Retrieval-Augmented Generation (RAG)

El motor detrás de la nueva generación de chatbots

PhD. Oscar Alberto Rodriguez Melendez

Universidad de la Sabana

2025-II

Contenido

Requerimientos del Entorno Productivo

Para generar los códigos que trabajaremos en el curso, es necesario crear un entorno que nos permita desarrollar en **Python**, mantener las mejores prácticas de la industria y controlar versiones.

Herramientas principales

- **Conda** o **Poetry**: gestión de entornos virtuales y dependencias.
- **Python**: lenguaje de programación base.
- **GitHub**: control de versiones y trabajo colaborativo.
- **PyCharm** o **VS Code**: entornos de desarrollo integrados (IDE).

Instalación de Miniconda

Miniconda nos permite crear entornos virtuales ligeros para ejecutar y aislar nuestros proyectos.

1. Ingresa a: `https://www.anaconda.com/docs/getting-started/miniconda/install`
2. Descarga la versión correspondiente a tu sistema operativo (Windows, macOS o Linux).
3. Instala siguiendo las instrucciones por defecto.
4. Verifica la instalación con: **conda –version**

Nota para Windows

Ejecuta los comandos de instalación desde la consola **Anaconda Prompt**, no desde CMD o PowerShell.

Creación de cuenta en GitHub

1. Ingresa a: `https://github.com`
2. Haz clic en **Sign up** y crea una cuenta gratuita.
3. Verifica tu correo electrónico.
4. Personaliza tu perfil con nombre, foto y descripción opcional.

Propósito

GitHub será nuestro repositorio remoto para almacenar y versionar el código del curso.

Creación del entorno con Conda

Una vez instalado Miniconda y creada la cuenta de GitHub:

1. Descarga el archivo `Chatbots.yml` desde:
   `https://github.com/orodriguezm1/Chatbots/tree/dev`
2. Guarda el archivo en tu carpeta de trabajo.
3. Abre una terminal (o Anaconda Prompt en Windows).
4. Ejecuta el comando: **conda env create -f Chatbots.yml**
5. Activa el entorno: **conda activate Chatbots**

Instalación de PyCharm

1. Ingresa a: `https://www.jetbrains.com/pycharm/download/`
2. Descarga la versión disponible.
3. Instala el IDE con las opciones por defecto.
4. Al abrir PyCharm por primera vez:
   - Selecciona el entorno de Conda (`Chatbots`).
   - Configura el tema visual (claro u oscuro).

Ventaja

PyCharm permite ejecutar notebooks, scripts y controlar GitHub desde la misma interfaz.

Conexión de PyCharm con GitHub

Objetivo: conectar el entorno local con tu cuenta de GitHub mediante un token seguro.

1. **Generar token en GitHub:**
   - Entra a tu cuenta y ve a **Settings → Developer settings → Personal access tokens**.
   - Selecciona **Tokens → Generate new token**.
   - Copia el token generado.
2. **Conectar en PyCharm:**
   - Ve a **File → Settings → Version Control → GitHub**.
   - Haz clic en "+" → **Log in via Token**.
   - Pega el token y presiona **OK**.

Clonar el repositorio desde tu cuenta (Fork)

1. Ingresa al repositorio original del curso:
   `https://github.com/orodriguezm1/Chatbots/tree/dev`
2. Haz clic en el botón **Fork** (parte superior derecha). Esto creará una copia del repositorio en tu cuenta personal de GitHub.
3. Abre tu cuenta y verifica que el nuevo repositorio aparezca como:
   `https://github.com/tu_usuario/Chatbots`
4. En PyCharm, ve al menú: **Git** → **Clone Repository**
5. Pega la URL de tu fork (tu copia personal), no la del repositorio original.
6. Haz clic en **Clone** y PyCharm descargará el proyecto localmente.

Importante

Trabaja siempre sobre tu fork personal. Los cambios que realices (commits, push, ramas) afectarán solo tu repositorio.

Actualizar el Fork desde GitHub

Puedes mantener tu fork sincronizado con el repositorio original directamente desde la página de GitHub:

1. Ingresa a tu cuenta de GitHub y abre tu repositorio personal `Chatbots`.
2. Si el repositorio original tiene cambios, verás un mensaje en la parte superior: **"This branch is behind orodriguezm1:dev by X commits"**.
3. Haz clic en el botón **"Sync fork"** o **"Fetch upstream"**.
4. Luego selecciona la opción **"Update branch"**.
5. GitHub traerá los nuevos commits del repositorio original a tu fork automáticamente.

Resultado

Tu fork se actualiza con el repositorio original sin escribir comandos. Al abrir PyCharm y ejecutar un **Git Pull**, los cambios aparecerán también en tu entorno local.

Consejo: Usa este método si solo necesitas sincronizar tu código. Si estás trabajando con ramas personalizadas o colaborando en grupo, es preferible hacerlo con los comandos de Git o desde PyCharm (o VS).

Crear cuenta en OpenRouter (openrouter.ai)

¿Qué es OpenRouter?

Un *gateway* que unifica acceso a múltiples modelos (Mistral, Llama, etc.) usando una API compatible con OpenAI.

1. Entra a `https://openrouter.ai` y pulsa **Sign in**.
2. Inicia sesión con **Google** o **GitHub**, etc.
3. Verifica tu correo si te lo solicita.

Generar tu Token de API

1. En **openrouter.ai** ve a **Dashboard** → **Keys** o **Settings** → **API Keys**.
2. Pulsa **Create new key**.
3. Asigna un nombre (p.ej., `Chatbots`) y crea el token.
4. Copia el token (formato típico `sk-or-v1-...`). **Guárdalo en lugar seguro**.

Importante

No compartas tu token en repositorios públicos. Trátalo como una contraseña.

Guardar el Token en `.env`

1. En la carpeta raíz de tu proyecto (donde está tu `.py`), existe un archivo `.env` con:

Contenido de `.env`

```
OPENAI_API_KEY = sk-or-v1-tu_token_de_openrouter
```

Reemplaza el valor que se encuentra en el archivo por el key que generaste.

Notas

- Asegúrate de que el token empiece por `sk-or-v1-`.
- Si usas **VS Code** o **PyCharm**, ejecuta desde la carpeta que contiene el `.env`.

Errores comunes (y cómo resolverlos)

- **Invalid authentication**: revisa que el token sea el de OpenRouter (`sk-or-v1-...`) y esté en `OPENAI_API_KEY`.
- **No se encuentra .env**: ejecuta el script desde la carpeta del proyecto o pasa ruta a `load_dotenv(ruta/.env")`.
- **Modelo no disponible**: verifica el `model_name` en **openrouter.ai/models**. Ej.: `mistralai/mistral-7b-instruct`.
- **403/429 (cuotas)**: revisa **Billing** en OpenRouter o baja `temperature` y frecuencia de llamadas.

Librerías Fundamentales

El sistema **RAG** combina componentes de búsqueda, embeddings y generación de lenguaje. A continuación se presentan las librerías más importantes en este proyecto:

- **LangChain**: Marco de orquestación que facilita la integración entre modelos, bases vectoriales y flujos de razonamiento.
- **LangChain-Community / LangChain-HuggingFace / LangChain-OpenAI**: Extensiones para usar modelos de embeddings, LLMs y loaders de documentos.
- **Sentence-Transformers**: Genera los embeddings vectoriales a partir de texto, usando modelos como `all-MiniLM-L6-v2`.
- **DeepLake**: Base de datos tensorial eficiente para almacenar y recuperar fragmentos embebidos.
- **Transformers (HuggingFace)**: Permite usar modelos de lenguaje y embeddings preentrenados.
- **Python-dotenv**: Carga variables de entorno (`.env`) para proteger claves de API.
- **OpenAI / LangChain-OpenAI**: Permite la comunicación con modelos de lenguaje grandes (LLMs) a través de la API de OpenRouter u OpenAI.

Otras librerías relevantes

- **PyPDFLoader (LangChain Community)**: Permite extraer texto de documentos PDF para crear corpus de conocimiento.
- **RecursiveCharacterTextSplitter**: Fragmenta textos grandes en secciones manejables para indexación.
- **NumPy / SciPy / Scikit-learn**: Proveen operaciones matemáticas, métricas de similitud y estructuras vectoriales.
- **Matplotlib / TQDM**: Visualización y seguimiento de progreso en tareas de entrenamiento o creación de embeddings.
- **Requests / HTTPX**: Permiten llamadas asíncronas a servicios externos como APIs.
- **Pandas**: Manipulación y análisis estructurado de datos cuando se requieren tablas o reportes.

Alternativas Profesionales

Aunque este curso usa las librerías anteriores, existen alternativas según el entorno de producción:

- **Embeddings**: OpenAI Embeddings, Cohere, o Vertex AI Embeddings (Google Cloud).
- **Vector Stores**: FAISS (Meta), Milvus, Chroma, Pinecone o Qdrant.
- **Orquestación**: LlamaIndex o Haystack (alternativas a LangChain).
- **LLMs**: GPT-4, Claude 3, Gemini, Mistral...

Nota

El stack mostrado está optimizado para un ejemplo básico y gratuito, y la estructura modular presentada facilita su migración a entornos productivos.

¿Qué es RAG?

RAG

La **información aumentada por recuperación** o RAG es el proceso mediante el cual la salida de un modelo de lenguaje de gran tamaño (LLM) es influenciada para hacer referencia a una base de conocimientos autorizada externa a los orígenes de entrenamiento del modelo previo a la generación de la respuesta.

Esquema ilustrativo

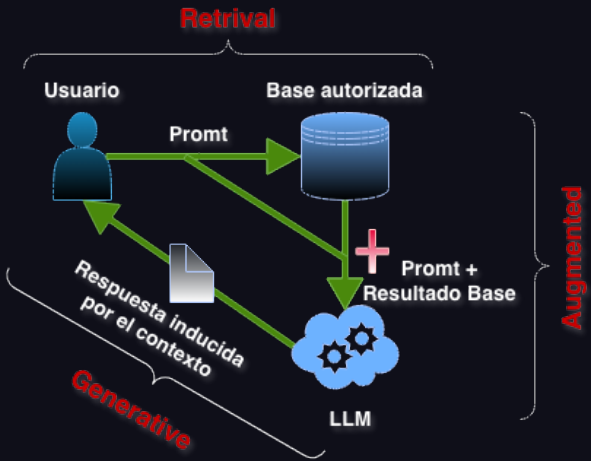


Figura: Flujo general de un sistema RAG: recuperación, aumento y generación.

¿Porqué es importante?

El enfoque RAG (Retrieval-Augmented Generation) representa una evolución clave en el uso de modelos de lenguaje grandes (LLMs), porque permite actualizar su conocimiento y adaptarlos a dominios específicos sin necesidad de reentrenamiento.
En un modelo tradicional, la única forma de incorporar nueva información o ajustar el comportamiento del modelo es mediante fine-tuning: volver a entrenar el modelo con datos adicionales.

¿Porqué es importante?

Tres limitaciones del fine-tuning

- **Costo y tiempo**: El reentrenamiento de un LLM requiere enormes recursos computacionales y tiempos prolongados.
- **Riesgo de degradar el modelo**: Cada ajuste puede alterar el conocimiento previo y generar errores o sesgos no deseados.
- **Desactualización rápida**: El modelo vuelve a quedar "congelado" en el conocimiento usado en el último fine-tuning.

Ventajas del enfoque RAG

Beneficios frente al fine-tuning

- **Actualización inmediata**: Basta con añadir nuevos documentos a la base externa.
- **Menor costo**: Evita el proceso de reentrenamiento completo.
- **Adaptabilidad**: Permite incorporar información específica sin modificar el modelo.
- **Trazabilidad**: Las fuentes utilizadas son identificables y auditables.

¿Qué es un LLM?

Definición

Un **Modelo de Lenguaje de Gran Tamaño (LLM, por sus siglas en inglés)** es una red neuronal entrenada con enormes cantidades de texto para predecir la siguiente palabra en una secuencia.

- Utiliza arquitecturas **Transformer**, que permiten procesar texto en paralelo y capturar dependencias contextuales.
- Aprende representaciones semánticas (embeddings) del lenguaje natural.
- Puede realizar tareas como: redacción, resumen, traducción, clasificación, "razonamiento" o generación de código.

Ejemplo

Entrada: "El café está caliente porque..." Salida posible: "acaba de prepararse."

¿Cómo funciona un chat con un LLM?

Flujo básico

En un chat, el usuario envía un **prompt** (mensaje de entrada) y el modelo genera una respuesta en lenguaje natural. El proceso incluye:

1. **Tokenización:** el texto se convierte en tokens (fragmentos numéricos).
2. **Inferencia:** el modelo predice el siguiente token según el contexto.
3. **Decodificación:** los tokens generados se transforman nuevamente en texto.

Ejemplo de flujo

Usuario: "Define qué es un embedding." $\rightarrow$ *LLM:* "Es una representación numérica que captura el significado de las palabras."

Relación entre Tokenización y Embeddings

De texto a vectores

Los modelos de lenguaje no procesan texto directamente. Primero, el texto se convierte en **tokens** (fragmentos de palabras), y luego cada token se transforma en un **vector numérico** llamado **embedding**.

Texto: "El gato duerme."

$$\text{Tokens} \rightarrow [\text{"El"}, \text{"gato"}, \text{"duerme"}, \text{"."}]$$

$$\text{Embeddings} \rightarrow \begin{bmatrix} 0{,}12 & -0{,}08 & 0{,}33 & \dots \\ 0{,}45 & -0{,}31 & 0{,}22 & \dots \\ 0{,}18 & 0{,}04 & 0{,}29 & \dots \\ -0{,}02 & 0{,}01 & 0{,}05 & \dots \end{bmatrix}$$

Cada fila representa un token convertido a un vector en $\mathbb{R}^n$.

El poder del contexto en los prompts

Contextualizar mejora la precisión

Los LLM no "saben" la respuesta; generan texto según el contexto que reciben. Por eso, un prompt bien diseñado influye directamente en la calidad de la salida.

Ejemplo

Prompt débil: "Explica embeddings." **Prompt contextualizado:** "En el contexto del procesamiento del lenguaje natural, explica qué son los embeddings y cómo representan las palabras en vectores."

Consejo

Cuanto más contexto y precisión tenga el prompt, más coherente, específica y útil será la respuesta.

Metaprompts y personalidad del modelo

¿Qué son los metaprompts?

Los **metaprompts** son instrucciones internas o de sistema que el usuario no ve, pero que orientan el comportamiento del modelo.

- Se usan para definir el **rol, tono y estilo** del modelo.
- Pueden inducir una "personalidad" o un enfoque particular en las respuestas.
- Ejemplo de metaprompt: *"Eres un asistente experto en IA que responde de manera técnica, clara y educativa."*

Comparación

Sin metaprompt: "El aprendizaje profundo es un conjunto de técnicas de IA."
Con metaprompt educativo: "El aprendizaje profundo es una rama de la inteligencia artificial que usa redes neuronales con múltiples capas para aprender representaciones complejas de los datos."

Memoria en Chats con LLM

¿Qué significa dar memoria a un LLM?

Los modelos no recuerdan por sí mismos conversaciones previas. Para mantener coherencia, debemos **proveerles contexto histórico** en cada nueva interacción.

Formas comunes de memoria

1. **Concatenar historial:** unir los últimos mensajes o un resumen al nuevo prompt.
2. **Memoria resumida:** guardar un resumen compacto del diálogo previo (reduce tokens).
3. **Memoria vectorial:** almacenar cada interacción como embedding y recuperar las más relevantes según similitud semántica.
4. **Memoria simbólica o estructurada:** usar bases de datos o grafos de conocimiento para representar hechos o relaciones.

Conclusión sobre LLMs y prompts

- Los LLM generan texto basado en patrones aprendidos, no en comprensión humana.
- La calidad del prompt define la calidad de la respuesta.
- Los metaprompts permiten adaptar el modelo a diferentes propósitos o estilos.
- Un RAG combina esta capacidad generativa con fuentes externas de conocimiento, evitando errores o desactualización.

¿Qué son los Embeddings?

Idea general

Un **embedding** es una representación vectorial de una palabra, frase o entidad en un espacio continuo $\mathbb{R}^n$.

- El objetivo es que palabras que aparecen en contextos similares tengan vectores similares.
- Cada palabra se representa como un vector numérico que captura su significado semántico.

Palabra	**Embedding (ejemplo)**
perro	[0.45, -0.31, 0.22, ...]
gato	[0.44, -0.28, 0.20, ...]
coche	[-0.12, 0.35, -0.10, ...]

"perro" y "gato" están cerca en el espacio vectorial.

Inicialización de los vectores

Representación inicial

Antes de entrenar, a cada palabra del vocabulario se le asigna un vector aleatorio.

$$E \in \mathbb{R}^{V \times n}$$

donde:

- V: tamaño del vocabulario (por ejemplo, 10 000).
- n: dimensión del embedding (por ejemplo, 300).
- Cada fila E_i es el vector de la palabra w_i.

Palabra	Vector inicial (aleatorio)
gato	[0.01, -0.02, 0.03]
perro	[-0.04, 0.01, 0.02]
casa	[0.02, 0.01, -0.01]

Entrenamiento de Embeddings

Idea general

Durante el entrenamiento, el modelo ajusta los vectores para que palabras que aparecen juntas tengan embeddings similares.

Ejemplo: "El gato duerme en la cama."

pares de entrenamiento: $(gato, duerme), (duerme, gato), (gato, cama), (cama, duerme)$

- El modelo aprende relaciones de contexto.
- Los vectores se actualizan iterativamente para reducir la pérdida.

Modelos clásicos: Word2Vec

Dos variantes principales

1. **Skip-gram:** predice palabras de contexto dado el centro.

$$P(w_o|w_c) = \frac{\exp(\mathbf{v}'_{w_o}{}^\top \mathbf{v}_{w_c})}{\sum_{j=1}^{V} \exp(\mathbf{v}'_j{}^\top \mathbf{v}_{w_c})}$$

2. **CBOW (Continuous Bag of Words):** predice la palabra central dado el contexto.

$$P(w_c|\text{contexto}) = \frac{\exp(\mathbf{v}'_{w_c}{}^\top \sum_{w \in contexto} \mathbf{v}_w)}{\sum_{j=1}^{V} \exp(\mathbf{v}'_j{}^\top \sum_{w \in contexto} \mathbf{v}_w)}$$

Función de pérdida (Skip-gram con Negative Sampling)

Objetivo del entrenamiento

Maximizar la similitud entre pares reales y minimizarla para pares falsos.

$$L = -\log \sigma(\mathbf{v'}_{w_o}{}^{\top} \mathbf{v}_{w_c}) - \sum_{i=1}^{k} \log \sigma(-\mathbf{v'}_{w_i^-}{}^{\top} \mathbf{v}_{w_c})$$

donde:

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- w_i^-: palabras negativas (aleatorias)
- k: número de muestras negativas

Actualización de los vectores

Gradiente y descenso

Los vectores se actualizan con descenso de gradiente:

$$\mathbf{v}_{new} = \mathbf{v}_{old} - \eta \frac{\partial L}{\partial \mathbf{v}}$$

Donde η es la tasa de aprendizaje.

Actualización de los vectores

Ejemplo práctico:

$$\mathbf{v}_{gato} = [0{,}2, 0{,}4]$$
$$\mathbf{v}'_{animal} = [0{,}3, 0{,}1]$$
$$\mathbf{v}'_{mesa} = [-0{,}2, 0{,}5]$$

$$\sigma({\mathbf{v}'_{animal}}^{\top} \mathbf{v}_{gato}) = \sigma(0{,}1) = 0{,}525$$
$$\sigma(-{\mathbf{v}'_{mesa}}^{\top} \mathbf{v}_{gato}) = \sigma(-0{,}16) = 0{,}46$$

Tras la actualización, $\mathbf{v}_{gato}$ se acerca a "animal" y se aleja de "mesa".

Resultado final del entrenamiento

Embeddings semánticos

Después del entrenamiento, los embeddings capturan relaciones semánticas:

$$\mathbf{v}_{rey} - \mathbf{v}_{hombre} + \mathbf{v}_{mujer} \approx \mathbf{v}_{reina}$$

- Las distancias reflejan similitudes semánticas.
- cos(“perro”, “gato”) $\approx$ alto.
- cos(“perro”, “mesa”) $\approx$ bajo.

¿Cómo comparamos la similitud entre palabras?

Objetivo

Una vez que cada palabra, frase o documento se representa como un vector, necesitamos una forma de medir qué tan **parecidos** son entre sí. Para esto, usamos métricas o medidas de similitud.

- En RAG, las medidas de similitud se utilizan para identificar los fragmentos más relevantes para una consulta.
- Permiten identificar qué embeddings son los más cercanos al embedding del **prompt**.
- Cuanto mayor sea la similitud, más relevante se considera el texto.

Tipos comunes de medidas de similitud

Principales medidas

- **Similitud del coseno**: compara el ángulo entre los vectores.
- **Distancia euclidiana**: mide la distancia "en línea recta".
- **Distancia de Manhattan**: suma las diferencias absolutas entre las coordenadas.

Medida	**Fórmula**
Coseno	$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$
Euclidiana	$d(A,B) = \sqrt{\sum_i (A_i - B_i)^2}$
Manhattan	$d(A,B) = \sum_i \|A_i - B_i\|$

Similitud del Coseno

Definición

Mide el **ángulo** entre dos vectores en el espacio. Si el ángulo es pequeño, los vectores apuntan en direcciones similares, lo que indica que los textos son semánticamente parecidos.

$$\text{sim}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

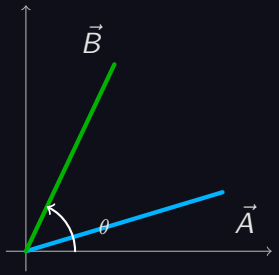
Interpretación:

- 1 $\rightarrow$ vectores idénticos.
- 0 $\rightarrow$ sin relación (perpendiculares).
- -1 $\rightarrow$ opuestos.

Visualización geométrica de la similitud

Ejemplo en 2D

Los embeddings se comportan como vectores en un espacio geométrico.



La similitud del coseno mide qué tan alineados están los vectores. Cuanto menor sea θ, mayor es la similitud.

Ejemplo numérico

Ejemplo

Sean:

$$A = [1, 2, 3], \quad B = [2, 3, 4]$$

$$\text{sim}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{1 * 2 + 2 * 3 + 3 * 4}{\sqrt{1^2 + 2^2 + 3^2}\sqrt{2^2 + 3^2 + 4^2}} = \frac{20}{\sqrt{14}\sqrt{29}} = 0{,}97$$

Interpretación: Los vectores A y B son muy similares porque el ángulo entre ellos es pequeño.

Comparación entre medidas

Ejemplo conceptual

Consideremos tres palabras con sus embeddings:

Palabra	Embedding (simplificado)
perro	[0.45, -0.31, 0.22]
gato	[0.44, -0.28, 0.20]
mesa	[-0.10, 0.33, -0.12]

- $\cos(\text{perro}, \text{gato}) \approx 0{,}99 \rightarrow$ alta similitud.
- $\cos(\text{perro}, \text{mesa}) \approx 0{,}05 \rightarrow$ casi nula.
- La distancia euclidiana entre "perro" y "gato" también es menor que la de "mesa".

Uso de las medidas en RAG

Cómo se usan

En un sistema **RAG**, la similitud del coseno se utiliza para identificar los fragmentos de texto más cercanos al embedding del **prompt**.

De escalares a tensores

Evolución de las estructuras numéricas

Un **tensor** es una generalización de los escalares, los vectores y las matrices. A medida que aumentan las dimensiones, la estructura crece en complejidad:

1. **Escalar:** un solo número. Ejemplo: $a = 5$
2. **Vector:** un arreglo de valores escalares. Ejemplo: $\vec{v} = [5, 2, 8]$
3. **Matriz:** un arreglo de vectores. Ejemplo:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

4. **Tensor:** un arreglo de matrices (o de arreglos de orden superior).

Idea clave

Los tensores permiten representar datos de múltiples dimensiones y relaciones complejas.

Ejemplo práctico: una imagen como tensor

Estructura tensorial de una imagen RGB

Una imagen digital puede representarse como un **tensor de orden 3**:

$$T \in \mathbb{R}^{\text{alto} \times \text{ancho} \times 3}$$

donde:

- Cada píxel tiene tres componentes: **Rojo (R)**, **Verde (G)** y **Azul (B)**.
- Cada canal es una matriz (intensidades por color).

Bases tensoriales para embeddings

Embeddings en estructuras tensoriales

Aunque los **embeddings** suelen representarse como vectores, también pueden almacenarse en una **base tensorial**, lo que permite una organización más general y eficiente.

- Una base tensorial almacena cada embedding como parte de un gran tensor multidimensional.
- Esto facilita búsquedas paralelas, filtrado por metadatos y relaciones de similitud complejas.
- La librería **DeepLake** de `activeloop.ai` nos permite trabajar con este tipo de estructura.

Ventaja

DeepLake gestiona embeddings como tensores, no solo como vectores, lo que ofrece mayor flexibilidad para integrar texto, imágenes, audio o video en una misma base.

Búsqueda en la base tensorial

Proceso general de consulta

Cuando realizamos una consulta (prompt o texto) en una base tensorial:

1. El texto se convierte en un embedding (vector o tensor) mediante un modelo de lenguaje.
2. Se calcula la **similitud** entre este embedding y los almacenados.
3. El sistema devuelve:
   - Las entradas cuya similitud supere un **umbral definido**, o
   - Las **más cercanas** según la medida de similitud.

Relación con las medidas de similitud

Estas búsquedas dependen directamente de métricas como la **similitud del coseno** o la **distancia euclidiana**, ya que determinan qué tan cerca están dos representaciones semánticas.

Conclusión: la búsqueda tensorial en RAG

- Un tensor es una extensión natural de los vectores y las matrices.
- Permite representar información multidimensional (texto, imagen, audio, etc.).
- DeepLake facilita el almacenamiento y la recuperación de embeddings en este formato.
- Las medidas de similitud son esenciales para identificar los fragmentos más relevantes en la base tensorial.

Idea final

La búsqueda tensorial es el corazón del RAG moderno: Conecta el conocimiento distribuido en embeddings con la generación contextual del modelo.

¿Qué son los prompts contextuales?

Definición

Un **prompt contextual** es una instrucción enriquecida que incluye información adicional —documentos, fragmentos, historial o datos estructurados— para que el modelo genere una respuesta más precisa y fundamentada.

- Permiten "inyectar" conocimiento recuperado por el sistema RAG sin necesidad de reentrenar el modelo.
- Transforman al LLM en un modelo temporalmente actualizado.
- Mejoran la coherencia, la trazabilidad y la relevancia de las respuestas.

Ejemplo

Prompt contextualizado: "Usando los siguientes fragmentos de conocimiento, explica qué es un embedding y su papel en RAG." *[Fragmentos recuperados]* **Pregunta:** ¿Cómo se relacionan los embeddings con la búsqueda tensorial?

Estructura general de un Prompt Contextual

Componentes principales

Un prompt contextual suele incluir tres secciones:

1. **Rol o sistema:** define el comportamiento del modelo.
2. **Contexto recuperado:** información obtenida desde la base tensorial (DeepLake, RAG).
3. **Instrucción o pregunta del usuario.**

Plantilla general

```
METAPROMT : Eres un asistente experto en IA que responde de forma técnica
y educativa.
CONTEXT : (Fragmentos relevantes recuperados del corpus)
PROMPT : Responde la siguiente pregunta usando exclusivamente el contexto
anterior:  "¿Qué diferencia hay entre embeddings y tensores?"
```

Tipos de contexto utilizados en los prompts

Contextos posibles

Los prompts contextuales pueden incorporar distintos tipos de información:

Tipo de contexto	Descripción	Ejemplo
Documental	Fragmentos recuperados del corpus	Artículos, manuales, PDFs
Conversacional	Historial o resumen del chat	"Como mencionaste antes..."
Estructurado	Datos tabulares o JSON	Fichas técnicas, logs, tablas
Ejemplos guiados	Casos de entrada/salida esperados	Few-shot prompting

Los mejores resultados se obtienen al combinar de manera coherente varios tipos de contexto.

Integración con el flujo RAG

Dónde se ubican los prompts contextuales

En un sistema **RAG**, los prompts contextuales constituyen el punto de unión entre la recuperación y la generación.

Buenas prácticas en prompts contextuales

Recomendaciones clave

- Coloca siempre el **contexto antes** de la pregunta.
- Indica explícitamente: "Usa solo la información del contexto anterior."
- Separa los fragmentos con delimitadores claros (como ---, ", etc.).
- Mantén el prompt dentro del límite de tokens del modelo.
- Define un **rol o tono** adecuado (técnico, docente, analítico...).
- Evalúa variaciones del prompt mediante A/B testing para optimizar las respuestas.

Un buen prompt contextual convierte al RAG en un sistema explicativo, trazable y especializado. Sin un diseño adecuado de prompt, la búsqueda tensorial pierde su poder interpretativo.

Integración con Modelos Generativos

Objetivo

Conectar los componentes del sistema RAG: la recuperación de conocimiento, la generación de embeddings y la respuesta final del modelo de lenguaje (LLM).

- La integración une tres capas principales:
  1. **Recuperación**: búsqueda de información relevante en la base tensorial.
  2. **Orquestación**: conexión lógica entre las etapas de RAG.
  3. **Generación**: uso del LLM para generar la respuesta contextualizada.
- Cada capa se implementa mediante librerías especializadas del entorno de Python.

Librerías principales en la integración

Bibliotecas del entorno

Estas librerías conforman el núcleo funcional que permite integrar búsqueda, embeddings y modelos generativos:

- **LangChain / LangChain-Core / LangChain-Community:** orquestación de cadenas de razonamiento y de la conexión entre módulos.
- **LangChain-OpenAI / LangChain-HuggingFace:** interfaz directa con modelos de lenguaje como GPT, Mistral o Llama.
- **Sentence-Transformers:** generación de embeddings semánticos a partir de texto.
- **DeepLake:** almacenamiento de embeddings en formato tensorial y búsqueda eficiente de similitudes.
- **Transformers (HuggingFace):** acceso a modelos preentrenados y procesamiento de texto avanzado.

Librerías auxiliares y de soporte

Complementos esenciales

Varias librerías del entorno refuerzan la robustez, asincronía y trazabilidad del sistema:

- **Python-dotenv:** gestión segura de variables de entorno y claves.
- **Pandas, NumPy, SciPy, Scikit-learn:** manejo de datos, métricas de similitud y operaciones vectoriales.
- **HTTPX / Requests:** conexión estable con APIs externas, como OpenRouter.
- **Matplotlib / TQDM:** visualización de resultados y monitoreo de procesos.
- **Tenacity / AIOHTTP / AnyIO:** control de asincronía y reintentos automáticos.

Flujo de integración del sistema RAG

Etapas del proceso

El flujo completo de integración sigue estos pasos:

1. El usuario envía una consulta en lenguaje natural.
2. Se genera un embedding del texto con **Sentence-Transformers**.
3. El embedding se busca en la base tensorial (**DeepLake**).
4. Los resultados más relevantes se integran en un **prompt contextual**.
5. El prompt se envía al modelo generativo vía **LangChain-OpenAI** u **HuggingFace**.
6. El LLM genera una respuesta coherente y fundamentada en el contexto recuperado.

Resultado

El modelo produce respuestas informadas y actualizadas, sin necesidad de reentrenamiento.

Arquitectura de un Chatbot RAG

Componentes principales

El chatbot RAG se construye como un flujo modular donde cada parte cumple un rol específico:

1. **Interfaz de usuario:** entrada y salida de mensajes (por terminal o web).
2. **Controlador:** gestiona los prompts y la interacción en general.
3. **Motor RAG:** combina embeddings, búsqueda tensorial y un modelo generativo.
4. **Base tensorial:** almacena los embeddings y los metadatos de referencia.

Cada módulo se comunica mediante las librerías del entorno de desarrollo.

Librerías empleadas en el chatbot RAG

Principales herramientas

Estas librerías permiten construir el flujo conversacional y mantener coherencia entre consultas:

- **LangChain / LangChain-Community:** administración de flujos conversacionales y recuperación contextual.
- **DeepLake:** gestión de la base tensorial y búsqueda de embeddings.
- **Sentence-Transformers:** creación de representaciones vectoriales de las consultas del usuario.
- **LangChain-Memory:** mantenimiento del contexto histórico y memoria de conversación.
- **Python-dotenv:** manejo de credenciales y configuración del entorno.

Flujo lógico de un Chatbot RAG

Secuencia operativa

El funcionamiento general del chatbot sigue un flujo claro y repetitivo:

1. El usuario formula una pregunta.
2. Se convierte la pregunta en embedding.
3. El sistema consulta la base tensorial para buscar fragmentos relevantes.
4. Se construye un prompt contextual combinando la información recuperada.
5. El prompt se envía al modelo generativo.
6. El modelo produce una respuesta contextualizada.
7. La conversación se actualiza y se conserva la memoria del diálogo.

Resultado

El chatbot mantiene coherencia, precisión y adaptación temática en cada interacción.

Síntesis de flujo completo (RAG paso a paso) [1/2]

Checklist end-to-end

1. **Preparar el entorno y las credenciales** (*python-dotenv*, tokens de OpenRouter/OpenAI).
2. **Definir alcance y corpus** (fuentes, dominios, PDFs y web) e **ingesta de documentos** (p.ej., *pypdf*).
3. **Normalizar y segmentar** el texto en **chunks** con solapamiento (tamaño/stride) usando *langchain-text-splitters*.
4. **Seleccionar modelo de embeddings** acorde a calidad/latencia (p. ej., *sentence-transformers* o *langchain-openai*).
5. **Generar embeddings** y **metadatos** (fuente, título, URL, etiquetas).
6. **Crear la base tensorial** y **persistir** embeddings + metadatos en **DeepLake** (búsqueda eficiente y almacenamiento tensorial).
7. **Elegir la medida de similitud** (recomendado *coseno*; alternativas: euclidiana, producto punto) y **estrategia de recuperación** (k, umbral, filtros).

El proceso RAG comienza desde la preparación del entorno hasta la creación de la base tensorial y la configuración de las métricas de similitud para la recuperación de contexto.

Síntesis de flujo completo (RAG paso a paso) [2/2]

Checklist end-to-end (continuación)

8. **Configurar el recuperador** (top-k, umbrales, re-ranking si aplica) contra la base **DeepLake**.
9. **Diseñar metaprompts y prompts contextuales** (rol, instrucciones, delimitadores; "usa solo el contexto").
10. **Integrar el LLM generativo** vía *langchain-openai* o *langchain-huggingface* (parámetros: temperatura, tokens).
11. **Conectar el chat con el recuperador (RAG)** y añadir **memoria conversacional** (*LangChain Memory*) para mantener el contexto.
12. **Observabilidad y evaluación**: logging/monitoreo (*LangSmith*, *TQDM*), métricas de calidad (precisión, cobertura, groundedness) y **ajuste iterativo** de k, chunking y prompts.
13. **Despliegue y robustez** (opcional): interfaz (FastAPI/Gradio/Streamlit), asincronía & red (*HTTPX/AIOHTTP*), reintentos (*Tenacity*), escalado y gobernanza.

Resultado: un chat que recupera el contexto a partir de la base tensorial (**DeepLake**) y genera respuestas fundamentadas con un LLM, sin reentrenamiento.

Librerías complementarias y soporte

- **HTTPX / AIOHTTP:** permiten comunicación eficiente con APIs externas.
- **Jinja2:** generación dinámica de plantillas de texto o prompts.
- **TQDM / Logging:** seguimiento del proceso y depuración de eventos.
- **Pathos / Multiprocess:** paralelización en la creación de embeddings masivos.
- **FastAPI / Streamlit / Gradio (opcional):** implementación de interfaces de chat interactivas.

Ventaja

El enfoque modular del chatbot RAG facilita su ampliación, personalización y despliegue en distintos entornos.

Síntesis general del aprendizaje

- Se exploraron los fundamentos teóricos: embeddings, tensores y medidas de similitud.
- Se comprendió la estructura de la búsqueda tensorial con DeepLake.
- Se integraron las librerías clave del entorno para construir un sistema RAG completo.
- Se mostró cómo un chatbot puede aprovechar esta arquitectura para ofrecer respuestas actualizadas y fundamentadas.

Ventajas del enfoque RAG

Principales beneficios

- **Actualización continua:** el conocimiento se amplía sin reentrenar el modelo.
- **Adaptabilidad:** ajustable a dominios o repositorios específicos.
- **Trazabilidad:** permite identificar las fuentes usadas en la respuesta.
- **Eficiencia:** menor costo computacional frente al fine-tuning.
- **Escalabilidad:** compatible con múltiples tipos de datos y entornos.

Perspectivas futuras del enfoque RAG

- Integración multimodal: texto, imagen, audio y video.
- Combinación con agentes autónomos y razonamiento simbólico.
- Evaluación automatizada de precisión contextual (Ragas, RAGAScore).
- Bases tensoriales distribuidas y sincronizadas en la nube.
- Incorporación de herramientas externas mediante frameworks como LangGraph o CrewAI.

Conclusión final

El enfoque RAG representa la base de la nueva generación de sistemas conversacionales, combinando recuperación de conocimiento, razonamiento y generación contextual.

¡Gracias!