

USO DE GRANDES MODELOS DE LENGUAJE (LLM) EN LA INVESTIGACIÓN: ¿CÓMO CREAR UN ASISTENTE LOCALMENTE?

DAVID*, JAFFET, LUIS

Dirección del proyecto

* corresponding author: `cmd@cmd.com`

ABSTRACT. En los últimos años se ha experimentado un crecimiento en la elaboración y consumo de herramientas que hacen uso del análisis de dato o "big data", entre ellos los chatbots con I.A. Sin embargo, la gran popularidad de estos productos deja de lado la transmisión de la información acerca de lo que ocurre detrás de la capa final presentada a los usuarios, el conocimiento más profundo de como funcionan los modelos se puede aplicar a la investigación para facilitar tareas como búsqueda y síntesis de información. En este sentido, el documento pretende acercar al lector a las tecnologías que se tienen ahora mismo en producción para los gigantes tecnológicos como google, openai, amazon entre otros, así como presentar alternativas de uso libre, que un academico o universitario pueda seguir desarrollando de forma independiente para fines educativos o de aplicación en el sector privado.

KEYWORDS: I.A. Agentes, Procesamiento de lenguaje, Modelos de código abierto, Agentes especializados.

1. INTRODUCCIÓN

En el año 2017, se publicó el artículo "Attention is All You Need", un trabajo fundamental que sentó las bases para el desarrollo de los modelos de lenguaje a gran escala. Este artículo introdujo el mecanismo de atención como componente central en el modelo Transformer, revolucionando el campo del procesamiento del lenguaje natural (NLP). Gracias a esta innovación, en pocos años surgieron productos como ChatGPT, Bard y Gemini, que destacaron como algunos de los lanzamientos tecnológicos más importantes del año 2023.

Sin embargo, estos avances no surgieron de la nada. Desde el siglo pasado, grandes científicos como Alan Turing sentaron las bases teóricas y prácticas de la inteligencia artificial (IA) y la automatización de tareas. Turing, con su célebre prueba que lleva su nombre y su trabajo sobre máquinas computacionales, abrió el camino hacia sistemas capaces de emular procesos de razonamiento humano. Los avances en IA son el resultado de décadas de investigación interdisciplinaria, integrando áreas como la matemática, la estadística, la computación y la lingüística.

En los últimos meses, se ha observado un creciente interés por las capacidades de la inteligencia artificial generativa, que promete transformar múltiples industrias. Según Amazon Web Services (AWS), un agente de inteligencia artificial (IA) es "un programa que puede interactuar con su entorno, recopilar datos y utilizarlos para realizar tareas definidas de forma autónoma a fin de cumplir unos objetivos predeterminados." Tradicionalmente, los programas o bots diseñados para realizar tareas automatizadas requieren la definición previa de las posibles situaciones que

podrían surgir en producción para tomar decisiones adecuadas. Incluso cuando ocurren acciones imprevisibles, estos sistemas deben implementar mecanismos de control, como estructuras de manejo de errores Try-Catch, para minimizar los riesgos y mantener el funcionamiento dentro de los parámetros definidos.

Además, en foros especializados como un subreddit de la comunidad de ChatGPT, se ha discutido ampliamente sobre las mejores prácticas para implementar modelos generativos. Al igual que otros servicios bajo demanda, se recomienda frecuentemente utilizar el modelo de pago "pay as you go" para maximizar la eficiencia económica. Sin embargo, para casos específicos, también es aconsejable optar por modelos preentrenados que puedan ser especializados para realizar tareas definidas con mayor precisión.

Por último, el éxito en la implementación de agentes locales también depende de la reproducibilidad de los entornos en los que operan. Esto implica establecer parámetros claros para garantizar consistencia y permitir el ajuste fino de los modelos en aplicaciones específicas, asegurando que el agente pueda adaptarse eficientemente a las particularidades de su contexto operativo.

2. METODOLOGÍA

En lo siguiente mostramos las condiciones para la reproducibilidad en la elaboración de agentes especializados. El código necesario se adjunta al presente documento y se puede consultar con los autores para una versión actualizada.

| Dependencia | Funcionalidad | Versión sugerida |
|-------------|---|------------------------------------|
| torch | Proporciona operaciones tensoriales y funcionalidades para machine learning, incluyendo cálculo de similitudes coseno y manejo de tensores. En este código, se usa para manejar embeddings y cálculos de similitud. | 2.0.1 o superior |
| ollama | Se utiliza para interactuar con el modelo de lenguaje Ollama, generando embeddings y completando consultas. Específico para la API Ollama. | Consulte documentación |
| os | Manejo del sistema operativo, como comprobar la existencia de archivos y manejar rutas. | Incluido en la biblioteca estándar |
| openai | Interfaz para interactuar con los modelos de lenguaje OpenAI. Aquí, se configura para conectarse a una API local de Ollama. | 0.27.2 |
| argparse | Permite manejar argumentos desde la línea de comandos, como especificar el modelo que se utilizará. | Incluido en la biblioteca estándar |
| json | Maneja la serialización y deserialización de datos JSON, útil para la comunicación entre sistemas. | Incluido en la biblioteca estándar |

TABLE 1. Dependencias utilizadas en el código y su funcionalidad

| Requirement | Specification |
|-------------------------------------|--|
| Operating System | Linux: Ubuntu 22.04 or later |
| RAM | 16 GB for running models up to 7B |
| Disk Space | 12 GB for installing Ollama and basic models. Additional space is required for storing model data depending on the used models. |
| Processor | Recommended to use a modern CPU with at least 4 cores. For running models up to 13B, a CPU with at least 8 cores is recommended. |
| Graphics Processing Unit (optional) | A GPU is not required for running Ollama, but can improve performance, especially when working with large models. If you have a GPU, you can use it to accelerate training of custom models. |

TABLE 2. System Requirements for Ollama

2.1. REQUISITOS Y CARACTERÍSTICAS
COMPUTACIONALES

Para poder ejecutar de forma exitosa los agentes especializados; se debe contar con Ollama de Meta instalado, el cuál tiene como requisitos mínimos los que se listan en la Tabla 2.

Una vez que nos hemos asegurado de tener instalado Ollama podemos crear un entorno virtual con venv en python, el cuál se crea escribiendo en un archivo de texto plano las dependencias : openai, torch, PyPDF2, ollama, pyyaml, BeautifulSoup4, lxml, python-dotenv. El archivo debe separar las paqueterías por cambios de línea y debe nombrarse "requirements.txt" en el directorio donde trabajaremos se ejecuta

```
$ python -m venv asistentesai
$ .\asistentesai\Scripts\activate
$ pip install -r requirements.txt
```

2.2. PRUEBA DE ACCESO A LOS MODELOS

Para probar si tenemos acceso a los modelos se puede hacer una solicitud de descarga de modelo:

```
$ ollama pull llama3
```

Lo anterior nos dara posibilidad de ejecutar el modelo de forma local:

```
$ ollama run llama3
```

3. RESULTADOS

Como resultado podemos obtener las respuestas de más de un agente-modelo de lenguaje utilizando la herramienta Ollama, como valor agregado a nuestro producto podemos ejecutar consultas al modelo con un contexto que se proporciona a través del script de python "upload.py" presentado en el apendice de este documento. Para establecer una pregunta inicial que permita ilustrar la diferencia en respuestas elegimos "Hi, please explain briefly: Why are different constellations observed in the sky depending on the time of the year?"

4. CONCLUSIONES

Se implementó el modelo de lenguaje Ollama en sus versiones 3 y 3.2 de forma local, se presentan los requerimientos necesarios para que se replique este entorno de trabajo facilitando la programación de agentes que busquen, guarden o escriban información en temas especializados a través de un archivo de texto plano que se proporciona como contexto.

Conversation 1: Ollama 3

User: Hi, please explain briefly: Why are different constellations observed in the sky depending on the time of the year?

Ollama3: Different constellations are visible in the sky at different times of the year because of the way the Earth and the stars move. The Earth rotates on its axis (from west to east) and orbits around the Sun (from December to June). This means that different parts of the celestial sphere, which is the imaginary sphere that surrounds the Earth, are visible from our planet at different times. As the Earth moves in its orbit, it also moves in relation to the stars. The stars appear to shift slightly over time due to this movement. This phenomenon is called "proper motion." Additionally, some stars are closer or farther away from us than others. As a result, those that are closer will appear to change position relative to more distant ones. So, when we look at the night sky on different dates, we're seeing the same celestial sphere but with different parts of it visible due to the Earth's motion and the stars' proper motion. In simple terms, the changing view of constellations in the sky is like looking through a window that moves slightly as you walk away.

Conversation 2: Ollama 3.2

User: Hello

Ollama3.2: The constellations we see in the sky change throughout the year due to the Earth's movement. Here's why:

1. **Earth's orbit around the Sun**: The Earth takes about 365 days to complete one orbit around the Sun.
2. **Axis tilt**: The Earth's axis is tilted at an angle of about 23.5 degrees relative to its orbital plane around the Sun.
3. **Seasonal changes**: As the Earth orbits the Sun, different parts of the planet are exposed or hidden from view, depending on their position in relation to the Sun and the Earth's axis.

As a result, when we're closer to the North Pole (winter), the stars in the Southern Hemisphere constellations are more visible. Conversely, when we're farther from the North Pole (summer), the stars in the Northern Hemisphere constellations are more prominent. This is why different constellations appear in the sky at various times of the year:

- * Winter: Orion, Taurus, and Gemini are prominent.
- * Spring: Lyra, Cygnus, and Draco become visible.
- * Summer: Scorpius, Sagittarius, and Andromeda are more prominent.
- * Autumn: Cassiopeia, Ursa Major, and Pegasus appear in the sky.

The changing constellations are a result of our planet's unique position in its orbit around the Sun and the tilt of its axis.

A. APPENDICES

Se presentan los dos archivos en python necesarios para ejecutar agentes especializados en un cuerpo de texto:

A.1. UPLOAD.PY

Carga el documento de texto como tokens para ser procesados por el modelo:

```

import os
import tkinter as tk
from tkinter import filedialog
import PyPDF2
import re
import json

# Function to convert PDF to text and append to vault.txt
def convert_pdf_to_text():
    file_path = filedialog.askopenfilename(filetypes=[("PDF Files", "*.pdf")])
    if file_path:
        with open(file_path, 'rb') as pdf_file:
            pdf_reader = PyPDF2.PdfReader(pdf_file)
            num_pages = len(pdf_reader.pages)
            text = ''
            for page_num in range(num_pages):
                page = pdf_reader.pages[page_num]
                if page.extract_text():
                    text += page.extract_text() + " "

            # Normalize whitespace and clean up text
            text = re.sub(r'\s+', ' ', text).strip()

            # Split text into chunks by sentences, respecting a maximum chunk size
            sentences = re.split(r'(?<[!?!]) +', text) # split on spaces
            chunks = []
            current_chunk = ""
            for sentence in sentences:
                # Check if the current sentence plus the current chunk exceeds the limit
                if len(current_chunk) + len(sentence) + 1 < 1000: # +1 for the space
                    current_chunk += (sentence + " ").strip()
                else:
                    # When the chunk exceeds 1000 characters, store it and start a new one
                    chunks.append(current_chunk)
                    current_chunk = sentence + " "
            if current_chunk: # Don't forget the last chunk!
                chunks.append(current_chunk)
            with open("vault.txt", "a", encoding="utf-8") as vault_file:
                for chunk in chunks:
                    # Write each chunk to its own line
                    vault_file.write(chunk.strip() + "\n") # Two newlines to separate chunks
            print(f"PDF content appended to vault.txt with each chunk on a separate line.")

# Function to upload a text file and append to vault.txt
def upload_txtfile():
    file_path = filedialog.askopenfilename(filetypes=[("Text Files", "*.txt")])
    if file_path:
        with open(file_path, 'r', encoding="utf-8") as txt_file:
            text = txt_file.read()

            # Normalize whitespace and clean up text
            text = re.sub(r'\s+', ' ', text).strip()

            # Split text into chunks by sentences, respecting a maximum chunk size
            sentences = re.split(r'(?<[!?!]) +', text) # split on spaces
            chunks = []
            current_chunk = ""

```

```

1         for sentence in sentences:
2             # Check if the current sentence plus the current chunk exceeds the limit
3             if len(current_chunk) + len(sentence) + 1 < 1000: # +1 for the space
4                 current_chunk += (sentence + " ").strip()
5             else:
6                 # When the chunk exceeds 1000 characters, store it and start a new one
7                 chunks.append(current_chunk)
8                 current_chunk = sentence + " "
9             if current_chunk: # Don't forget the last chunk!
10                chunks.append(current_chunk)
11            with open("vault.txt", "a", encoding="utf-8") as vault_file:
12                for chunk in chunks:
13                    # Write each chunk to its own line
14                    vault_file.write(chunk.strip() + "\n") # Two newlines to separate chunks
15            print(f"Text file content appended to vault.txt with each chunk on a separate line.")
16
17        # Function to upload a JSON file and append to vault.txt
18        def upload_jsonfile():
19            file_path = filedialog.askopenfilename(filetypes=[("JSON Files", "*.json")])
20            if file_path:
21                with open(file_path, 'r', encoding="utf-8") as json_file:
22                    data = json.load(json_file)
23
24                # Flatten the JSON data into a single string
25                text = json.dumps(data, ensure_ascii=False)
26
27                # Normalize whitespace and clean up text
28                text = re.sub(r'\s+', ' ', text).strip()
29
30                # Split text into chunks by sentences, respecting a maximum chunk size
31                sentences = re.split(r'(?<=[.!?]) +', text) # split on spaces
32                chunks = []
33                current_chunk = ""
34                for sentence in sentences:
35                    # Check if the current sentence plus the current chunk exceeds the limit
36                    if len(current_chunk) + len(sentence) + 1 < 1000: # +1 for the space
37                        current_chunk += (sentence + " ").strip()
38                    else:
39                        # When the chunk exceeds 1000 characters, store it and start a new one
40                        chunks.append(current_chunk)
41                        current_chunk = sentence + " "
42                    if current_chunk: # Don't forget the last chunk!
43                        chunks.append(current_chunk)
44                with open("vault.txt", "a", encoding="utf-8") as vault_file:
45                    for chunk in chunks:
46                        # Write each chunk to its own line
47                        vault_file.write(chunk.strip() + "\n") # Two newlines to separate chunks
48                print(f"JSON file content appended to vault.txt with each chunk on a separate line.")
49
50        # Create the main window
51        root = tk.Tk()
52        root.title("Upload .pdf, .txt, or .json")
53
54        # Create a button to open the file dialog for PDF
55        pdf_button = tk.Button(root, text="Upload PDF", command=convert_pdf_to_text)
56        pdf_button.pack(pady=10)
57
58        # Create a button to open the file dialog for text file
59        txt_button = tk.Button(root, text="Upload Text File", command=upload_txtfile)
60        txt_button.pack(pady=10)
61
62        # Create a button to open the file dialog for JSON file
63        json_button = tk.Button(root, text="Upload JSON File", command=upload_jsonfile)
64        json_button.pack(pady=10)

```

```

1  # Run the main event loop
2  root.mainloop()

```

A.2. RESPONSE.PY

Responde a una pregunta al ejecutarse, se basa en el contexto que se le proporciona en upload.py:

```

10 import torch
11 import ollama
12 import os
13 from openai import OpenAI
14 import argparse
15 import json
16
17 # ANSI escape codes for colors
18 PINK = '\033[95m'
19 CYAN = '\033[96m'
20 YELLOW = '\033[93m'
21 NEON_GREEN = '\033[92m'
22 RESET_COLOR = '\033[0m'
23
24 # Function to open a file and return its contents as a string
25 def open_file(filepath):
26     with open(filepath, 'r', encoding='utf-8') as infile:
27         return infile.read()
28
29 # Function to get relevant context from the vault based on user input
30 def get_relevant_context(rewritten_input, vault_embeddings, vault_content, top_k=3):
31     if vault_embeddings.nelement() == 0: # Check if the tensor has any elements
32         return []
33     # Encode the rewritten input
34     input_embedding = ollama.embeddings(model='mxbai-embed-large', prompt=rewritten_input)["embedding"]
35     # Compute cosine similarity between the input and vault embeddings
36     cos_scores = torch.cosine_similarity(torch.tensor(input_embedding).unsqueeze(0), vault_embeddings)
37     # Adjust top_k if it's greater than the number of available scores
38     top_k = min(top_k, len(cos_scores))
39     # Sort the scores and get the top-k indices
40     top_indices = torch.topk(cos_scores, k=top_k)[1].tolist()
41     # Get the corresponding context from the vault
42     relevant_context = [vault_content[idx].strip() for idx in top_indices]
43     return relevant_context
44
45 def rewrite_query(user_input_json, conversation_history, ollama_model):
46     user_input = json.loads(user_input_json)["Query"]
47     context = "\n".join([f"{msg['role']}: {msg['content']}" for msg in conversation_history[-2:]])
48     prompt = f"""Rewrite the following query by incorporating relevant context from the conversation history.
49     The rewritten query should:
50
51     - Preserve the core intent and meaning of the original query
52     - Expand and clarify the query to make it more specific and informative for retrieving relevant context
53     - Avoid introducing new topics or queries that deviate from the original query
54     - DONT EVER ANSWER the Original query, but instead focus on rephrasing and expanding it into a new query
55
56     Return ONLY the rewritten query text, without any additional formatting or explanations.
57
58     Conversation History:
59     {context}
60
61     Original query: [{user_input}]
62
63     Rewritten query:
64     """

```

```

1      response = client.chat.completions.create(
2          model=ollama_model,
3          messages=[{"role": "system", "content": prompt}],
4          max_tokens=200,
5          n=1,
6          temperature=0.1,
7      )
8      rewritten_query = response.choices[0].message.content.strip()
9      return json.dumps({"Rewritten Query": rewritten_query})
10
11 def ollama_chat(
12     user_input,
13     system_message,
14     vault_embeddings,
15     vault_content,
16     ollama_model,
17     conversation_history):
18     conversation_history.append({"role": "user", "content": user_input})
19
20     if len(conversation_history) > 1:
21         query_json = {
22             "Query": user_input,
23             "Rewritten Query": ""
24         }
25         rewritten_query_json = rewrite_query(json.dumps(query_json), conversation_history, ollama_model)
26         rewritten_query_data = json.loads(rewritten_query_json)
27         rewritten_query = rewritten_query_data["Rewritten Query"]
28         print(PINK + "Original Query: " + user_input + RESET_COLOR)
29         print(PINK + "Rewritten Query: " + rewritten_query + RESET_COLOR)
30     else:
31         rewritten_query = user_input
32
33     relevant_context = get_relevant_context(rewritten_query, vault_embeddings, vault_content)
34     if relevant_context:
35         context_str = "\n".join(relevant_context)
36         print(CYAN + "Context Pulled from Documents: \n\n" + CYAN + context_str + RESET_COLOR)
37     else:
38         print(CYAN + "No relevant context found." + RESET_COLOR)
39
40     user_input_with_context = user_input
41     if relevant_context:
42         user_input_with_context = user_input + "\n\nRelevant Context:\n" + context_str
43
44     conversation_history[-1]["content"] = user_input_with_context
45
46     messages = [
47         {"role": "system", "content": system_message},
48         *conversation_history
49     ]
50
51     response = client.chat.completions.create(
52         model=ollama_model,
53         messages=messages,
54         max_tokens=2000,
55     )
56
57     conversation_history.append({"role": "assistant", "content": response.choices[0].message.content})
58
59     return response.choices[0].message.content
60
61 # Parse command-line arguments
62 print(NEON_GREEN + "Parsing command-line arguments..." + RESET_COLOR)
63 parser = argparse.ArgumentParser(description="Ollama Chat")
64 parser.add_argument("--model", default="llama3", help="Ollama model to use (default: llama3)")

```



```
1  args = parser.parse_args()
2
3  # Configuration for the Ollama API client
4  print(NEON_GREEN + "Initializing Ollama API client..." + RESET_COLOR)
5  client = OpenAI(
6      base_url='http://localhost:11434/v1',
7      api_key='llama3'
8  )
9
10 # Load the vault content
11 print(NEON_GREEN + "Loading vault content..." + RESET_COLOR)
12 vault_content = []
13 if os.path.exists("vault.txt"):
14     with open("vault.txt", "r", encoding='utf-8') as vault_file:
15         vault_content = vault_file.readlines()
16
17 # Generate embeddings for the vault content using Ollama
18 print(NEON_GREEN + "Generating embeddings for the vault content..." + RESET_COLOR)
19 vault_embeddings = []
20 for content in vault_content:
21     response = ollama.embeddings(model='mxbai-embed-large', prompt=content)
22     vault_embeddings.append(response["embedding"])
23
24 # Convert to tensor and print embeddings
25 print("Converting embeddings to tensor...")
26 vault_embeddings_tensor = torch.tensor(vault_embeddings)
27 print("Embeddings for each line in the vault:")
28 print(vault_embeddings_tensor)
29
30 # Conversation loop
31 print("Starting conversation loop...")
32 conversation_history = []
33 system_message = """You are a helpful assistant that is an expert at extracting
34 the most useful information from a given text.
35 Also bring in extra relevant information to the user query from outside the given context.
36 """
37
38 while True:
39     user_input = input(YELLOW + "Ask a query about your documents (or type 'quit' to exit): " + RESET_COLOR)
40     if user_input.lower() == 'quit':
41         break
42
43     response = ollama_chat(
44         user_input,
45         system_message,
46         vault_embeddings_tensor,
47         vault_content, args.model,
48         conversation_history)
49     print(NEON_GREEN + "Response: \n\n" + response + RESET_COLOR)
```