Facultad de
Ciencias Exactas,
Ingeniería y Agrimensura

Trabajo Práctico 2

IA4.2 Procesamiento del Lenguaje Natural

Tecnicatura Universitaria en Inteligencia Artificial

Fontana Gustavo Julián
18/02/2024

Ejercicio 1 - RAG

Creación de un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation) y utilizando fuentes de conocimiento especificadas en las pautas generales del trabajo.
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.

Ejercicio 2 - Agentes

Realizar 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.

Ejercicio 1 - RAG

Introducción

Elijo crear un chatbot experto en la segunda guerra mundial, que pueda responder preguntas referidas a esta temática, evitando otros tópicos de los cuales no tiene conocimiento. El chatbot obtiene la información de una serie de documentos PDFs y una página de wikipedia, como además de un archivo tabular proveniente de la plataforma Kaggle y un grafo creado localmente con entidades y relaciones definidas a modo de ejemplo.

La información se encuentra procesada e indexada, utilizando diferentes librerías, que luego puede ser recuperada para responder a las solicitudes del usuario. La función que llama al modelo de lenguaje, la instrucción que formatea los prompts y la creación de contexto son obtenidas del material de estudio y modificadas a conveniencia.

El conjunto de las funciones definidas permiten la implementación de un programa interactivo que recibe una consulta, la cual es clasificada, permitiendo recuperar información del contexto más conveniente y dar una respuesta útil y precisa.

Explicación del código

Carga de datos y manejo de las fuentes

En primer lugar se define una función `extraer_texto_pdf` para extraer el texto de un PDF. Se utiliza internamente la función extract_text de la librería ***pdfminer***. La función permite extraer el cuerpo del texto y especificar de ser necesario el inicio y fin de página a extraer.
Los parámetros admitidos son:

- ruta_pdf: recibe la ruta del archivo a procesar.
- pagina_inicio: página de inicio de extracción (opcional).
- pagina_final: página final de extracción (opcional).

Luego se comienza a extraer el texto de la página de Wikipedia utilizando ***wikipedia-api***, un contenedor. Se inicializa el contenedor indicando el idioma preferido y el agente. Aplicando el método ***page*** se extrae el cuerpo del texto de la página hasta un determinado número de caracteres, para evitar traer las referencias y enlaces al final de la página. Utilizando expresiones regulares de la librería ***re*** se quitan los enlaces de referencia y se almacena el texto ya procesado y limpio.

Utilizando la librería ‘gdown’ se descargan los archivos PDFs que servirán como fuente de datos y se almacenan en una carpeta en el entorno local.

Se utiliza la función mencionada anteriormente para extraer el texto y almacenarlo. El texto extraído es procesado con el uso de expresiones regulares para quitar los encabezados, pies de páginas y títulos o menciones que indiquen la procedencia del texto, ya que decido concatenar toda la información de los PDFs y la de wikipedia en un solo archivo de texto y proporcionar un contexto unificado al modelo de lenguaje.

Los datos tabulares como mencioné antes, provienen de la plataforma de Kaggle y corresponde a un archivo de texto tabulado en dos columnas, evento y fecha. Y menciona eventos ocurridos durante la Segunda Guerra Mundial y su respectiva fecha.
El archivo original se encontraba en inglés por lo que decidí traducir los nombres de los eventos utilizando ChatGPT.
El archivo en general se encontraba bien formateado, por lo que solo elimine el índice y la existencia de algunas entradas duplicadas.
Para la carga y manipulación del archivo utilice la librería de Pandas.
El archivo fue guardado dentro de una carpeta en el entorno local con la extensión csv. Esto permite cargarlo en el formato adecuado utilizando un cargador de la librería de ***llama-index*** al momento de indexarlo más adelante.

Para la creación del grafo utilice la librería ***networkx***. En primer lugar, creé una instancia de un objeto grafo y luego una lista de entidades, que posteriormente agregué al grafo utilizando el método ***add_nodes_from***.
Luego creé las relaciones entre entidades y las agregue al grafo aplicando el método ***add_edge*** en un ciclo for, y de esta manera hacer coincidir entidades y relaciones.
Al igual que los datos tabulares, guarde el grafo en una carpeta en el entorno local con el mismo objetivo futuro.
Utilizando la librería ***matplolib*** se puede ejecutar una visualización del grafo para observar como se relacionan las entidades.

Luego de tener preparadas todas las fuentes de información, inicio con la preparación de las funciones que conformarán el sistema RAG.

Modelo

Para construir el asistente voy a basarme en la librería ***llama-index*** y en un modelo de ***HuggingFace***. Se va a utilizar el modelo de lenguaje de forma remota, accediendo desde su API. El modelo de generación de texto utilizado es ***zephyr-7b-beta*** y se accede a través de la API de ***HuggingaFace***. Es un modelo simple y pequeño, entrenado para actuar como un asistente útil, que fué entrenado en una mezcla de conjunto de datos públicos y sintéticos.

Las siguientes líneas de código serán las aplicaciones correspondientes a las funciones que permiten la implementación de la técnica RAG y el programa interactivo.

En primer lugar almaceno las claves necesarias para ejecutar el modelo de lenguaje y ciertas herramientas de la librería de ***llama-index*** que utilizan internamente a gpt.

La primera función desplegada `zephyr_instruct_template`, fué extraída del material de estudio y define un formato de plantilla que permite ajustar el contenido del mensaje enviado al modelo según el rol presente. La plantilla admite el rol de sistema, que indica el estilo, tarea que debe cumplir internamente el modelo; el rol usuario, que es quien interactúa realizando solicitudes; y el rol de asistente, es quien siguiendo las reglas del sistema debe responder las solicitudes del usuario.
La plantilla se formatea utilizando la librería ***jinja***, que permite resolver templates.

Los parámetros admitidos son:

- messages: recibe el prompt en formato diccionario con claves rol y contenido.
- add_generation_promt: se encuentra seteado en True, e indica si el prompt generado se debe agregar al final de la plantilla.

Generar

La siguiente función `generate_answer` realiza una solicitud post a la API del modelo enviando los argumentos requeridos.
Los datos requeridos son:

- api_key
- Headers
- Parámetros para el modelo:
  - Temperatura: determina si la generación de texto resulta aleatorio o determinista.
  - max_new_tokens: longitud máxima de las respuestas.
  - top_k: indica al modelo elegir los k token más probables de su lista.
  - top_p: similar a top_k, pero la elección se da por la suma de probabilidades.

Si la solicitud post es exitosa se puede obtener la respuesta al prompt, en caso contrario se devuelve una excepción y se imprime un mensaje de que ha ocurrido un error. La función solo retorna la parte importante de la respuesta.
Las solicitudes a la API se realizan utilizando la librería ***request*** que permite llamadas por servicio web.
Los parámetros admitidos son:

- prompt: recibe el prompt en formato str.
- max_new_tokens: entero seteado que indica el máximo de token por respuesta.

Aumentar

La próxima función implementada `prepare_prompt`, permite aumentar el prompt en base al contexto del mensaje, que será formateado según las instrucciones definidas en la función `zephyr_instruct_template` y será retornado. El contexto se recupera iterando sobre los fragmentos de los documentos (nodos) sobre la base de datos vectorial, cercanos por búsqueda semántica al mensaje del usuario.
El prompt formateado es el que luego recibe la función `generate_answer` y envía al modelo de lenguaje.
Esta función fue tomada del material de estudio y modificada para sólo recibir el texto. Los mensajes al sistema fueron diseñados específicamente para la especialización del chatbot y traducidos al inglés, ya que parecieron ser comprendidos mejor por el LLM y aumentaron su precisión.
Los parámetros admitidos son:

- query_str: recibe la solicitud del usuario en formato str.
- nodes: recibe la lista de nodos más relevantes para responder la solicitud.

Recuperar

Lo siguiente que se realizó fue el almacenamiento de la información extraída de los PDFs en una base de datos vectorial. Para esto se utilizó ***chromadb*** que permite almacenar los embeddings.

Se utilizaron las librerías ***llama-index*** y ***LangChain*** para obtener los modelos de embeddings y split. Los split del texto se realizaron con ***SentenceSplitter***, ya que deseo mantener las oraciones y párrafos juntos, y esta clase me permite evitar pérdida de información. Se setea el tamaño del chunk y el solapamiento en tamaños adecuados para conservar mucho mejor el hilo de la información y evitar perder coherencia.
Como modelo de embeddings se utiliza de la librería ***sentence-transformer*** un modelo que soporta español ***sentence-transformers/paraphrase-multilingual-mpnet-base-v2***.

En primer lugar se realiza el split del texto y luego los embeddings de los fragmentos. A continuación, se crea un objeto para almacenar el cliente de ***chromadb***, seguido se crea una colección, la que posteriormente es añadida al almacenamiento de ***chromadb***.
Finalmente se añaden los ids de los chunks y los embeddings a la colección creada.

Con las fuentes de datos ya procesadas, ahora puedo cargar los datos utilizando la clase ***SimpleDirectoryReader***, un cargador integrado (admite los parámetros de ruta de los archivos, extensión, codificación, etc.) y el método ***load_data***. Este cargador provisto por la librería de ***llama-index***, me permite cargar los datos en el formato Document o Node, que son los formatos requeridos para poder indexarlos.
El formato Document es un contenedor para fuentes de datos, y el Node representa un fragmento del documento.
La carga de los datos tabulares y el grafo, se realiza accediendo y obteniendo los archivos almacenados en las carpetas creadas previamente en el entorno.

Los índices son una estructura de datos que va a estar compuesta por objetos y que me va a permitir realizar consultas por parte de un LLM. Las clases de índices que utilizo son ***VectorStoreIndex*** que toma los documentos y los divide en nodos. Utilizo esta primera clase para indexar el archivo de texto y los datos tabulares. La otra clase es ***KnowledgeGraphIndex*** que extrae los tripletes de los nodos del grafo.
Estas clases admiten una serie de parámetros que me permiten indicar los contenedores de utilidades como ***ServiceContext*** en el cual se puede indicar el modelo de lenguaje a utilizar, los modelos de embeddings, splits y otros valores importantes. También puedo indicar el ***StorageContext***, otro contenedor que me permite almacenar nodos, índices y vectores.
Estos índices me permiten administrar los documentos y procesarlos directamente por default sin la necesidad de hacer splits y embeddings previamente. A pesar de esto, siguiendo la consigna del trabajo, hago uso de un método (***from_vector_store***) que provee el índice, y cargo directamente la base de datos de vectores de ***chromadb*** obtenida previamente al procesar el texto.
Los índices de los datos tabulares los cargo directamente. A través del ***ServiceContext*** indico el modelo de embeddings (el mismo que se utilizó para el texto), y especifico uno para el split del archivo.
En este caso utilizo ***CharacterTextSplitter*** de ***LangChain***, ya que al tratarse de un documento tabular se encuentra ordenado por filas. Entonces, utilizó el caracter de salto de líneas (\n) como separador de chunks. De esta manera, cada chunk es una entrada del archivo csv. El tamaño de los chunks fué elegido en base a la longitud de la entrada más extensa. Además, no se admite el solapamiento de fragmentos, ya que cada fila representa información diferente que no debería mezclarse.

En el caso de los datos del grafo, utilizo la clase de índice por default. Esta clase de cargador tiene los embeddings desactivados.
Por último, el método ***as_retriver*** de los índices, me permite recuperar los nodos o vectores que sean más cercanos semánticamente a la consulta que esté procesando el LLM.

Clasificación

Para poder clasificar las consultas y que el modelo decida desde que indice usar el recuperador, utilizo una técnica de few-shot para proporcionar ejemplos en el prompt y poder condicionar su respuesta.

Para llevar a cabo esta tarea defino la función `clasificar_prompt`, que utiliza un mensaje con formato, en donde se especifican el rol del sistema y el estilo que debe seguir. Después, una serie de ejemplos simulan la entrada de la consulta de un usuario y la clasificación por parte del asistente. En un último mensaje, con el rol de usuario, escribo las pautas para condicionar la clasificación en la respuesta del asistente.
Utilizando la plantilla de zephyr de la función `zephyr_instruct_template` para formatear el mensaje y enviarlo al modelo por medio de `generate_answer`. Esta función envía de vuelta la clasificación, y ahora puede ser retornada.
Los parámetros admitidos son:

- prompt: recibe el prompt en formato str.
- max_new_tokens: entero seteado que indica el máximo de token por respuesta.

Después de especificar diferentes estilos y formatos de ejemplos, logre que el asistente clasificara correctamente y en el formato que yo quería. Nuevamente, las instrucciones las escribí en inglés para mejorar la comprensión.

Ejemplo de clasificación:

```
[ ]  preguntas = ['¿En qué año se llevaron a cabo los juicios contra Japón?',
                  '¿Cuál fué el motivo de la Segunda Guerra Mundial?',
                  '¿Quiénes integraban las fuerzas aliadas?',
                  '¿Cuándo se firmó el tratado de Versalles?',
                  'Dar una breve descripción de la Segunda Guerra Mundial',
                  '¿Quién lideró Alemania durante la Segunda Guerra Mundial?'
                  ]
     for pregunta in preguntas:
       print(f'{pregunta} --> {clasificar_prompt(pregunta)}\n')
```

```
¿En qué año se llevaron a cabo los juicios contra Japón? --> Categoría: Tabular

¿Cuál fué el motivo de la Segunda Guerra Mundial? --> Categoría: Texto

¿Quiénes integraban las fuerzas aliadas? --> Categoría: Grafo

¿Cuándo se firmó el tratado de Versalles? --> Categoría: Tabular

Dar una breve descripción de la Segunda Guerra Mundial --> Categoría: Texto

¿Quién lideró Alemania durante la Segunda Guerra Mundial? --> Categoría: Grafo
```

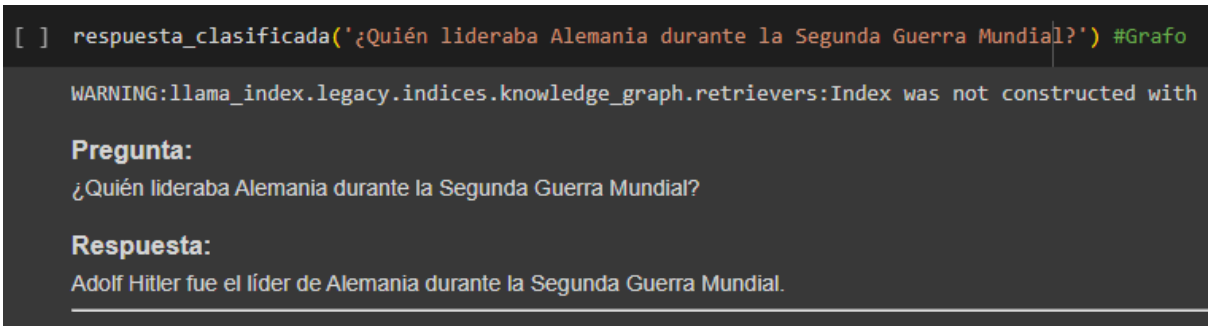
Llegando al final, defino una función para obtener la respuesta de la consulta en base al recuperador adecuado, la función respuesta_general.
Los parámetros admitidos son:

- prompt: recibe el prompt en formato str.
- retriever: indica el recuperador adecuado al cual realizar la consulta.
- max_new_tokens: entero seteado que indica el máximo de token por respuesta.

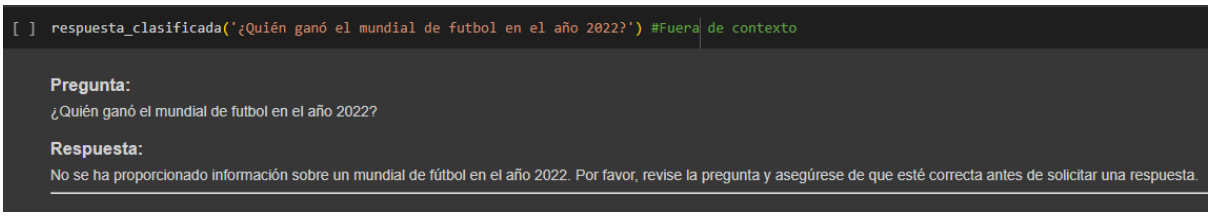
Y defino también, la función respuesta_clasifiada que recibe un prompt y se lo envía a la función clasificar_prompt obteniéndose la etiqueta de clasificación. Siguiendo unos condicionales se envía el prompt a su correspondiente recuperador para obtener la respuesta final.
Los parámetros admitidos son:

- prompt: recibe el prompt en formato str.
- max_new_tokens: entero seteado que indica el máximo de token por respuesta.

Ejemplo de clasificación de prompts sobre los recuperadores:



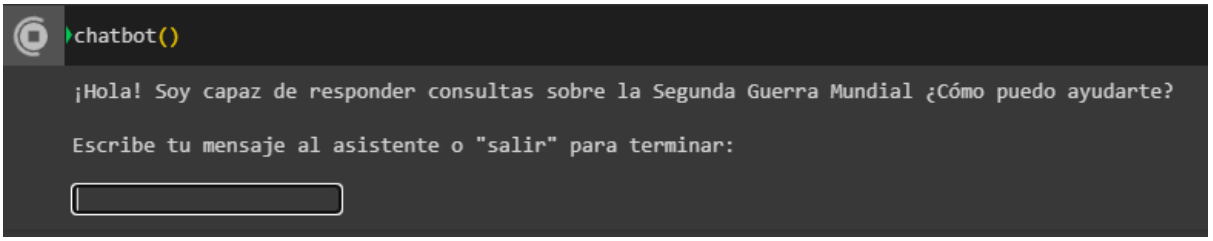
Nota: al no utilizar embeddings en la creación del índice de grafo se recibe una advertencia que no logre desactivar.

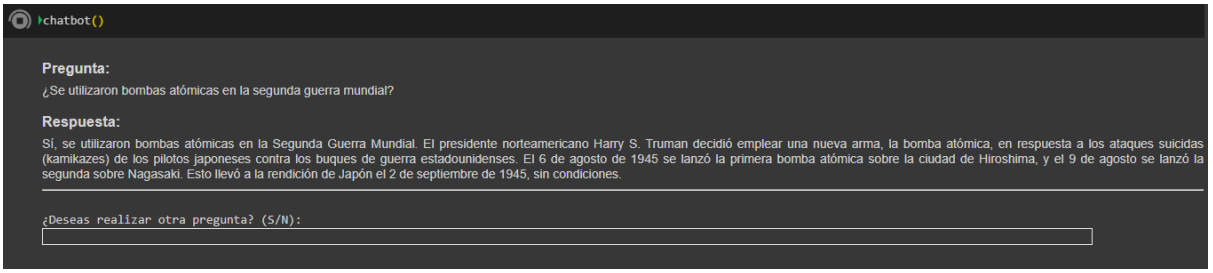


Chatbot

Para simular un chatbot, creo un programa interactivo que genera un bucle que se ejecute mientras el usuario tenga consultas para realizar. El programa recibe la consulta y se la envía a la función respuesta_clasifiada para obtener la respuesta. El usuario puede elegir entre las opciones disponibles para seguir consultando o salir del programa. La opción salir termina el ciclo y finaliza la llamada al modelo de lenguaje.
Para el chatbot, se utiliza la librería IPython, específicamente módulos para limpiar la pantalla y mejorar la visualización de la escritura impresa.

Uso del chatbot:





Ejercicio 2 - Agentes

Agentes Inteligentes:

Lo primero sería intentar definir que es un agente inteligente, y para ello podríamos utilizar una definición simple y decir que son programas informáticos que están diseñados específicamente para interactuar con su entorno, procesar información y tomar decisiones de forma autónoma para llevar a cabo una determinada tarea.
La existencia de los agentes inteligentes está ligada a un propósito capaz de satisfacer a un usuario final. Este propósito debe poder ser alcanzable por el propio agente, lo que requiere capacidad de construir su propio conocimiento a partir de fuentes existentes y la capacidad de razonar sobre él para evolucionarlo. Recopilar feedback es algo importante que le permite mejorar la inteligencia, obtener experiencia, poder lidiar con errores y tomar decisiones más óptimas con el tiempo.

La evolución de estos tipos de agentes ha traído múltiples beneficios, por ejemplo en la automatización de tareas, que antes debían ser realizadas por personas y que llegaban a resultar tediosas y repetitivas.
Un ecosistema de agentes se puede aplicar a cualquier proyecto en el que haya una necesidad de automatizar tareas complejas que solo pueden realizar personas expertas. Resultando así, aplicaciones médicas, chatbots de asistencia, robótica, etc. Cada agente con una función específica capaz de utilizar herramientas y comandos para realizar acciones dentro de su dominio.

Con los avances en la inteligencia artificial de los últimos tiempos, los lenguajes de programación evolucionaron posibilitando la integración de grandes modelos de lenguaje, que permitieron la creación de agentes inteligentes capaces de interactuar con el mundo real. Esto permitió acompañar el incesante crecimiento de la demanda de servicios cada vez más complejos y difíciles de ejecutar por parte de los usuarios.

La creación de aplicaciones con esta nueva dinámica posibilitó ofrecer una experiencia más natural en la integración de estos usuarios, que ahora pueden expresarse a través del lenguaje natural y resolver tareas de una manera más eficiente.

Si lo vemos desde un punto de vista empresarial, importantes empresas en los últimos años han comenzado a implementar pruebas de estas tecnologías convirtiéndolas en un diferenciador competitivo, transformando los modelos de negocios convencionales, y proveyendo interacciones más eficientes y naturales con sus clientes y empleados.

Nota: la integración de múltiples agentes inteligentes trabajando colectivamente para lograr un objetivo común conforma un sistema multiagente, donde cada agente puede especializarse en una tarea particular.

Ejemplo del uso de agentes inteligentes

- Asistentes personales: funcionalidades de agenda o asesoramiento financiero. Por ejemplo, actuando como un secretario personal.
- Chatbots: conversación informal o asistencia.
- Asistentes de voz: utilizan lenguaje natural y son capaces de ejecutar órdenes, por ejemplo Google Home.
- Indicaciones y geolocalización: información en tiempo real de rutas y caminos.
- Transcripción o traducción de texto a partir de un video.
- Asistentes médicos: colaboración en sugerencias de diagnósticos médicos.
- Robótica: control y navegación de robots.

Aplicaciones de agentes inteligentes usando modelos LLM libres

Actualmente se utilizan diversas aplicaciones para la creación de agentes inteligentes que integran modelos libres de lenguaje, algunos muy populares pueden ser LangChain y AutoGen.

LangChain es una herramienta popular que permite trabajar con agentes de forma sencilla, facilitando la realización de consultas y análisis de datos, lo que le posibilita la interacción con objetos del mundo real de manera automatizada y eficiente.
Un agente de LangChain se considera una entidad virtual capaz de utilizar modelos de lenguaje generativo para interactuar con bases de datos, archivos tabulares, bases de datos SQL, etc.

El funcionamiento se basa en iteración y retroalimentación continua. Los agentes a través de los modelos de lenguaje como GPT-3 adquieren la capacidad de comprender y manejar el lenguaje natural.

Las utilidades que provee LangChain permiten crear agentes inteligentes con los parámetros requeridos para su funcionamiento. Con el agente activo, este puede recibir comandos y consultas en lenguaje natural, los cuales podrá procesar mediante el modelo de lenguaje y responder con las posibles acciones a tomar para resolver la tarea.

LangChain ofrece diversos agentes que permiten interactuar con el mundo real, con su función específica y herramientas particulares para desempeñar una tarea. Algunos ejemplos de sus agentes son:

- Agente de marcos para archivos pandas, para analizar y manipular dataframes utilizando el paquete de Pandas en Python.
- Agente de JSON, que permite interactuar con datos JSON de manera fácil.
- Agente de Vector Store, que proporciona herramientas para realizar búsquedas y consultas en vectores almacenados.

AutoGen es un famework que permite abstraer e implementar agentes conversacionales que integran modelos de lenguaje, herramientas y humanos a través de chat automatizado. La automatización permite que cada agente pueda realizar una tarea particular dentro de la problemática global. Cada agente puede comunicarse con otro agente y realizar acciones en base al conocimiento recibido o diferir completamente de él.

Los agentes de AutoGen tienen características particulares:

- Conversables: significa que cada agente puede enviar o recibir mensajes de otros agentes para iniciar o continuar una conversación.
- Personalizable: los agentes se pueden personalizar para integrar LLM, humanos, herramientas o una combinación de ellos.

AutoGen provee de la clase ConversableAgents para que los agentes sean capaces de conversar entre sí a través del intercambio de mensajes. Además, ofrece una subclase AssistantAgent diseñado para actuar como un asistente de IA sin requerir intervención humana y ejecución de código, y UserProxyAgent que solicita entrada humana como respuesta en cada turno de interacción de forma predeterminada, y que también tiene la capacidad de ejecutar código y realizar llamada a funciones o herramientas.

Algunos ejemplos de chat automatizado de multiagentes:

- Generación, depuración y ejecución de código, puede escribir código Python para que un usuario lo ejecute en una tarea determinada.
- Juego de Ajedrez automatizado.
- Navegación Web, permite navegar en páginas, descargar archivos, recuperar el historial, etc.
- Traducción a lenguaje natural de instrucciones a un SQL.

Problemática

Como elección de problemática elijo simular un sistema multiagentes que participe en un juego de video. Particularmente, en las batallas por turnos de Pokémon, y sea capaz de analizar la situación y proceder de manera inteligente, generando alguna estrategia que le permita resultar vencedor.

Descripción del juego

La serie de videojuegos de Pokémon, son videojuegos de rol donde el jugador controla al protagonista desde una perspectiva aérea y recorre una región ficticia para dominar las batallas Pokémon. El objetivo principal del juego es convertirse en el campeón de la región al derrotar a los mejores entrenadores Pokémon.

Cuando el jugador se encuentra con un Pokémon salvaje o es desafiado por un entrenador, la pantalla cambia a una pantalla de batalla basada en turnos que muestra el Pokémon del jugador y el Pokémon retador.

Descripción general de cómo funciona el sistema de batallas siguiendo reglas básicas:

1. Turnos: La batalla se desarrolla por turnos, donde cada entrenador y sus Pokémon tienen la oportunidad de realizar una acción en su turno.
2. Selección de Movimientos: En cada turno, el jugador elige un movimiento para su Pokémon. Los movimientos pueden ser ataques ofensivos, movimientos de soporte, cambios de Pokémon, uso de objetos, entre otros.
3. Tipos de Pokémon y Movimientos: Cada Pokémon y cada movimiento tienen un tipo asociado (por ejemplo, Fuego, Agua, Planta, Eléctrico, etc.). Los tipos interactúan entre sí siguiendo un sistema de fortalezas y debilidades. Por ejemplo, un Pokémon de tipo Agua será débil contra movimientos de tipo Eléctrico, pero fuerte contra movimientos de tipo Fuego.
4. Estados: Los Pokémon pueden ser afectados por diversos estados durante la batalla, como quemaduras, parálisis, sueño, confusión, etc. Estos estados pueden influir en su capacidad para atacar o defenderse.
5. Estadísticas: Cada Pokémon tiene estadísticas que determinan su fuerza, velocidad, defensa, etc. Estas estadísticas influyen en el daño que causan los movimientos y en su capacidad para resistir los ataques.
6. Intercambio de Pokémon: Durante la batalla, los entrenadores pueden optar por cambiar de Pokémon para adaptarse mejor a la situación. Esto permite aprovechar las fortalezas y debilidades de los diferentes Pokémon en el equipo.
7. Victoria y Derrota: La batalla continúa hasta que uno de los entrenadores derrota a todos los Pokémon del oponente. El entrenador que resulta vencedor, recibe recompensas como monedas y experiencia para sus Pokémon.

Agentes involucrados:

1 - Agente de Toma de Decisiones:
Este agente está encargado de seleccionar a los Pokemones y movimientos en cada turno de la batalla.
La selección se realiza en base a la retroalimentación recibida de los otros agentes y el acceso a una API pokémon.

2 - Agente de Evaluación del Equipo Oponente:
Este agente evalúa el equipo oponente durante la batalla para identificar posibles amenazas y debilidades.

Analiza los tipos de Pokémon del oponente, sus movimientos conocidos y las estrategias empleadas para ayudar en la toma de decisiones durante la batalla.
Utiliza un base de datos para acceder a la información requerida para su análisis.

3 - Agente de Coordinación de Estrategias:
Este agente coordina las estrategias entre los Pokémon del mismo equipo durante la batalla, y maneja información relevante como nivel, movimientos disponibles, salud, etc .
Identifica oportunidades para la combinación de movimientos y tácticas entre los Pokémon del equipo para maximizar la efectividad y minimizar los riesgos.
Obtiene información de una posible base de datos.

4 - Agente de Adaptación y Aprendizaje:
Este agente está diseñado para adaptar las estrategias y tácticas durante la batalla en función de los movimientos del oponente y los cambios en la situación como los cambios en el campo de batalla y la efectividad de los ataques.
Almacena el historial en una base de datos relacional a la que puede acceder para mejorar su desempeño a lo largo del tiempo y ajustar su comportamiento en función de las experiencias pasadas

Simulación de batalla e interacción de agentes

Equipo del Sistema Multiagente:
-Pokémon 1: Un Pokémon de tipo Agua. Es eficaz para contrarrestar los Pokémon de tipo Fuego y Roca.
Pokémon 2: Un Pokémon de tipo Planta/Veneno. Es útil contra Pokémon de tipo Agua y Tierra.

Equipo del adversario:
-Pokémon 1: Un Pokémon de tipo Eléctrico. Puede ser utilizado para dañar a los Pokémon de tipo Agua y Volador.
Pokémon 2: Un Pokémon de tipo Tierra/Roca. Es resistente y puede infligir daño fuerte a los Pokémon de tipo Eléctrico y Fuego..

Ejemplo de conversación entre los Agentes:

1 - [Agente de Toma de Decisiones] mensaje al agente 2 = “El equipo adversario desplegó un pokémon tipo agua, ¿cómo debería responder?”

2 - [Agente de Evaluación del Equipo Oponente] Ejecutando: Obteniendo los movimientos del pokemon adversario de la base de datos, y analizando las debilidades a diferentes tipos de movimiento ; mensaje al agente 3 = “Para contrarrestar el daño es conveniente un pokémon tipo eléctrico y movimientos del mismo tipo.”

3 - [Agente de Coordinación de Estrategias] Ejecutando: consultando el equipo disponible ; mensaje al agente 1 = “Deberías responder desplegando el pokémon 1 y utilizando movimientos tipo eléctrico.”

1 - [Agente de Toma de Decisiones] Ejecutando: se selecciona al pokémon elegido y se lo despliega.

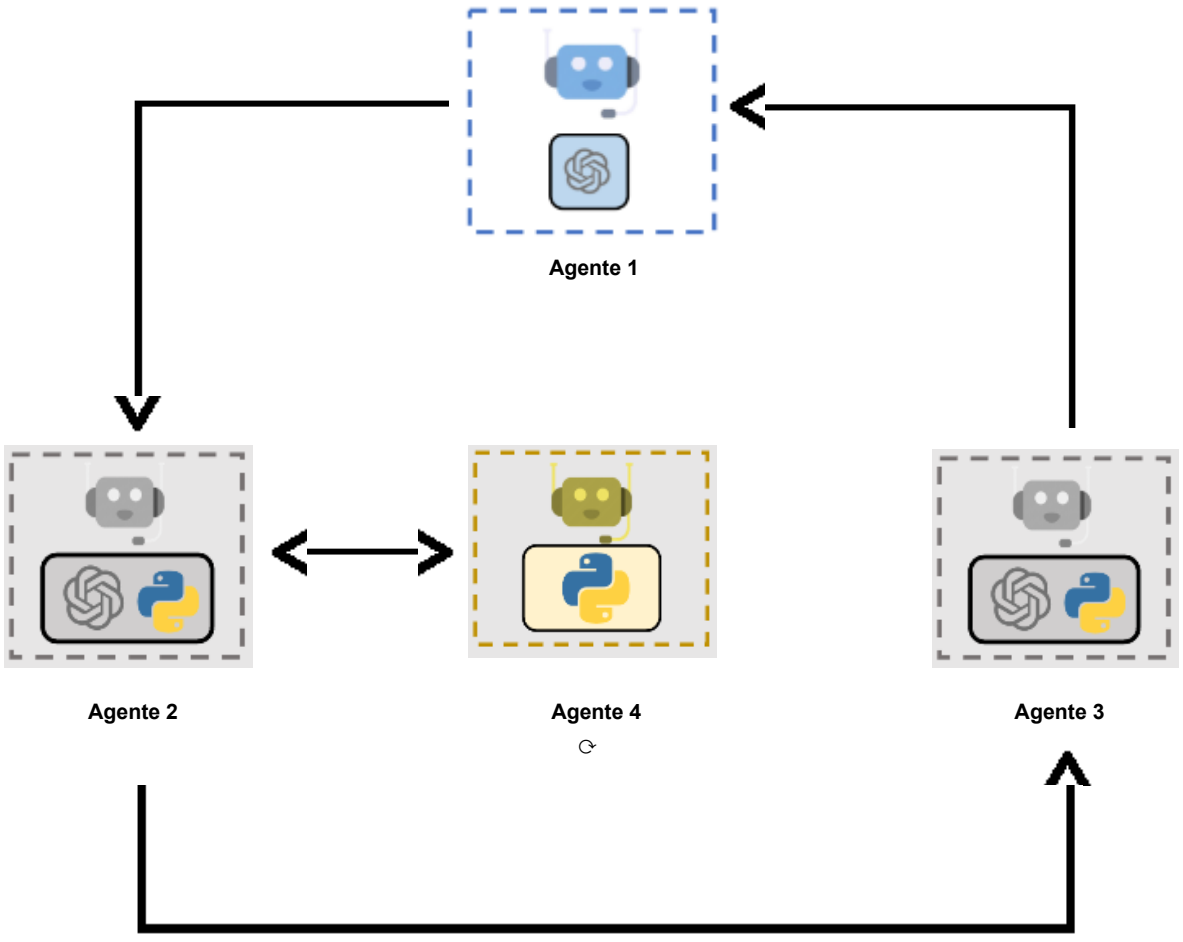
4 - [Agente de Adaptación y Aprendizaje] Registra los resultados de la batalla y puede retroalimentar en contextos similares al agente 2.

Conclusión del sistema multiagentes en la problemática elegida

Teniendo en cuenta la información obtenida de la API y la base de datos, así como la experiencia acumulada, turno a turno se selecciona el Pokémon del equipo del sistema multiagente y el movimiento adecuado como respuesta al ataque de un adversario.

Este enfoque permite que los agentes tomen decisiones más informadas y adapten sus estrategias en función de la situación actual y la experiencia acumulada, lo que debería mejorar su desempeño en futuras batallas.

Esquema del sistema multiagentes



- ➔ **Agente 1:** Agente de toma de decisiones;
- ➔ **Agente 2:** Agente de evaluación el equipo oponente;
- ➔ **Agente 3:** Agente de coordinación de estrategias;
- ➔ **Agente 4:** Agente de adaptación y aprendizaje.

Fuentes:

ICCSI. (s.f.). Agentes de software inteligentes: historia y evolución. Recuperado de [https://iccsi.com.ar/agentes-de-software-inteligentes-historia/]

Kejriwal, K. (22 de noviembre de 2023). OpenAgents: una plataforma abierta para agentes lingüísticos en la naturaleza. Recuperado de [https://www.unite.ai/es/openagents-una-plataforma-abierta-para-agentes-ling%C3%BC%C3%ADsticos-en-la-naturaleza/]

Botpress Blog. (25 de noviembre de 2023). ¿Cuáles son los distintos tipos de agentes de IA? Recuperado de [https://botpress.com/es/blog/what-are-the-different-types-of-ai-agents]

Marín, R. (14 de diciembre de 2023.). Agentes inteligentes que aprenden, deciden y actúan. Recuperado de [https://www.inesem.es/revistadigital/informatica-y-tics/agentes-inteligentes/]

Kamalraj, M. (07 de febrero de 2024). Agentes inteligentes en LangChain: IA y automatización. Recuperado de [https://www.toolify.ai/es/ai-news-es/agentes-inteligentes-en-langchain-ia-y-automatizacin-991751]

Microsoft Learn. (11 de julio de 2023). ¿Qué son los agentes de inteligencia artificial? Recuperado de [https://learn.microsoft.com/es-es/azure/cloud-adoption-framework/innovate/best-practices/conversational-ai]

Ingeniería y tecnología. (02 de febrero de 2023). Los agentes inteligentes: funciones y ejemplos de uso. Recuperado de [https://www.unir.net/ingenieria/revista/agentes-inteligentes/]

Microsoft. (s.f.). Ejemplos. Recuperado de [https://microsoft.github.io/autogen/docs/Examples/#automated-multi-agent-chat]

Ábalos, N. (27 de noviembre de 2014). Ecosistema de agentes inteligentes: aproximación a la Inteligencia Artificial. Recuperado de [https://medium.com/@nieves_as/ecosistema-de-agentes-inteligentes-aproximaci%C3%B3n-a-la-inteligencia-artificial-babea4dfcbfa]

Microsoft. (s.f.). Multi-agent Conversation Framework. Recuperado de [https://microsoft.github.io/autogen/docs/Use-Cases/agent_chat/]