Universidad de León

Grado en Ingeniería Informática

Memoria RAG
Recomendador inteligente de vehículos

Autor: Ignacio gil Rojo

Tutor: Enrique López González

Fecha: 10 de enero de 2026

El repositorio con la base de datos está disponible aquí (GitHub).

Índice general

Capítulo 1

Introducción

En este proyecto se ha desarrollado un asistente conversacional inteligente capaz de recomendar vehículos combinando un grafo de conocimiento en Neo4j con un modelo de lenguaje alojado en Ollama. El grafo almacena la información estructurada de los modelos (precio, potencia, autonomía, tipo de cambio, tracción y puntuaciones de uso), mientras que el modelo de lenguaje se encarga de interpretar las consultas en lenguaje natural y generar explicaciones comprensibles para el usuario. La aplicación se expone mediante una interfaz web en Streamlit que permite una interacción natural en lenguaje libre y mantiene memoria de la conversación para refinar progresivamente las recomendaciones. Gracias a esta memoria, el sistema es capaz de recordar preferencias y temas mencionados en mensajes anteriores, de forma que las siguientes respuestas pueden adaptarse mejor al perfil y al contexto de cada usuario.

Capítulo 2

Problema

En el contexto actual, los configuradores de coches y buscadores tradicionales obligan al usuario a navegar por numerosos filtros técnicos y catálogos poco explicativos, lo que dificulta encontrar un modelo alineado con su uso real y su presupuesto. Este enfoque exige que la persona conozca de antemano conceptos como segmentos, potencias mínimas o tipos de motorización, y que vaya probando combinaciones de filtros hasta acercarse a algo que encaje con sus necesidades, con un proceso poco guiado y a menudo frustrante.

Capítulo 3

Objetivo

El objetivo de este trabajo es diseñar e implementar un sistema de recomendación de vehículos que:

- Entienda consultas en lenguaje natural y extraiga criterios estructurados de manera automática.
- Ser capaz de adaptarse a cualquier nivel de conocimineto del usuario y adaptarse para ayudar.
- Combine un grafo de conocimiento automovilístico con un modelo de lenguaje para ofrecer recomendaciones explicadas.
- Aproveche la memoria de la conversación para ir aprendiendo preferencias y refinar resultados.

Capítulo 4

Solución propuesta

Para resolver el problema planteado se propone una arquitectura de tipo RAG en la que:

- Un módulo de extracción de criterios interpreta las consultas del usuario y el contexto previo para generar filtros estructurados sobre el dominio de vehículos.
- Un motor de recuperación filtra y puntúa vehículos en Neo4j de acuerdo con estos criterios, aplicando restricciones de precio, potencia, autonomía, cambio, tracción, marcas y tipo de vehículo, etc.
- Un componente de memoria conversacional almacena mensajes, filtros aplicados, preferencias detectadas y temas de interés, enriqueciendo las siguientes consultas.
- Una capa generativa basada en Ollama construye respuestas amigables, justificadas y comparativas a partir de los vehículos filtrados y los criterios extraídos.
- Una interfaz de chat en Streamlit integra todos los módulos y ofrece una experiencia interactiva sencilla para el usuario final.

Esta separación entre recuperación estructurada y generación de lenguaje facilita evolucionar cada parte de forma independiente (por ejemplo, cambiando el modelo de lenguaje o mejorando la lógica de filtrado) sin reescribir todo el sistema.

Capítulo 5

Tecnologías y uso

5.1. Tecnologías utilizadas

El sistema se apoya en las siguientes tecnologías:

- **Neo4j** como base de datos de grafo para almacenar los nodos `MODELO` y sus relaciones, consultados desde `car_recommender.py`.
- **Python** como lenguaje principal de implementación de todos los módulos.
- **Streamlit** para la interfaz web de chat interactivo y la gestión del estado de sesión.
- **Ollama** como backend de LLM y modelo de *embeddings*, configurado mediante el módulo `config.py` y utilizado por la clase `CarRecommender`.
- **LlamaIndex** como capa de integración con Ollama, proporcionando las clases `Ollama` y `OllamaEmbedding` usadas para la generación de respuestas y el cálculo de *embeddings* de texto.
- **dotenv** y **logging** para la carga de configuración desde entorno y el registro de eventos.

5.2. Uso e importancia del sistema

El asistente se utiliza mediante una interfaz de chat donde el usuario describe qué coche busca (tipo, uso, presupuesto, potencia, autonomía, marca, etc.) en lenguaje natural. La importancia de la solución radica en que traduce estas descripciones informales en criterios estructurados, consulta un grafo de vehículos y devuelve recomendaciones explicadas, mejorando la experiencia frente a buscadores basados únicamente en filtros manuales.

Capítulo 6

Arquitectura general

6.1. Visión global

La arquitectura se organiza en cuatro componentes principales:

- **Módulo de configuración** (`config.py`): centraliza parámetros de conexión a Neo4j, Ollama, y constantes de negocio como tipos de vehículo, tipos de motor y tópicos de uso.
- **Recomendador** (`car_recommender.py`): se conecta a Neo4j, carga la base de vehículos, extrae criterios, filtra modelos y genera respuestas inteligentes.
- **Gestor de memoria** (`memory_manager.py`): mantiene el contexto conversacional, temas de interés y preferencias aprendidas del usuario.
- **Aplicación web** (`app.py`): implementa el chatbot en Streamlit, orquesta las llamadas al recomendador y a la memoria, y presenta las recomendaciones.

La solución se estructura siguiendo una separación clara entre capa de presentación (chat en Streamlit), capa de lógica de negocio (recomendador y memoria) y capa de datos (grafo en Neo4j y configuración).

6.2. Flujo de una interacción típica

El flujo desde el punto de vista del usuario es el siguiente:

1. El usuario introduce un mensaje en la interfaz de chat (por ejemplo: "Busco un SUV híbrido, máximo 40k para viajes largos").
2. La aplicación registra el mensaje en el historial, lo envía al `MemoryManager` y obtiene un contexto textual resumido de la conversación.
3. El `CarRecommender` recibe la consulta y el contexto, extrae criterios estructurados y busca vehículos que cumplan los filtros en la base cargada desde Neo4j.
4. Si los criterios son suficientes, el recomendador genera una respuesta final usando el LLM; si no lo son, genera preguntas aclaratorias.
5. La respuesta se muestra en la interfaz.

Capítulo 7

Análisis DAFO

7.1. Fortalezas

- **Arquitectura modular**: separación clara entre configuración, recomendador, memoria conversacional y capa de presentación, lo que facilita el mantenimiento y la evolución del proyecto.
- **Comprensión de lenguaje natural**: el módulo de extracción de criterios detecta marcas, tipos de vehículo, tópicos de uso, motor, cambio, tracción, presupuesto, potencia y autonomía entre otros a partir del texto del usuario y del contexto previo.
- **Uso de memoria conversacional**: el gestor de memoria aprende temas y preferencias, y las reutiliza en consultas posteriores, generando una experiencia más coherente y personalizada.
- **Explicabilidad**: la combinación de filtros explícitos sobre la base de vehículos y generación controlada de texto permite justificar por qué se recomienda cada modelo y sugerir ajustes de criterios.

7.2. Debilidades

- **Dependencia de la calidad del grafo**: la precisión de las recomendaciones depende directamente de la cobertura y limpieza de los datos de modelos y relaciones almacenados en Neo4j.
- **Heurísticas simples en algunos filtros**: ciertos filtros, como el de tipo de vehículo a partir del nombre del modelo, se basan en reglas heurísticas que pueden fallar en casos límite o nombres poco convencionales.
- **Uso parcial de atributos**: aunque el código contempla motores y tracción, todavía no se explotan todos los posibles campos del grafo (por ejemplo, motor detallado o equipamiento).
- **Coste computacional**: la combinación de carga inicial de todos los vehículos más llamadas al LLM puede suponer latencias apreciables en entornos con recursos limitados.

7.3. Oportunidades

- **Ampliación del dominio**: la misma arquitectura podría aplicarse a otros catálogos de producto (motos, vehículos industriales o incluso otros sectores) reutilizando buena parte de la lógica.

- **Integración de datos dinámicos**: se pueden incorporar precios en tiempo real, disponibilidad por concesionario u ofertas, mejorando el valor práctico de las recomendaciones.
- **Mejora de la evaluación**: incluir métricas específicas de sistemas RAG y de recomendación permitiría optimizar el ranking y ajustar los pesos de `score_*`.
- **RAG avanzado sobre grafos**: explorar variantes como *GraphRAG* o consultas semánticas más ricas sobre Neo4j puede aumentar la calidad en consultas complejas y multimodales.

7.4. Amenazas

- **Evolución rápida de modelos**: la aparición continua de nuevos modelos de lenguaje y herramientas puede dejar obsoletas algunas dependencias actuales (por ejemplo, versiones concretas de Ollama o bibliotecas de integración).
- **Cambios en catálogos y precios**: la información de vehículos queda desactualizada con rapidez, lo que puede afectar a la confianza del usuario si no se mantiene un proceso de actualización de datos.
- **Requisitos de cumplimiento**: en un entorno real de concesionarios o fabricantes, podrían aparecer restricciones legales, de privacidad o de responsabilidad sobre las recomendaciones que obliguen a adaptar la arquitectura.
- **Competencia tecnológica**: otros sistemas de recomendación con acceso a datos de mercado más completos o modelos propietarios más potentes podrían ofrecer resultados superiores si no se continúa evolucionando la solución.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Conclusiones

El sistema desarrollado aborda el problema de la búsqueda de vehículos mediante formularios rígidos, sustituyéndolos por un chatbot que entiende lenguaje natural y combina recuperación estructurada con generación de texto.

8.2. Líneas de mejora y avances futuros

Como trabajo futuro se proponen las siguientes mejoras:

- Incorporar filtrado explícito por tipo de motor cuando el grafo incluya dicho atributo de forma normalizada.
- Añadir un módulo de evaluación con usuarios finales para medir la calidad percibida de las recomendaciones y ajustar los pesos de los `score_*`.
- Integrar información dinámica sobre disponibilidad, ofertas o tiempos de entrega, enriqueciendo el contexto que se pasa al modelo generativo.
- Explorar variantes avanzadas de RAG (por ejemplo, *GraphRAG* o enfoques adaptativos) para mejorar la relevancia en consultas complejas.
- Optimizar el rendimiento mediante técnicas de cacheo, despliegue en GPU u optimización de consultas a Neo4j para reducir la latencia en entornos de producción.
- Hacer uso de una base o API actualizada y no de un grafo de nodos en local.