Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Trabajo Práctico 2

Santiago Ferrero

Tecnicatura en Inteligencia Artificial

Procesamiento del Lenguaje Natural

27/02/2024

Ejercicio 1:

El presente trabajo se enfoca en la creación de un chatbot que emplea la técnica RAG para obtener información relacionada con los Mundiales de Fútbol.

Fuentes de Información:
Para llevar a cabo este proyecto, se utilizaron archivos PDF como fuente de información. Estos archivos contenían datos obtenidos de distintas páginas de Wikipedia, que fueron obtenidos mediante técnicas de web scraping, y se puede acceder a ellos en el Colab complementario.

Datos del CSV:
El archivo CSV fue obtenido de Kaggle, a partir de los siguientes enlaces:

- FIFA World Cup Dataset
- FIFA World Cup 2022 Complete Dataset

Estos conjuntos de datos fueron sometidos a un proceso de limpieza y unificación de columnas utilizando la biblioteca Pandas. Este proceso se detalla en el archivo complementario adjunto.

Almacenamiento de Archivos:
Posteriormente, tanto los archivos PDF como el CSV fueron cargados en un almacenamiento en la nube, específicamente en Google Drive.

Base de Datos de grafos:
El grafo necesario para el funcionamiento del chatbot se obtuvo de Neo4j Bloom, una plataforma diseñada para explorar y visualizar grafos de manera intuitiva.
Para acceder a este grafo, se descargó un archivo con extensión ".dump", que luego fue cargado en una instancia de Neo4j en AuraDB.
AuraDB es una plataforma en la nube que ofrece servicios de Neo4j de manera gratuita. La base de datos se encuentra alojada en esta instancia, y es desde donde el chatbot obtiene la información necesaria.

Herramientas Utilizadas:
Para el desarrollo del chatbot, se emplearon herramientas de Langchain para procesamiento de lenguaje natural. Además, se utilizaron embeddings de texto en Chromadb, y la base de datos de Neo4j para la manipulación de grafos.

Descripción del Código:
El código comienza con la instalación e importación de las librerías necesarias. Luego, se procede a cargar los archivos desde Google Drive, donde se encuentran alojados.
El Drive cuenta con dos carpetas distintas, una para los archivos relacionados con los Mundiales masculinos y otra para los Mundiales femeninos.
También se cuenta con el archivo CSV en dicha plataforma.

```
!pip install pymupdf
!pip install pypdf
!pip install rdflib
!pip install langchain
!pip install python-decouple
!pip install tensorflow_text
!pip install chromadb
!pip install neo4j
!pip install sentence-transformers
!pip install stanza
```

```
import gdown
import os
import shutil
import sys
import pandas as pd
import datetime
import numpy as np
import tensorflow_text
import tensorflow_hub as hub
import langchain
import chromadb
import fitz
import re
import requests
import spacy
from sentence_transformers import SentenceTransformer, util
from langchain.text_splitter import CharacterTextSplitter
from rdflib import Graph, URIRef, Literal, Namespace
from sklearn.metrics.pairwise import cosine_similarity
```

```
url = 'https://drive.google.com/drive/u/1/folders/1O8H_PMY-nTO1X8p8xDdNmu_GnH5Agy-j'

gdown.download_folder(url, quiet=True, output='/content')
```

Luego se sigue con la limpieza del csv, donde se unifican los datos de la columna ‘date’ y se le da formato fecha, para que pueda pasarse como contexto, los resultados de los partidos de un año en particular.

```
df = pd.read_csv('/content/partidos.csv')

# Convertir la columna 'date' a un formato uniforme de fecha y hora
df['date'] = pd.to_datetime(df['date'], errors='coerce')

# Rellenar los valores NaT (Not a Time) con una fecha adecuada
df['date'] = df['date'].fillna(pd.to_datetime(df['date'], format='%d %b %Y', errors='coerce'))

# Convertir la columna 'date' al formato deseado "YYYY-MM-DD"
df['date'] = df['date'].dt.strftime('%Y-%m-%d')

# Convertir la columna 'date' a tipo datetime
df['date'] = pd.to_datetime(df['date'])
```

```
df = df.fillna('No hubo')
```

```
df = df.rename(columns = {'date':'fecha', 'home_team':'local','away_team':'visitante','home_score':'local','away_score':'visitante',
```

```
df.head(2)
```

	fecha	local	visitante	local	visitante	etapa	alargue o penales
0	1930-07-13	France	Mexico	4	1	Group 1	No hubo
1	1930-07-13	Belgium	United States	0	3	Group 4	No hubo

Se define el "splitter" como CharacterTextSplitter para separar el texto de los documentos en párrafos, facilitando así el proceso de embedding.

El modelo de embedding utilizado es el Universal Sentence Encoder, proporcionado por TensorFlow Hub. Este modelo puede ser encontrado en la página "https://tfhub.dev/google/universal-sentence-encoder-multilingual/3".

Textos

```
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=200, separator='\n')
```

```
#Establecemos el modelo de embedding

embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder-multilingual/3")
```

```
# Eliminar una coleccion
#client.delete_collection(name="all_my_documents")
```

```
# #Creación de la base con ChromaDB
client = chromadb.Client()
collection= client.get_or_create_collection("all_my_documents")
```

Luego se procede a crear una colección en ChromaDB y extraer por separado los textos de la carpeta de fútbol masculino y los de fútbol femenino. Esto se realiza con el fin de obtener los nombres de las fuentes y permitir que el chatbot maneje de manera más efectiva los resultados.

```
client = chromadb.Client()
collection= client.get_or_create_collection("all_my_documents")
```

```
textos = []
ids_textos = []
fuentes = []
lista_pdf = []
lista_pdf.extend(f for f in os.listdir('/content/Masculino')if f.endswith(".pdf"))

for i, pdf in enumerate(lista_pdf, start=1):
    pdf_path = os.path.join('Masculino', pdf)
    with fitz.open(pdf_path) as doc:
        text = ""
        for page_num in range(doc.page_count):
            page = doc[page_num]
            text += page.get_text()
        documents = text_splitter.split_text(text)
        textos.extend(documents)
        ids_textos.extend([f"masc{i}__{j}" for j in range(1, len(documents) + 1)])
        fuentes.extend([f"{pdf_path}" for _ in range(len(documents))])
```

```
embeddings = embed(textos).numpy().tolist()
# Para cada lote, realizar la adición a la colección
for batch, fuente, ids, embedding in zip(textos, fuentes, ids_textos, embeddings):
  collection.add(
        documents=batch,
        metadatas=[{'source':fuente}],
        ids= ids,
        embeddings=embedding )
```

Se hace lo mismo con los archivos de fútbol femenino.

Ahora se crea una función que realiza la búsqueda en Chroma, a la cual se le debe pasar el embedding de una consulta y devuelve los chunks con la distancia más cercana.
También se le puede pasar un año y entonces la consulta se hará solamente en los documentos que contengan ese año en la fuente.

```python
def query_chromadb(query_embed,anio=None):
  if anio:
    results = collection.query(
        query_embeddings=query_embed,
        n_results=3, # Traemos los 3 resultados más cercanos
        where = {"$or" : [
        {'source': f'Masculino/mundial_{anio}_wikipedia.pdf'},
        {'source': f'Femenino/mundial_{anio}_wikipedia.pdf'} ] } )
  else:
      results = collection.query(
      query_embeddings=query_embed,
      n_results=5)
  return results
```

Para la información obtenida de grafos se crea una función que se comunica con la base de datos alojada en Auradb y realiza una consulta en lenguaje Cypher.

Grafos

```python
] from neo4j import GraphDatabase

  hugginface_token = config('HUGGINGFACE_TOKEN')
  username=config("USERNAME")
  password=config("PASSWORD")

  URI = "neo4j+s://7fc395b0.databases.neo4j.io"
  AUTH = (username, password)
```

```python
  def run_query(query: str, parameters = None) :
      driver = GraphDatabase.driver(URI, auth=AUTH)
      try:
        results = driver.session().run(query,parameters)
        return results.data()
      except Exception as e:
        return ('No se puede responder.')
```

Se utiliza Llama-index para comunicarse con la base de datos y obtener un esquema de esta, que luego será pasado a un llm para que pueda entender la estructura y generar consultas Cypher por su cuenta.

```
!pip install llama-index-graph-stores-neo4j
from llama_index.core import StorageContext
from llama_index.graph_stores.neo4j import Neo4jGraphStore

graph_store = Neo4jGraphStore(
    username=username,
    password=password,
    url=URI,
    database='neo4j',
)

storage_context = StorageContext.from_defaults(graph_store=graph_store)

schema = graph_store.get_schema()
```

Ahora se define la función que prepara el prompt para la creación de la consulta.

```
def prompt_cypher(query_str):
    PROMPT_CYPHER_TMPL ="""
Esquema: {esquema}

Aqui tienes tres ejemplos para que veas como debes responder:
ej 1:
MATCH (t:Tournament {{year: 2015}})<-[:PARTICIPATED_IN]-(team)
RETURN team.name as seleccion

ej 2:
MATCH (s:Squad)-[:FOR]->(t:Tournament {{year: 2003}})
MATCH (p:Person)-[:IN_SQUAD]->(s)
MATCH (p)-[:SCORED_GOAL]->(m:Match)
MATCH n = (m)-[:IN_TOURNAMENT]->(t)
WHERE toLower(p.name) CONTAINS Marta
WITH  count(n) as goals, p.name AS Person, s.id as seleccion
RETURN Person, seleccion,  goals

ej 3:
MATCH (s:Squad)-[:FOR]->(t:Tournament)
MATCH (p:Person)-[:IN_SQUAD]->(s)
MATCH (p)-[:SCORED_GOAL]->(m:Match)
MATCH n = (m)-[:IN_TOURNAMENT]->(t)
WHERE toLower(p.name) CONTAINS 'Pottasa'
WITH  count(n) as goals, p.name AS Person, s.id as seleccion
RETURN Person, seleccion,  goals


Responde solo a la pregunta de abajo, exclusivamente en lenguaje cypher, no dar explicaciones. verbose=no

Pregunta: "{pregunta}"
    """
    messages = [
```

```
    messages = [

      {"role": "assistant",
      "content": """
Eres un experto traductor de Cypher, tienes que convertir la pregunta que se te pase a Cypher.

Recuerda siempre basarte en el esquema de Neo4j proporcionado, siguiendo las instrucciones a continuación:

1. Genera una consulta Cypher compatible ÚNICAMENTE para la Versión 5 de Neo4j.

2. No utilices las palabras clave EXISTS, HAVING, SIZE. Usa alias al utilizar la palabra clave WITH.

3. Realiza siempre una búsqueda insensible a mayúsculas y minúsculas y difusa para cualquier búsqueda relacionada con propiedades. Por ejemplo,

4. Utiliza solo nodos y relaciones mencionados en el esquema proporcionado.
      """},

        {"role": "user", "content":  PROMPT_CYPHER_TMPL.format(esquema=schema, pregunta=query_str)}

       ]

    final_prompt = zephyr_chat_template(messages)
    return final_prompt
```

Luego se definen 3 estilos de querys en lenguaje Cypher que son utilizadas cuando el clasificador así lo determine, y se le pueden pasar parámetros para los años o nombres propios.

```
cypher_query = '''
MATCH (t:Tournament {year: $year})<-[:PARTICIPATED_IN]-(team)
RETURN team.name as seleccion
'''
#run_query(cypher_query, {'year':2019})

cypher_query2 = """
MATCH (s:Squad)-[:FOR]->(t:Tournament {year: $year})
MATCH (p:Person)-[:IN_SQUAD]->(s)
MATCH (p)-[:SCORED_GOAL]->(m:Match)
MATCH n = (m)-[:IN_TOURNAMENT]->(t)
WHERE toLower(p.name) CONTAINS $name
WITH  count(n) as goals, p.name AS Person, s.id as seleccion
RETURN Person, seleccion,  goals
"""
#run_query(cypher_query2, {'name':'ma', 'year':2015})

cypher_query3 = """
MATCH (s:Squad)-[:FOR]->(t:Tournament)
MATCH (p:Person)-[:IN_SQUAD]->(s)
WHERE toLower(p.name) CONTAINS $name
WITH   p.name AS Person, p.dob as nacimiento, s.id as seleccion
RETURN Person, seleccion, nacimiento
"""
#run_query(cypher_query3, {'name':'pot'})
```

Clasificador:

```python
def classifier(query_str: str) -> str:
    query_embeddings = embed([query_str]).numpy().tolist()

#veo cuantos nombres propios (paises) y anio hay en la query
    doc = nlp(query_str)
    paises = []
    for i, sent in enumerate(doc.sentences):
      cont = 0
      for word in sent.words:
        if word.pos == 'PROPN':
          cont += 1
          paises.append(word.text)
    patron = r"\d{4}"
    anio = re.findall(patron, query_str)
    anio = int(anio[0]) if anio != [] else None

#si la consulta contiene la palabra femenino, se busca en chroma fem
    referencia_fem = "femenino"
    referencia = embed([referencia_fem]).numpy().tolist()
    max_cosine_similarity_fem = -1000
    for query in query_str.split():
      embedding = embed([query]).numpy().tolist()
      similarity_fem = cosine_similarity(embedding, referencia)
      if similarity_fem[0][0] > 0.85 :
        #print('femenino')
        return classifier_fem(query_str, query_embeddings, paises, anio)


#si en la query se especifica un anio y dos paises, se busca resultado en csv
    if anio and cont == 2:
        df['fecha'] = pd.to_datetime(df['fecha'])
        context_csv = df[df['fecha'].dt.year == anio]
        #print('csv')
        return context_csv

#si no, se busca en chroma
```

```python
    #si no, se busca en chroma
        documents = query_chromadb(query_embeddings, anio)
        context_chroma = "\n".join([f"{doc}" for documento in documents["documents"] for doc in documento])
        return context_chroma
```

A esta función se le pasa la pregunta hecha por el usuario.

Procesamiento de la consulta y extracción de características:

- Se incrusta la consulta utilizando un modelo previamente entrenado, y se convierte en una lista de características (vectores).
- Se utiliza el modelo de Stanza para identificar nombres propios (paises) y se almacenan en la lista paises.
- Se utiliza una expresión regular para identificar un año en la consulta y se almacena en la variable anio.

Determinación del contexto de la consulta:

- Se busca la palabra "femenino" en la consulta. Si se encuentra, se invoca la función classifier_fem para manejar la consulta específica relacionada con el fútbol femenino.
- Si la consulta contiene un año y exactamente dos nombres propios (paises), se realiza una búsqueda en un archivo CSV para encontrar información relacionada con ese año y esos países.
- Si no se cumplen las condiciones anteriores, se utiliza la base de datos de ChromaDB para buscar información relacionada con la consulta. Se compara la similitud entre la consulta y los datos en la base de datos.

Generación de respuesta:

- Se elige la fuente de datos (CSV o ChromaDB) que tenga una mayor similitud con la consulta.
- Se devuelve el contexto encontrado como respuesta.

```python
def classifier_fem(query_str, query_embeddings, jugadora = None, anio=None) -> str:
  similarity1 = 0
  similarity2 = 0
  similarity3 = 0

  if anio and jugadora:
    jugadora =  ' '.join(jugadora).lower()
    referencia1= "cuantos goles hizo "+ jugadora +" en el mundial femenino "+ str(anio)
    embedding1 = embed(referencia1).numpy().tolist()
    similarity1 = cosine_similarity(np.array(query_embeddings).reshape(1, -1), np.array(embedding1).reshape(1, -1))[0][0]
    #print('1',similarity1)
    if similarity1 > 0.7:
      #anio = int(anio[0])
      contexto1 = run_query(cypher_query2, {'name': jugadora,'year':anio})
      #print('contexto1: ', contexto1)
      if contexto1 != [] :
        return contexto1
      else:
        return "No se encontro a nadie llamada "+ jugadora +" que haya marcado goles en este torneo."

  elif anio:
    referencia2= "que selecciones participaron en el mundial femenino del anio "+ str(anio)
    embedding2 = embed(referencia2).numpy().tolist()
    similarity2 = cosine_similarity(np.array(query_embeddings).reshape(1, -1), np.array(embedding2).reshape(1, -1))[0][0]
    #print('2',similarity2)
    if similarity2 > 0.7:
      #anio = int(anio[0])
      contexto2 = run_query(cypher_query, {'year':anio})
      #print('contexto2: ', contexto2)
      if contexto2 != [] :
        return contexto2
      else:
        return "En el año "+ str(anio) +" no se disputó ningun mundial femenino"

  elif jugadora:
```

```
  elif jugadora:
    jugadora =  ' '.join(jugadora).lower()
    referencia3= ["dame informacion sobre la jugadora "+ jugadora, "para que seleccion de futbol femenino juega "+jugadora,
                  "cuantos años tiene hoy la jugadora de futbol femenino "+jugadora]
    for referencia in referencia3:
      ref_embedding3 = embed([referencia]).numpy().tolist()
      max_similarity3 = cosine_similarity(query_embeddings, ref_embedding3)[0][0]
      if max_similarity3 > similarity3:
        similarity3 = max_similarity3
    #print('3',similarity3)
    if similarity3 > 0.7:
      contexto3 = run_query(cypher_query3, {'name': jugadora})
      #print('contexto3: ', contexto3)
      if contexto3 != [] :
        return contexto3
      else:
        return "No se conoce a nadie llamada "+ jugadora +" fijese que haya escrito bien el nombre."


#creo query con llm
  similarity_graph = 0
  consulta = prompt_cypher(query_str)
  query = generate_answer(consulta)
  #print('llm: ', query)
  context_graph = run_query(query)
  #print('contexto graph: ', context_graph)
  try:
    embedding_graph = embed(context_graph).numpy().tolist()
    similarity_graph = cosine_similarity(query_embeddings, embedding_graph)[0][0]
    #print('graph',similarity_graph)
  except:
    pass

#femenino, texto
  documents = query_chromadb(query_embeddings, anio)
  diference_min_chromadb = min(abs(x - 1) for x in documents["distances"][0])
```

```
#femenino, texto
  documents = query_chromadb(query_embeddings, anio)
  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
  #print('chroma',similarity_chroma)

  if similarity_chroma > similarity_graph:
    return context_chroma
  else:
    return context_graph
```

Procesamiento de consultas con jugadoras y año: Si se proporcionan tanto el nombre de una jugadora como un año, se ejecuta la siguiente lógica:

- Se normaliza el nombre de la jugadora.
- Se construye una referencia sobre la consulta de cuántos goles hizo esa jugadora en un año específico.
- Se calcula la similitud coseno entre la consulta embebida y la referencia construida.
- Si la similitud es alta (más de 0.7), se realiza una consulta a una base de datos concreta (contexto1), y si se encuentra información, se devuelve. De lo contrario, se informa que no se encontró información sobre la jugadora y el año especificado.

Procesamiento de consultas solo con año: Si solo se proporciona el año, se realiza una lógica similar a la anterior pero para consultar qué selecciones participaron en un Mundial Femenino específico.

Procesamiento de consultas solo con jugadora: Si solo se proporciona el nombre de una jugadora, se ejecuta la siguiente lógica:

- Se construyen varias referencias relacionadas con la jugadora.
- Se calcula la similitud coseno entre cada referencia y la consulta embebida, conservando la mayor similitud encontrada.
- Si la similitud es alta (más de 0.7), se realiza una consulta, y si se encuentra información, se devuelve. De lo contrario, se informa que no se encontró información sobre la jugadora especificada.

Generación de respuesta con un modelo de lenguaje: Se utiliza un modelo de lenguaje para generar una consulta en Cypher y obtener contexto de la base de datos de grafos.

.

Decisión de retorno: Se decide qué información retornar basándose en la similitud entre la consulta y los datos en la base de datos versus la generada por el modelo de lenguaje.

```python
from jinja2 import Template

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)
```

En la función de arriba se define un template para formalizar la forma en que le paso al modelo LLM el contenido de las instrucciones,la consulta y el contexto.

```
def prepare_prompt(query_str, context_prompt, fecha):

    TEXT_QA_PROMPT_TMPL = ("""
Contexto:
{context}

Pregunta: {pregunta}
Respuesta:
     """
    )

    messages = [

      {"role": "system",
      "content": f"""
Eres un chatbot diseñado para responder a preguntas sobre los mundiales de futbol.

La respuesta debe ser corta y concisa, y mantener una estructura coherente.

Utiliza solo la información proporcionada en el contexto para generar la respuesta.

Si consideras que el contexto no es suficiente para responder a la pregunta, simplemente responde: "No es posible responder esa pregunta. Por favor, formula una nueva".

No digas nada sobre el contexto proporcionado.

Tienes conocimiento actualizado hasta la fecha de hoy, que es: {fecha}. Por lo tanto el ultimo mundial fue el de 2022.

Proporciona únicamente la respuesta a la pregunta, sin comentarios adicionales. verbose=no

Responde siempre en español.

     """},
```

```
     """},

       {"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(context=context_prompt, pregunta=query_str)},

      ]

   final_prompt = zephyr_chat_template(messages)
   return final_prompt
```

Esta es la función que prepara el prompt, con las instrucciones, y la información.
Aquí se define cómo debe comportarse el modelo y cómo responder.

```python
def generate_answer(prompt: str, max_new_tokens: int = 600) -> None:
    try:
        api_key = hugginface_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": 0.1,
                "top_k": 5,
                "top_p": 0.25
            }
        }

        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}")
```

Aquí en esta función es donde se define el modelo que genera nuestras respuestas, tanto de la creación de la consulta en Cypher como la respuesta que se le da al usuario.

En este caso es Zephyr-7b-beta, obtenido de HugginFace, aqui tambien se le definen los distintos parámetros, los cuales quiero que responda lo más preciso posible sin tanta creatividad.

```
print('**Bienvenidos al chat sobre los mundiales de futbol**\n')
print('Le pedimos que sea claro con su consulta al querer hablar de futbol femenino o masculino, ')
print('tambien que por favor utilize las mayusculas en los nombres propios. \n')
print('Las preguntas puede estar relacionadas con los siguiente temas: \n')
print('-Selecciones participantes, resultados de partidos masculinos, jugadoras de futbol femenino, curiosidades de los mundiales\n')


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

    hoy = datetime.date.today()
    context_final = classifier(user_query)
    final_prompt = prepare_prompt(user_query, context_final, hoy)
    print('Respuesta:')
    print(generate_answer(final_prompt))
    print('-------------------------------------------------------')
```

Este es el bucle principal donde se llaman a todas las funciones y se puede realizar la conversación, permite ingresar una consulta y muestra la respuesta.

Aqui hay un ejemplo del funcionamiento:

```
**Bienvenidos al chat sobre los mundiales de futbol**

Le pedimos que sea claro con su consulta al querer hablar de futbol femenino o masculino,
tambien que por favor utilize las mayusculas en los nombres propios.

Las preguntas puede estar relacionadas con los siguiente temas:

-Selecciones participantes, resultados de partidos masculinos, jugadoras de futbol femenino, curiosidades de los mundiales

Ingrese su consulta (o 'salir' para salir): que puedes contarme del mundial 1986, curiosidades
Respuesta:
En la Copa Mundial de Fútbol de 1986, que se desarrolló en México, hubo dos hechos destacados y sin precedentes en el fútbol. En el partido Arge
-------------------------------------------------------
Ingrese su consulta (o 'salir' para salir):
```

Ejercicio 2:

Los Agentes Inteligentes, respaldados por Modelos de Lenguaje Grande (LLM), están en la vanguardia de la tecnología, desempeñando un papel fundamental en una amplia gama de aplicaciones. Estos agentes, diseñados para interactuar con el entorno, procesar información y tomar decisiones de manera autónoma, han revolucionado diversos sectores con su capacidad para comprender el lenguaje natural y realizar tareas complejas. A continuación, se presenta una investigación sobre el estado del arte de las aplicaciones actuales de agentes inteligentes que utilizan modelos LLM libres.

Definición de Agentes Inteligentes:
Los agentes inteligentes son programas informáticos diseñados para interactuar con su entorno, procesar información y tomar decisiones de forma autónoma para llevar a cabo una determinada tarea. Estos agentes son capaces de construir su propio conocimiento a partir de fuentes existentes, razonar sobre él y evolucionarlo a través de la retroalimentación obtenida del entorno y los usuarios.

Integración de Modelos de Lenguaje Grande (LLM) en Agentes Inteligentes:
Los Modelos de Lenguaje Grande (LLM) han mejorado la capacidad de los agentes inteligentes para comprender el lenguaje natural y realizar tareas más complejas. Logrando que dos o más agentes puedan comunicarse y darse indicaciones a través del lenguaje natural.

Los agentes son un tema de muchos debates en línea, particularmente con respecto a ser inteligencia general artificial (AGI). Los agentes usan un LLM avanzado como 'gpt4' o 'PaLM2' para planificar tareas en lugar de tener cadenas predefinidas. Entonces, ahora, cuando hay solicitudes de usuarios, según la consulta, el agente decide qué conjunto de tareas llamar y construye dinámicamente una cadena.

Los agentes pueden usar herramientas con las que el modelo de lenguaje puede interactuar. Algunos ejemplos de estas herramientas son buscar en google, ejecutar código, extraer información de Wikipedia y mucho más.

Algunos de los sistemas que están revolucionando el campo de los agentes LLM por su facilidad de uso y la creación de agentes potentes son:

1. AutoGen: Este proyecto de Microsoft Research permite a los desarrolladores construir equipos de agentes de IA multi-tarea para lograr casi cualquier objetivo. Los agentes de IA pueden escribir y ejecutar código, criticarse mutuamente, planear, adoptar diferentes roles y personalidades, y más. AutoGen destaca por su capacidad para ejecutar iteraciones automatizadas y ejecutar código en un entorno. Además, con múltiples agentes de IA trabajando juntos, pueden usar herramientas como bibliotecas de Python.

2. Crew AI: Esta herramienta de inteligencia artificial permite automatizar tareas al establecer equipos de agentes. Crew AI ha ganado popularidad rápidamente con más de 1300 estrellas en GitHub. Organiza múltiples agentes inteligentes en un

equipo, enfatizando la colaboración para asegurar un rendimiento sin problemas durante la ejecución de tareas. Los agentes poseen excelentes capacidades de comunicación y colaboración.

3. ChatDev: Este enfoque innovador basado en un modelo de lenguaje grande (LLM) busca revolucionar el campo del desarrollo de software. ChatDev elimina la necesidad de modelos especializados durante cada fase del proceso de desarrollo. Utiliza la comunicación en lenguaje natural para unificar y optimizar procesos clave de desarrollo de software. Cada etapa del proceso de desarrollo implementa un equipo de agentes virtuales, como programadores de código o evaluadores, que colaboran entre sí mediante diálogos que dan como resultado un flujo de trabajo fluido.

4. LangChain: Los agentes en LangChain son entidades inteligentes que actúan como intermediarios entre el usuario y las herramientas disponibles en la plataforma. Estos agentes son responsables de procesar las entradas del usuario, decidir qué herramientas utilizar y cómo interactuar con ellas, y finalmente devolver una respuesta adecuada al usuario. La idea central de los agentes es utilizar un modelo de lenguaje para elegir una secuencia de acciones a realizar. En LangChain, un agente es un envoltorio alrededor de un modelo que recibe la entrada de un usuario y responde con una “acción” a tomar y una correspondiente “entrada de acción”, por lo que es capaz de "moverse” a través de diferentes herramientas para ejecutar tareas de forma automática.

Problemática a resolver:
Gestión de Inventarios en una Cadena de Suministro

Descripción:
La gestión de inventarios en una cadena de suministro puede ser compleja debido a la variabilidad en la demanda, los tiempos de entrega y otros factores. La falta de coordinación entre los diferentes nodos de la cadena de suministro puede llevar a problemas como exceso o falta de inventario, costos elevados de almacenamiento y pérdida de ventas debido a la falta de productos disponibles.

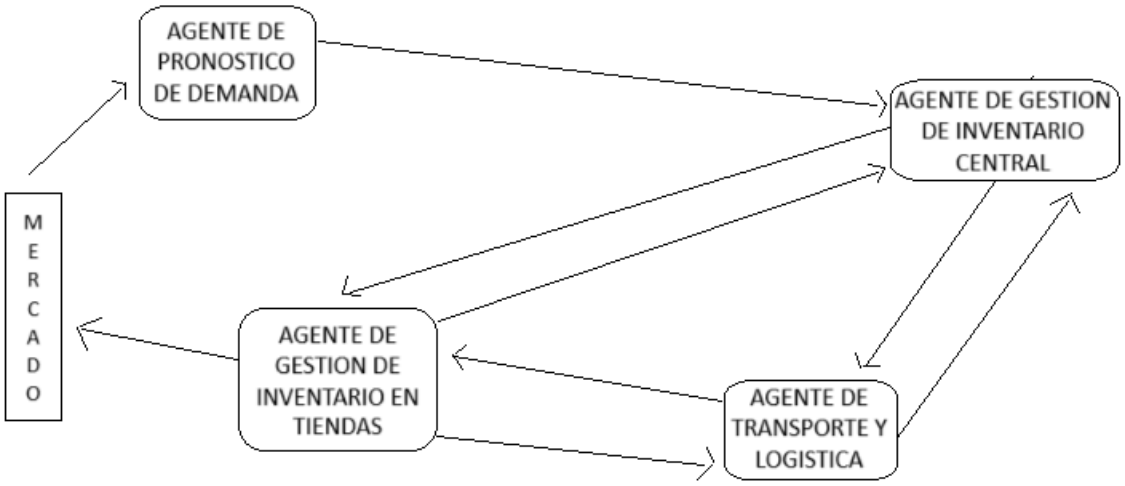
Objetivo:
Desarrollar un sistema multiagente que permita una gestión eficiente de inventarios en toda la cadena de suministro, optimizando los niveles de inventario, reduciendo los costos de almacenamiento y maximizando la disponibilidad de productos para satisfacer la demanda del cliente.

Agentes Involucrados:
- Agente de Pronóstico de Demanda: Utiliza datos históricos de ventas y otros factores relevantes para predecir la demanda futura de productos.
- Agente de Gestión de Inventario Central: Coordina los niveles de inventario en los almacenes centrales y toma decisiones sobre la reposición de productos.

- Agente de Gestión de Inventario en Tiendas: Gestiona los niveles de inventario en las tiendas minoristas y realiza pedidos de reposición según la demanda local.
- Agente de Transporte y Logística: Coordinar la entrega de productos entre los diferentes nodos de la cadena de suministro, optimizando rutas y tiempos de entrega.

Implementación de Tecnologías:
El sistema puede utilizar tecnologías como el análisis de datos en tiempo real, el aprendizaje automático para mejorar los pronósticos de demanda y la optimización de rutas de transporte utilizando algoritmos genéticos.
También puede hacer uso de sistemas de identificación por radiofrecuencia (RFID) para el seguimiento en tiempo real de los productos a lo largo de la cadena de suministro. Esto proporcionaría una mayor visibilidad y precisión en la gestión de inventarios, permitiendo una respuesta más rápida a los cambios en la demanda y en los niveles de inventario.

Esquema:



Conversación de Ejemplo entre Agentes:
[Agente de Pronóstico de Demanda] mensaje al Agente de Gestión de Inventario Central: "Se espera un aumento en la demanda del producto X para la próxima semana."

[Agente de Gestión de Inventario Central] mensaje al Agente de Gestión de Inventario en Tiendas: "Hemos recibido un pronóstico de aumento en la demanda del producto X para la próxima semana. Por favor, revisa los niveles de inventario en cada tienda y dime cuántos productos necesitan para cubrir esta demanda adicional."

[Agente de Gestión de Inventario en Tiendas] Ejecutando: Revisando los niveles de inventario en cada tienda y solicitando la cantidad necesaria de producto X para cubrir el

aumento de demanda; mensaje al Agente de Gestión de Inventario Central: "Tienda A necesita 50 productos, Tienda B necesita 30 productos y Tienda C necesita 70 productos para cubrir el aumento de demanda."

[Agente de Gestión de Inventario Central] mensaje al Agente de Transporte y Logística: "Necesitamos organizar el transporte de 50 productos a Tienda A, 30 productos a Tienda B y 70 productos a Tienda C debido al aumento de demanda del producto X. Por favor, asegúrate de encontrar los transportes adecuados y optimizar las rutas de entrega."

[Agente de Transporte y Logística] mensaje a Tiendas: "Los productos serán enviados el próximo martes. Por favor, asegúrense de tener personal disponible para recibir la mercancía y descargarla a las 10:00 a.m. en tienda A, a las 11:00 a.m. en tienda B y a las 12:30 en tienda C "

[Agente de Gestión de Inventario en Tiendas] Ejecutando: Avisando a cada tienda del horario de llegada de los productos: mensaje al Agente de Transporte y Logística: "Todo está organizado para recibir los productos el próximo martes en todas las tiendas."

[Agente de Transporte y Logística] mensaje al Agente de Gestión de Inventario Central: "La entrega está programada sin problemas para el próximo martes en todas las tiendas."

Fuentes

https://www.unite.ai/es/c%C3%B3mo-los-grandes-modelos-de-lenguaje-llm-impulsar%C3%A1n-las-aplicaciones-del-futuro/

https://www.dimensionia.com/autogen/

https://www.consultor365.com/ia/microsoft-autogen/

https://github.com/microsoft/autogen

https://www.toolify.ai/es/ai-news-es/aprende-a-usar-crewai-para-equipos-de-agentes-de-ia-de-nueva-generacin-completamente-local-537043

https://crewai.net/

https://github.com/joaomdmoura/crewAI?tab=readme-ov-file#connecting-your-crew-to-a-model

https://www.unite.ai/es/agentes-comunicativos-chatdev-para-el-desarrollo-de-software/

https://brainq.ai/agentes-langchain/

https://python.langchain.com/docs/modules/agents/

https://ichi.pro/es/una-guia-completa-de-langchain-creacion-de-aplicaciones-potentes-con-modelos-de-lenguaje-grandes-92003914344427

https://www.ibm.com/es-es/topics/langchain