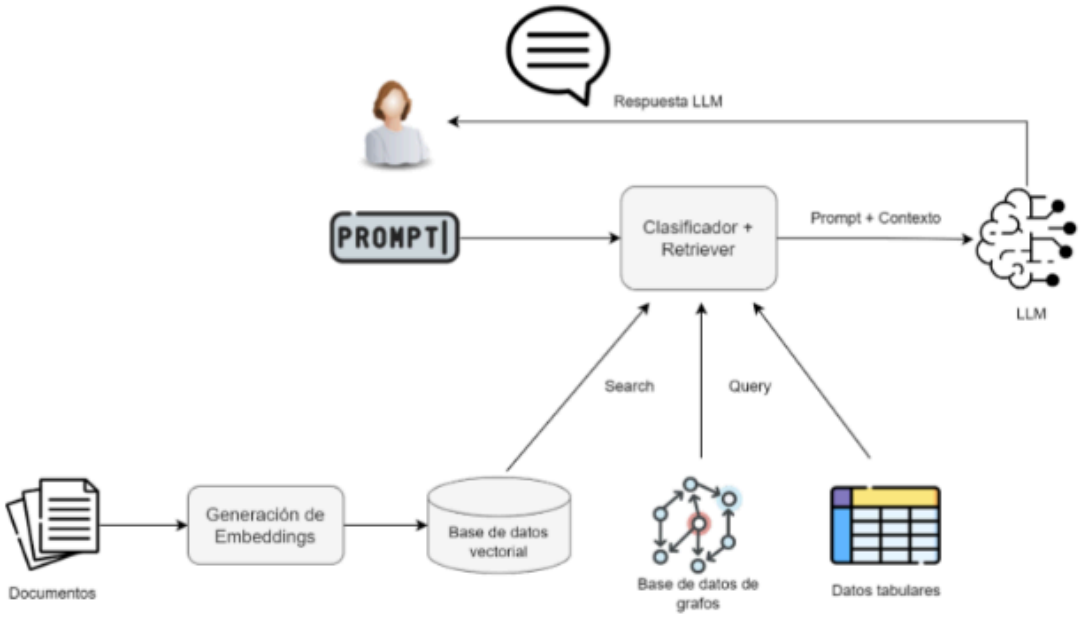
En este caso el tema elegido es el **cine**, es decir, el chatbot va a poder responder a aquellas preguntas que sean relacionadas a la historia del cine, información de películas, directores, etc.

Como fuentes de conocimiento se utilizarán las siguientes fuentes:

- Documentos de texto (PDF)
- Datos tabulares
- Base de datos de grafos

El sistema llevará a cabo una conversación en lenguaje español. El usuario podrá hacer preguntas, que el chatbot intentará responder a partir de datos de algunas de estas fuentes. El asistente podrá clasificar las preguntas, para saber qué fuentes de datos utiliza como contexto para generar la respuesta.

La ruta de nuestro chatbot será la siguiente:



Como segunda instancia, se realizará una investigación respecto al estado del arte de las aplicaciones actuales de **agentes inteligentes** usando modelos LLM libres, planteando una problemática a solucionar con un sistema multiagente.

Se desarrollará un informe con los resultados de la investigación y sus fuentes de información.

Enlace al repositorio del proyecto:

https://github.com/euglpz/nlp_chatbot

DESARROLLO

Para informar el desarrollo del proyecto, se va a ir comentando lo que se fue realizando paso a paso.

La totalidad del proyecto se realizó en un entorno **Google Colab**.

Ejercicio 1 - RAG

En primer lugar se realiza el tratamiento de los documentos de texto.

En este caso, se utilizan archivos PDF.

Carga de documentos PDF

El archivo PDF que se va a utilizar es una versión del libro titulado **"La Historia del Cine"** del autor Roman Gubern, del año 1969, actualizada al año 2014.

Para la carga de los documentos necesarios para el proyecto, se accede a un link a Google Drive que contiene una carpeta con todos los archivos a utilizar

```
# Link al drive con archivos sobre películas e historia del cine
url = https://drive.google.com/drive/folders/1no__wakSVZvBfctC0wy-65LmB79TfSi6?usp=drive_link
```

A partir de ese link, se descarga la carpeta y se mueven todos los archivos a una carpeta destino dentro del entorno colab.

```
carpeta_destino = 'data_cine'
```

Para la extracción del texto se crea una función llamada 'extraer_texto_pdf' a la cual se le pasa como parámetro la ruta para acceder a los archivos pdf.

Se utiliza la librería *pdfplumber* ya que se logra una extracción más estructurada del texto que usando otras librerías.

```
# Extracción de texto de PDF

import pdfplumber

def extraer_texto_pdf(ruta_pdf):
  with pdfplumber.open(ruta_pdf) as pdf:
    texto = ''
    for page in pdf.pages:
      texto += page.extract_text()
    return texto
```

Para acceder a la ruta de los archivos se busca dentro de los directorios y se obtiene la lista de archivos dentro de la carpeta data_cine.

Luego se filtra por terminación '.pdf' y por último se iteran los archivos contenidos en la carpeta y se llama a la función para realizar la extracción del texto.

```
carpeta_cine = 'data_cine'

# Obtener la lista de archivos en la carpeta 'data_cine'

archivos_en_carpeta = os.listdir(carpeta_cine)


# Filtra solo los archivos PDF

archivos_pdf = [archivo for archivo in archivos_en_carpeta if
archivo.endswith('.pdf')]


# Itera sobre los archivos PDF y extrae el texto

for archivo_pdf in archivos_pdf:

    ruta_completa = os.path.join(carpeta_cine, archivo_pdf)

    texto_extraido = extraer_texto_pdf(ruta_completa)


    print(f'\nTexto extraído de {archivo_pdf}:\n')

    print(texto_extraido)
```

Limpieza y acondicionamiento del texto extraído

Una vez obtenido el texto de los documentos, se procede a realizar la limpieza y acondicionamiento del mismo.

Se pasa todo el texto a minúsculas y se eliminan tildes, saltos de línea y caracteres especiales.

Para eliminar las tildes se utiliza la librería Unidecode.

```
# Texto en minúsculas
texto_extraido = texto_extraido.lower()

# Elimino tildes
from unidecode import unidecode
texto_extraido = unidecode(texto_extraido)

# Elimino caracteres especiales
import re
texto_extraido = re.sub(r'[^a-zA-Z0-9\s]', '', texto_extraido)
# Elimino los saltos de línea (\n)
texto_limpio = texto_extraido.replace("\n", " ")
```

Split del texto

Una vez extraído el texto, se realiza el split del mismo con el propósito de dividirlo en partes más pequeñas o chunks. Esta división facilita el procesamiento y análisis posterior del texto, ya que permite trabajar con unidades más manejables y específicas.

En este caso se va a realizar una **segmentación recursiva** ya que es útil en la identificación de palabras clave, la clasificación de textos y la extracción de información. Además, ayuda a mejorar la eficiencia del procesamiento de lenguaje natural al reducir la cantidad de texto que se debe analizar en cada paso.

Se utiliza **RecursiveCharacterTextSplitter** de la librería **LangChain.**

```
from langchain.text_splitter import RecursiveCharacterTextSplitter


text_splitter = RecursiveCharacterTextSplitter(
```

```
    chunk_size=1000,
    chunk_overlap=100,
    length_function=len,
    is_separator_regex=False,
)
splitted_text = text_splitter.split_text(texto_limpio)
```

Como parámetros se usan chunk_size=1000 y chunk_overlap=100 para obtener una granularidad justa del split, ni muy grande ni muy chica.

De este modo el total de chunks es de 1418 cada uno de un largo de unos 1000 caracteres aproximadamente.

Generación de embeddings

Para la generación de embeddings se va a utilizar el modelo BERT desarrollado por Google.

Es conveniente para este proyecto ya que es un modelo Contexto-Dependiente. Se basa en la arquitectura de "transformers", que utiliza mecanismos de atención para capturar el contexto en diferentes partes de un texto, es decir, genera embeddings contextuales,

```
from transformers import BertModel, BertTokenizer
import torch
# Cargar el modelo BERT pre-entrenado y el tokenizador
modelo_bert =
BertModel.from_pretrained('bert-base-multilingual-cased')
tokenizer =
BertTokenizer.from_pretrained('bert-base-multilingual-cased')

def obtener_embeddings(text):
    embeddings = []
    for fragmento in text:
            tokens = tokenizer(fragmento, truncation=True,
            padding=True, return_tensors='pt', max_length=512)
```

```
            outputs = modelo_bert(**tokens)
            embedding_vector =
            outputs.last_hidden_state.mean(dim=1).squeeze()
            embeddings.append(embedding_vector.tolist())
    return embeddings
# Generación de embeddings (20 min aprox)
embeddings = obtener_embeddings(splitted_text)
```

Base de datos vectorial

Para realizar el almacenamiento de los embeddings se utilizará **ChromaDB**

Se deberá crear una colección donde se almacenarán:

- embeddings
- documentos
- ids

```
# Generamos el cliente
import chromadb
chroma_client = chromadb.Client()
# Creamos la colección
collection = chroma_client.create_collection(name="cine_db")
if collection is not None:
    print("Collection already exists.")
else:
    collection = chroma_client.create_collection(name="cine_db")
# Generación de ids para almacenar en la chromaDB
ids = [f'id{i+1}' for i in range(len(splitted_text))]
len(ids)
# Se agregan los datos a la chromaDB
collection.add(
    embeddings=embeddings,
```

```
    documents=splitted_text,
    ids=ids)
```

Datos tabulares

Como dato tabular se cargará un archivo csv que tiene información del Top 1000 de películas mejor rankeadas por los suscriptores de la plataforma de datos e información de cine, análisis y críticas IMDB.

```
import pandas as pd
# Filtra solo los archivos csv de la carpeta 'data_cine'
archivos_csv = [archivo for archivo in archivos_en_carpeta if archivo.endswith('.csv')]
carpeta_cine = 'data_cine'
# Itera sobre los archivos csv
for archivo_csv in archivos_csv:
    ruta_completa = os.path.join(carpeta_cine, archivo_csv)
df_top_1000 = pd.read_csv(ruta_completa)
```

Se tendrán dos funciones para acceder a estos datos dependiendo la consulta que desee realizar el usuario:

En primer lugar:

```
def top_peliculas(df):
  '''
  Función que devuelve top de películas según puntaje
  '''
```

A esta función se le pasa como parámetro el DataFrame cargado y filtra las películas por puntaje, devolviendo el top de películas que desee el usuario.

El tamaño del top de películas lo define el usuario a través de un prompt input, por ejemplo top 3, top 5, top 10, etc.

En segundo lugar:

```
def encontrar_peliculas_por_puntaje(df):
```

```
  '''
  Función para encontrar películas dependiendo el puntaje
solicitado por el usuario
  '''
```

A esta función se le pasa como parámetro el DataFrame cargado y devuelve todas aquellas películas con el puntaje solicitado por el usuario.

Por ejemplo: puntaje deseado: 9

Devolverá todas las películas con puntaje 9 del DataFrame.

Base de datos de Grafos

Para la base de datos de Grafos se utilizará Wikidata. Es una base de datos gratuita, colaborativa y multilingüe, que recopila datos estructurados para brindar soporte a proyectos de Wikimedia, como Wikipedia.

Además se usa la librería SPARQLWrapper para realizar las consultas a la base de datos en lenguaje SPARQL.

```
# Función para obtener información de películas de la Base de
datos de Grafos
# En este caso se utiliza Wikidata
def obtener_info_pelicula(titulo_pelicula, ano_estreno=None):
    '''
    Función que realiza una consulta a base de datos de grafos
Wikidata
    devuelve información sobre película solicitada como parámetro.
    '''
```

Como parámetros se le pasa el título de la película que se quiere consultar y como opcional el año, si es que hay dos películas con el mismo título y distinto año.

Clasificador + Retriever

Se utilizarán plantillas Jinja para guardar información de formato.

Para el modelo de generación de texto como chat se usará ***"HuggingFaceH4/zephyr-7b-beta"***, a través de su API en Hugging Face.

Este modelo está entrenado para actuar como asistente útil.

Se tienen dos funciones, la primera:

```
def zephyr_chat_template(messages, add_generation_prompt=True):
```

Para definir los templates Jinja.

La segunda:

```
def generar_respuesta(prompt: str, api_key, max_new_tokens: int = 768) -> None:
    '''
    Función que llama al Modelo de HuggingFace y elige la fuente que considera correcta para cada pregunta.
    Parámetros: prompt  --> la pregunta que queremos que el modelo responda
                api_key --> nuestra clave API de HuggingFace
    '''
```

Para llamar al modelo de HuggingFace y elegir la fuente que se considere correcta para cada pregunta.

El parámetro api_key será nuestro token de HuggingFace.

Para el diseño y formato de prompts, es decir para darle las indicaciones sobre preguntas y respuestas al asistente, se utilizaron diferentes few-shots dependiendo las posibles preguntas que el usuario podría realizar.

Algunos ejemplos:

```
{
"role": "user",
      "content": "Brindame información de la película Taxi Driver. ¿Quién la dirigió y de qué año es?"
},
{
```

```
 "role": "assistant",
 "content": "Película: [Taxi Driver]"
},
{
 "role": "user",
 "content": "¿Cuáles fueron las primeras películas sonoras de la historia?"
 },
 {
 "role": "assistant",
 "content": "Historia cine"
  }
```

Ejemplo de análisis de preguntas utilizando el modelo:

```
prompt1 = 'Dame información de la película Amadeus. ¿De qué año es? ¿Ganó el Oscar a mejor película?'
print(prompt1)
print('Respuesta:')
answer1 = generar_respuesta(prompt1, api_key=api_key)
```

```
Respuesta:
Película: [Amadeus]
Amadeus fue estrenada en el año 1984. En la edición de los Premios de la Academia de ese año, la película obtuvo 8 estatuillas, incluyendo las categorías de Mejor Película, Mejor Director (Milos Forman), Mejor Guion Adaptado (Peter Shaffer), Mejor Actor (F. Murray Abraham), Mejor Diseño de Producción, Mejor Diseño de Vestuario, Mejor Maquillaje y Mejor Sonido.
```

Obtener contexto para clasificar preguntas

Para saber qué fuentes de datos utilizar como contexto para generar una respuesta, se va a pasar la información obtenida del clasificador a cada fuente, según corresponda.

Primero, se cambia el formato de la respuesta para que solamente nos devuelva lo que nos interesa y acceder de una forma más precisa al contexto.

Esto se hace a partir de la función: `formatear_respuesta(texto)`

```
print(f'Pregunta 1: {prompt1}')
respuesta_limpia1 = formatear_respuesta(answer1)
print(respuesta_limpia1)
```

```
Pregunta 1: Dame información de la película Amadeus. ¿De qué año es? ¿Ganó el Oscar a mejor película?
['película', 'Amadeus']
```

Luego, con la siguiente función: `devolver_contexto(respuesta_LLM, prompt)` podemos obtener el contexto de cada respuesta a nuestras preguntas.

Como parámetros se le pasan:

- la respuesta limpia obtenida a partir de la función ***formatear_respuesta***()
- el prompt, es decir la pregunta que hayamos hecho

```
# Ejemplo de uso
contexto1, fuente1 = devolver_contexto(respuesta_limpia1, prompt1)
print(contexto1)
print(fuente1)
```

Nos devuelve:

```
{'titulo': 'Amadeus', 'anioEstreno': '1984', 'director': 'Miloš Forman', 'actores_principales': ['F. Murray Abraham', 'Tom Hulce', 'Elizabeth Berridge'], 'genero': 'drama film', 'ganadorOscar': True}
Fuente: Base de datos de grafos
```

A continuación se prepara el prompt en estilo QA.

A su vez, se agrega el contexto y la fuente de información a la que accede el asistente para responder la pregunta.

Para ello se instancia esta nueva función:

```
def preparar_prompt(prompt: str, context: str, api_key,
max_new_tokens: int = 768) -> None:
```

Ejemplo de uso:

```
respuesta_final = preparar_prompt(prompt1, contexto1,
api_key=api_key)
print(f'Prompt: {prompt1}\n')
print('------------------------------------------------------------
------------')
print(f'Contexto: {contexto1}')
print(fuente1)
print('------------------------------------------------------------
')
print(respuesta_final)
```

Nos devuelve:

```
Prompt: Dame información de la película Amadeus. ¿De qué año es?
¿Ganó el Oscar a mejor película?

----------------------------------------------------------------

Contexto: {'titulo': 'Amadeus', 'anioEstreno': '1984', 'director':
'Miloš Forman', 'actores_principales': ['F. Murray Abraham', 'Tom
Hulce', 'Elizabeth Berridge'], 'genero': 'drama film',
'ganadorOscar': True}


Fuente: Base de datos de grafos

----------------------------------------------------------------

La película Amadeus estrenó en el año 1984 y si, ganó el Oscar a
la mejor película.
```

Chatbot especializado en cine

Para simplificar el acceso al asistente, se unen todas las funciones anteriormente instanciadas, en una sola función:

```
def chatbot(prompt, api_key):
    '''
    Función para acceder al funcionamiento del chatbot
    Llama a las funciones:
            generar_respuesta()
            formatear_respuesta()
            devolver_contexto()
            preparar_prompt()

    Parámetros: prompt --> la pregunta que queremos que el chatbot nos responda
    '''
```

Ejemplo de funcionamiento del chatbot con una pregunta en específico:

```
prompt = '¿Quién dirigió la película The Dark Knight? Ganó el Oscar? QUien la protagoniza?'
print(f'Pregunta:\n{prompt}')
print('------------------------------------------------------------')
respuesta, fuente, contexto = chatbot(prompt, api_key)
# Imprimimos el resultado
print('------------------------------------------------------------')
print(fuente)
if fuente != 'Error al identificar la fuente':
```

```
        print('-----------------------------------------------------')
print('-----------------------------------------------------------------')
print(f'Respuesta:\n{respuesta}')
```

Finalmente, nos devuelve:

```
Pregunta:

¿Quién dirigió la película The Dark Knight? Ganó el Oscar? QUien la
protagoniza?

-------------------------------------------------------------------------

Película: [The Dark Knight]

El director de la película The Dark Knight es Christopher Nolan. La
película no ganó el Oscar a la mejor película, sin embargo, el actor
Heath Ledger, que interpretó al personaje de Joker en la película, ganó
el Oscar a Mejor Actor de Reparto por su actuación en la misma. Los
protagonistas principales de la película son Christian Bale como Batman
/ Bruce Wayne, y Aaron Eckhart como Harvey Dent / Two-Face.

-------------------------------------------------------------------------

Fuente: Base de datos de grafos

-------------------------------------------------------------------------

Contexto:

{'titulo': 'The Dark Knight', 'anioEstreno': '2008', 'director':
'Christopher Nolan', 'actores_principales': ['Christian Bale', 'Morgan
Freeman', 'Gary Oldman'], 'genero': 'drama film', 'ganadorOscar':
False}

-------------------------------------------------------------------------

Respuesta:

El cineasta que dirigió la película The Dark Knight se llama
Christopher Nolan. No, la película no ganó el Oscar. Los actores
```

principales que la protagonizan son Christian Bale, Morgan Freeman y Gary Oldman.

CONCLUSIÓN

En resumen, el trabajo práctico concluye con la implementación de un asistente virtual especializado en cine, respaldado por una infraestructura de bases de datos que incluye sistemas vectoriales, de grafos y tabulares.
La elección de estas bases de datos ha permitido una gestión eficiente de la información y un desempeño optimizado del asistente.

Ejercicio 2 - Agentes

¿Qué es un Agente Inteligente y un Sistema Multiagente?

Un **agente inteligente** es un sistema perceptivo que puede interpretar y procesar la información que recibe de su entorno y actuar en consecuencia de acuerdo a los datos que obtiene y procesa. Su forma de actuación es lógica y racional basándose en el comportamiento normal de un sistema concreto. Utiliza sensores para recibir la información y actuadores para ejercer sus funciones.

Un **sistema multiagente** es un sistema compuesto por múltiples agentes inteligentes que interactúan entre ellos. Los sistemas multiagente pueden ser utilizados para resolver problemas que son difíciles o imposibles de resolver para un agente individual.

Los ámbitos en los que la investigación de sistemas multiagente puede ofrecer un enfoque adecuado incluyen, por ejemplo, el comercio en línea, la respuesta a desastres y el modelado de estructuras sociales.

Tipos de agentes:

- **Agente de reactivo simple**: Cuando una percepción coincide con una regla programada, el agente responde según la forma en la que fue programado. (Condición -> acción)
- **Agente reactivo basado en modelo**: Puede simular su acción de respuesta para estudiar su comportamiento y sus consecuencias en el espacio de actuación.
- **Agente basado en metas**: Combina los dos tipos de agentes previos. Tiene un objetivo concreto y busca la vía más óptima y planifica las acciones para cumplir su propósito.
- **Agente basado en utilidad**: Puede medir el valor de su comportamiento en el cumplimiento de las metas establecidas. Garantiza alta calidad en sus acciones.
- **Agente que aprende**: Aprende de sus acciones mientras está en funcionamiento. Tiene la capacidad de interactuar en entornos que no conoce.
- **Agente de consulta**: Responde consultas por parte de las personas que interactúan con este sistema. Crea varios agentes y divide la pregunta del usuario para dar una respuesta adecuada. En caso de que los agentes no puedan responder la pregunta, crea más agentes y busca en más bases de datos para dar una solución.

Estado del arte de agentes inteligentes usando modelos LLM libres:

Los modelos de lenguaje basados en aprendizaje profundo, como GPT, Perplexity y PaLM, son una de las tecnologías más avanzadas en el procesamiento del lenguaje natural (PLN) y se utilizan en una variedad de aplicaciones de agentes inteligentes. Son conocidos por su capacidad para generar texto similar al humano y realizar

tareas de PLN, como traducción automática, generación de texto y respuesta a preguntas.

Sin embargo, es importante tener en cuenta que los modelos LLM (Large Language Models) libres, como los de OpenAI (GPT), requieren un alto costo computacional y de recursos de entrenamiento, lo que limita su disponibilidad para todos los desarrolladores. A pesar de esto, han surgido varias aplicaciones de agentes inteligentes que aprovechan estos modelos LLM libres en diversas áreas:

Asistentes Virtuales: Los modelos LLM libres se utilizan para crear asistentes virtuales avanzados que pueden entender consultas complejas, realizar tareas específicas y brindar respuestas contextualmente relevantes.

Chatbots: Los chatbots impulsados por modelos LLM libres son capaces de mantener conversaciones más naturales y significativas con los usuarios, ya que tienen una comprensión más profunda del lenguaje natural.

Generación de Texto: Los modelos LLM libres pueden generar texto de manera creativa y coherente en una variedad de estilos y temas. Se utilizan en aplicaciones de redacción automática, creación de contenido, resumen de documentos y más.

Análisis de Sentimientos y Opiniones: Estos modelos pueden analizar grandes volúmenes de texto para identificar tendencias, opiniones y emociones. Se utilizan en aplicaciones de análisis de sentimientos en redes sociales, revisión de productos, encuestas en línea y más.

Traducción Automática: también se utilizan en sistemas de traducción automática para mejorar la precisión y la fluidez de las traducciones entre diferentes idiomas.

Es importante destacar que el desarrollo y la implementación de aplicaciones de agentes inteligentes utilizando modelos LLM libres deben considerar aspectos éticos, como la privacidad de los datos, el sesgo en el lenguaje y la transparencia en el uso de la inteligencia artificial.

Problemática a solucionar con un sistema multiagente:

Mejorar la experiencia del usuario y la eficiencia en la gestión de tareas mediante un asistente virtual en un entorno de trabajo remoto.

La idea es que un asistente virtual pueda ayudar a los empleados a organizar sus agendas, acceder a recursos y realizar tareas rutinarias de manera más eficiente.

Agentes involucrados:

Empleados: son los usuarios que utilizan el asistente virtual para gestionar sus tareas y solicitar información relevante para su trabajo.

Agente de Gestión de Tareas: es el agente encargado de gestionar las tareas y programar reuniones en función de las solicitudes de los empleados y la disponibilidad de los participantes.

Agente de Acceso a Información: es el encargado de proporcionar acceso rápido a documentos, archivos y datos relevantes para el trabajo de los empleados.

Ejemplos de interacción:

1. - Empleado: "Necesito programar una reunión con el equipo de ventas para el próximo martes a las 10:00 a.m."

   - Agente de Gestión de Tareas: "Claro, ¿hay algún otro detalle que deba tener en cuenta para esta reunión, como la duración o la agenda?"

   - Empleado: "Sí, la reunión debería durar aproximadamente una hora y discutiremos las estrategias de ventas para el próximo trimestre."

   - Agente de Gestión de Tareas: "Perfecto, la reunión con el equipo de ventas ha sido programada para el próximo martes a las 10:00 a.m. ¿Hay alguien más que deba ser invitado a esta reunión?"

   - Empleado: "Sí, por favor invita también a nuestro director de marketing."

2. - Empleado: "Asistente, ¿puedes encontrar el informe de ventas del mes pasado y enviármelo por correo electrónico?"

   - Agente de Acceso a Información: "Claro, estoy buscando el informe ahora. ¿A qué dirección de correo electrónico debo enviarlo?"

   - Empleado: "Por favor, envíalo a mi dirección de correo electrónico habitual."

Resultado final

Gracias al asistente virtual, integrado por estos dos agentes, los empleados podrán tener una mejora significativa en la experiencia del usuario y la eficiencia en la gestión de tareas dentro del entorno de trabajo remoto.:

Fuentes de información:

- https://es.wikipedia.org/wiki/Sistema_multiagente
- https://www.b2chat.io/blog/b2chat/sistemas-multiagente-que-son-como-funcionan/
- https://weremote.net/retos-asistentes-virtuales/
- https://fastercapital.com/es/tema/%C2%BFc%C3%B3mo-puede-hacer-que-sus-asistentes-virtuales-sean-exitosos.html