Informe de Proyecto:
LexiQuipu - RAG sobre Sentencias Judiciales con LlamaIndex

Franklin E.P.

22 de julio de 2025

Resumen

Este informe detalla la implementación de un sistema de Chatbot basado en la arquitectura de Generación Aumentada por Recuperación (RAG), especializado en sentencias jurisprudenciales peruanas. El sistema aprovecha un script de preprocesamiento personalizado para la limpieza de texto y extracción de metadatos, y utiliza el framework **LlamaIndex** para orquestar la indexación y recuperación de información. El objetivo es responder preguntas complejas basándose en un corpus de documentos legales, combinando la flexibilidad del código propio con la robustez del framework.

1. Introducción y Objetivo

El objetivo central de este proyecto es construir una aplicación RAG capaz de dialogar sobre el contenido de sentencias judiciales peruanas. A diferencia del enfoque genérico funcional anterior, este proyecto pone énfasis en la calidad de los datos ingeridos, utilizando un pipeline de procesamiento a medida y haciendo uso del framework LlamaIndex.

Los objetivos específicos son:

- **Procesamiento de Documentos a Medida:** Utilizar un script propio para limpiar el texto de los PDFs y extraer metadatos clave (Nº de casación, lugar, materia).
- **Indexación Eficiente con LlamaIndex:** Emplear LlamaIndex para convertir los documentos procesados en embeddings y almacenarlos en una base de datos vectorial (ChromaDB).
- **Recuperación de Información Relevante:** Implementar un servicio que, ante una pregunta, utilice el retriever de LlamaIndex para encontrar los fragmentos de texto más pertinentes en el corpus.
- **Generación de Respuestas Aumentadas:** Usar los fragmentos recuperados como contexto para que un LLM (Google Gemini) genere una respuesta precisa y fundamentada.

2. Metodología: Enfoque Híbrido con LlamaIndex

La arquitectura del proyecto combina el control granular del procesamiento de datos con la simplicidad de LlamaIndex para las tareas de IA.

2.1. Proceso de Indexación (build-index.py)

La ingesta de datos es un proceso de dos etapas que maximiza la calidad de la información indexada.

1. **Pre-procesamiento Personalizado (‘pdf_processing.py‘):** Antes de interactuar con LlamaIndex, cada PDF del directorio de entrada es procesado por el script `procesar_pdf_completo`. Esta función es crucial, ya que:
   - Realiza una limpieza profunda del texto, eliminando las notificaciones SEO y normalizando espacios.
   - Utiliza expresiones regulares para extraer metadatos estructurados y de alto valor, como el número de casación, el lugar y la materia de la sentencia.
2. **Creación de Documentos y Vectorización:** El texto limpio y los metadatos extraídos se empaquetan en un objeto `Document` de LlamaIndex. Este enfoque permite que los metadatos ricos se asocien directamente con el contenido del texto.
3. **Indexación con LlamaIndex:** Se configura un modelo de embedding de Google (`GoogleGenAIEmbedding`) de forma global. Luego, `VectorStoreIndex.from_documents` orquesta la conversión de cada `Document` en un vector y su almacenamiento en una colección de **ChromaDB** a través de un `StorageContext`.

2.2. Motor de Consultas (Retrieval-First)

El proceso de respuesta también se separa en dos fases lógicas: recuperación y generación.

- **Recuperación de Nodos (vector-query.py):** En lugar de usar un ‘query-engine‘ de caja negra, se emplea un `retriever`. Este componente se encarga exclusivamente de una tarea: recibir la pregunta del usuario, vectorizarla y devolver los nodos (chunks de texto y sus metadatos) más relevantes desde ChromaDB.
- **Generación Aumentada (Lógica de la API):** Un segundo componente (en el servicio de la API) toma los nodos recuperados. Concatena su contenido de texto para formar un contexto rico y fáctico. Finalmente, construye un prompt que instruye al LLM para que responda a la pregunta del usuario basándose estrictamente en ese contexto. Este desacoplamiento ofrece un control total sobre cómo se presenta el contexto al modelo generativo.

3. Resultados y Conclusiones

El proyecto implementa con éxito un sistema RAG avanzado y eficiente. La decisión de utilizar un script de pre-procesamiento propio es la mayor fortaleza del sistema, ya

que asegura que la base de datos vectorial contenga información limpia y metadatos estructurados, mejorando drásticamente la precisión de la recuperación.

El uso de **LlamaIndex** para manejar la vectorización, el almacenamiento y la recuperación simplifica enormemente el desarrollo, permitiendo al programador centrarse en la calidad de los datos. Esta metodología híbrida representa un punto de equilibrio ideal entre el control y la automatización, resultando en una aplicación robusta, precisa y mantenible, perfectamente adaptada para un dominio tan específico como la jurisprudencia peruana.

4. Anexo: Fragmentos de Código Clave

```python
import chromadb
from llama_index.core import VectorStoreIndex, Document, Settings
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext
from llama_index.embeddings.google_genai import GoogleGenAIEmbedding
from app.utils.pdf_processing import procesar_pdf_completo
import os

def build_index(collection_name: str, persist_dir: str, input_dir: str):
    # 1. Configurar modelo de embedding de Google
    Settings.embed_model = GoogleGenAIEmbedding(model_name="models/embedding-001")

    # 2. Conectar a ChromaDB
    db = chromadb.PersistentClient(path=persist_dir)
    chroma_collection = db.get_or_create_collection(collection_name)

    # 3. Procesar PDFs con script personalizado y crear Documentos LlamaIndex
    documentos_llama = []
    for pdf_file in os.listdir(input_dir):
        if pdf_file.lower().endswith(".pdf"):
            pdf_path = os.path.join(input_dir, pdf_file)
            texto_limpio, metadatos = procesar_pdf_completo(pdf_path)
            if texto_limpio:
                documento = Document(text=texto_limpio, metadata=metadatos)
                documentos_llama.append(documento)

    # 4. Configurar almacenamiento y construir el indice
    vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
    storage_context = StorageContext.from_defaults(vector_store=vector_store)

    index = VectorStoreIndex.from_documents(
        documentos_llama,
        storage_context=storage_context,
        show_progress=True
    )
    print("  ndice   construido y guardado exitosamente!")
```

Listing 1: Construcción del índice con pre-procesamiento personalizado (‘build$_i$*ndex.py*‘)

```python
from llama_index.core import VectorStoreIndex, Settings
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.embeddings.google_genai import GoogleGenAIEmbedding
import chromadb

def search_documents(query: str, top_k: int):
    # 1. Configurar el modelo de embedding
    Settings.embed_model = GoogleGenAIEmbedding(model_name="models/
    embedding-001")

    # 2. Conectar a la base de datos y obtener el indice
    db = chromadb.PersistentClient(path="./db")
    chroma_collection = db.get_or_create_collection("lexiquipu")
    vector_store = ChromaVectorStore(chroma_collection=chroma_collection
    )
    index = VectorStoreIndex.from_vector_store(vector_store)

    # 3. Crear un retriever para buscar los nodos mas relevantes
    retriever = index.as_retriever(similarity_top_k=top_k)
    nodes = retriever.retrieve(query)

    # 4. Devolver los nodos para ser usados como contexto
    return nodes
```

Listing 2: Recuperación de documentos relevantes (‘vector$_q$*uery.py*‘)