UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO
DE INFORMÁTICA

Módulo 6
IA Generativa

Diploma en Inteligencia Artificial

informatica.usm.cl
@informaticausm

UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

DEPARTAMENTO
DE INFORMÁTICA

Capítulo 6: Aplicaciones LLM

01

Paisaje de Aplicaciones Generativas

Exploramos las principales categorías de aplicaciones LLM en la industria actual

02

Frameworks de Desarrollo: LangChain

Introducción práctica al framework que orquesta aplicaciones inteligentes

03

Diseño de Chatbots y Memoria

Arquitecturas conversacionales con persistencia de contexto

04

Deep Dive: RAG con Llama 3.1

Implementación paso a paso con código real y optimizaciones

05

Agentes y el Futuro

Sistemas autónomos que razonan y actúan de forma independiente

De la Teoría a la Práctica

Lo que ya dominamos

- Arquitectura de Transformers
- Pre-entrenamiento de modelos
- Técnicas de Fine-tuning
- Conceptos fundamentales de LLMs

El salto a la implementación

- **Inferencia eficiente** en producción
- **Orquestación** de componentes
- Integración con datos externos
- Construcción de productos útiles

La pregunta central de hoy: ¿Cómo convertimos un modelo base en una aplicación de software lista para resolver problemas reales? Este módulo cierra la brecha entre teoría académica y deployment profesional.

Categorías Principales de Aplicaciones

Generación

Creación de contenido original: campañas de marketing personalizadas, código fuente, arte digital, copywriting creativo

Resumen

Condensación inteligente: análisis de documentos legales extensos, transcripciones de reuniones, agregación de noticias

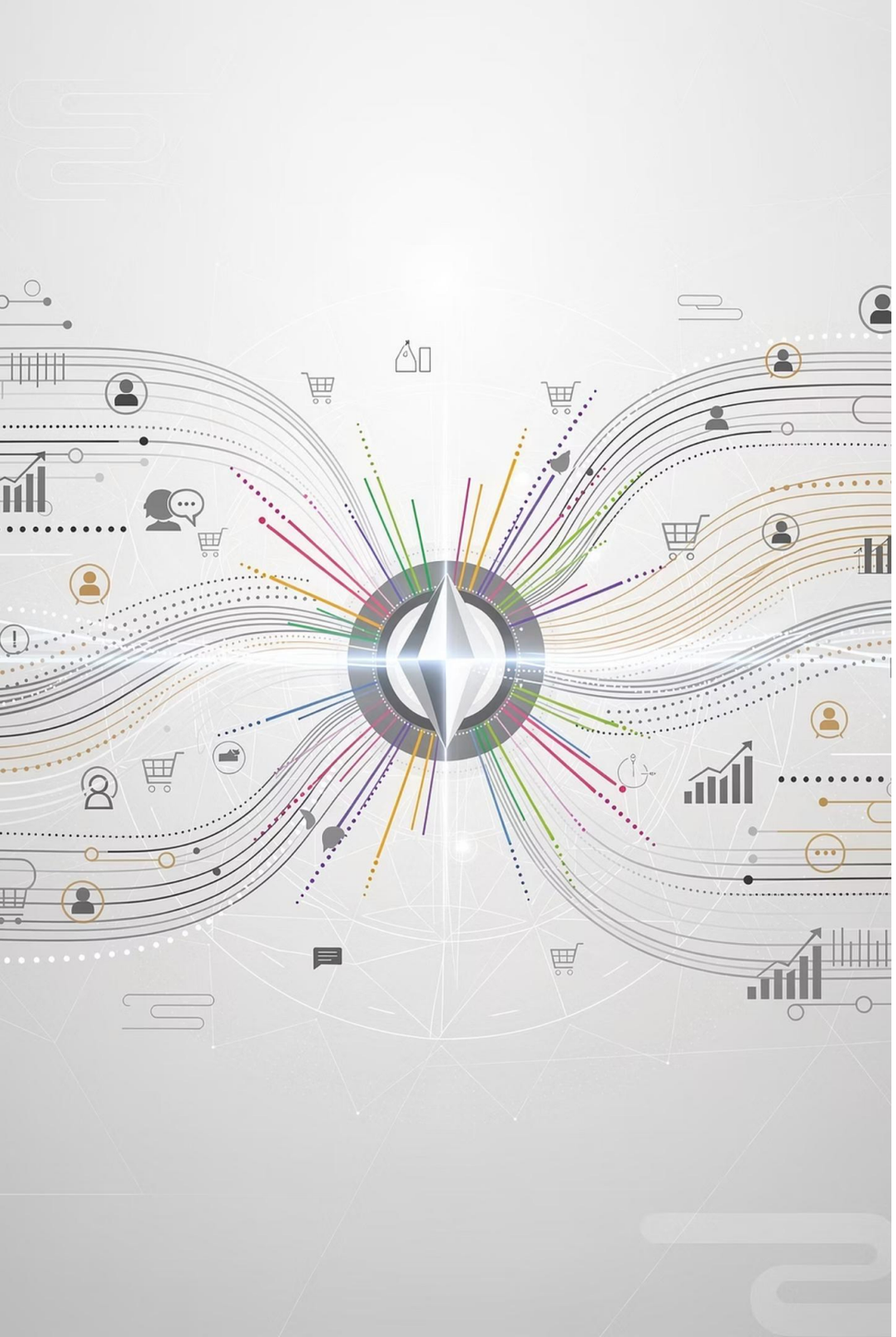
Transformación

Modificación de contenido existente: traducción multilingüe, cambio de tono y estilo, corrección gramatical avanzada

Extracción

Estructuración de datos: conversión de texto libre a JSON, extracción de entidades nombradas, parsing de facturas

Cada categoría presenta desafíos técnicos únicos y requiere estrategias de prompt engineering específicas para optimizar resultados.



Caso de Uso 1: Generación de Contenido Avanzada

Más allá del simple "escribe un email"

Hyper-personalización a escala

Los sistemas modernos pueden generar miles de variaciones de contenido adaptadas a perfiles individuales. Un anunciante puede crear 10,000 versiones únicas de un mensaje publicitario, cada una optimizada para demografía, historial de compras y comportamiento de navegación específicos.

Co-pilot de escritura inteligente

Autocompletado semántico que va más allá de palabras individuales. El sistema comprende el contexto completo del documento, sugiriendo párrafos enteros coherentes con el tono, estilo y estructura establecidos.

Aplicaciones empresariales

- Generación de reportes técnicos
- Creación de contenido SEO optimizado
- Redacción de propuestas comerciales
- Desarrollo de guiones para video
- Personalización de newsletters

Caso de Uso 2: Traducción Automática Contextual

Evolución más allá de Google Translate tradicional

Los LLMs modernos transforman la traducción automática al comprender tres dimensiones críticas que los sistemas estadísticos tradicionales no capturan:

Matices culturales y expresiones idiomáticas

El modelo reconoce que "break a leg" no debe traducirse literalmente, sino adaptarse culturalmente como "mucha mierda" en español.

Jerga técnica específica del dominio

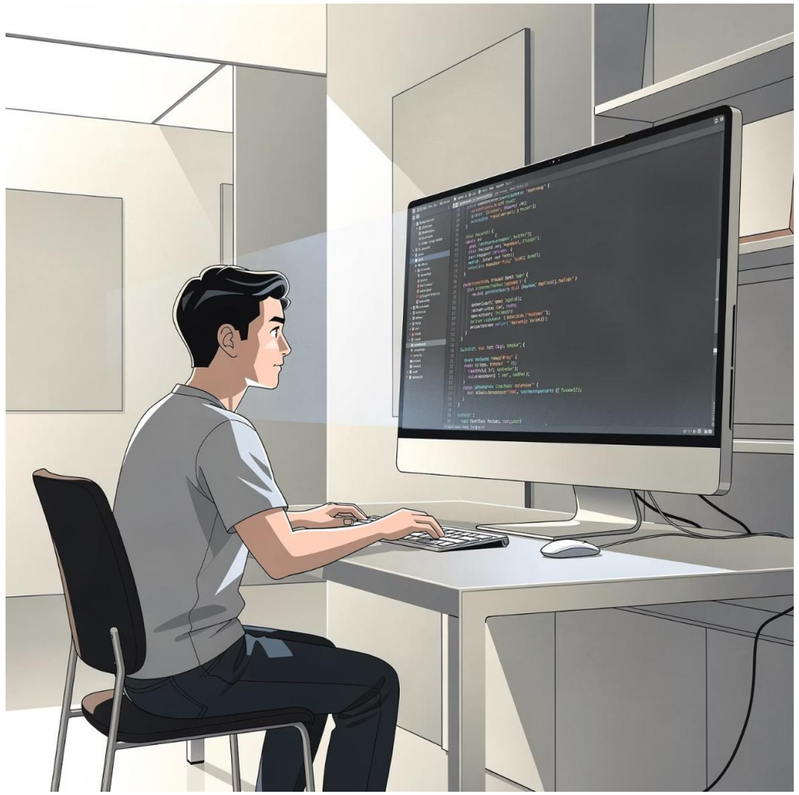
Un término médico como "loading dose" se traduce correctamente según el contexto farmacológico, no su significado literal.

Ambigüedad resuelta por contexto previo

La palabra "banco" se interpreta correctamente (institución financiera vs. asiento) basándose en el contexto de toda la conversación.

Ejemplo práctico: Traducir un manual médico requiere precisión clínica absoluta, mientras que una novela necesita preservar el tono emocional y la voz narrativa. El mismo modelo, con diferentes prompts, logra ambos objetivos.

Caso de Uso 3: Asistentes de Código



Revolucionando el desarrollo de software

Explicación de código legacy

Transformación de sistemas antiguos (COBOL, Java monolítico) a arquitecturas modernas (Python, microservicios) con documentación automática del proceso de migración.

Generación automática de Unit Tests

Análisis del código fuente para crear tests exhaustivos que cubren casos edge, validaciones de entrada y escenarios de error.

Documentación técnica inteligente

Generación de comentarios inline, docstrings, y README files que explican no solo qué hace el código, sino por qué se tomaron ciertas decisiones arquitectónicas.

GitHub Copilot

Autocompletado contextual durante el desarrollo

Cursor AI

Editor completo con IA integrada

CodeLlama

Modelo especializado en múltiples lenguajes

El Desafío de las Aplicaciones LLM

Alucinaciones

El modelo genera información falsa pero convincente. ¿Cómo validamos la veracidad de cada respuesta en un sistema de producción?

Contexto Limitado

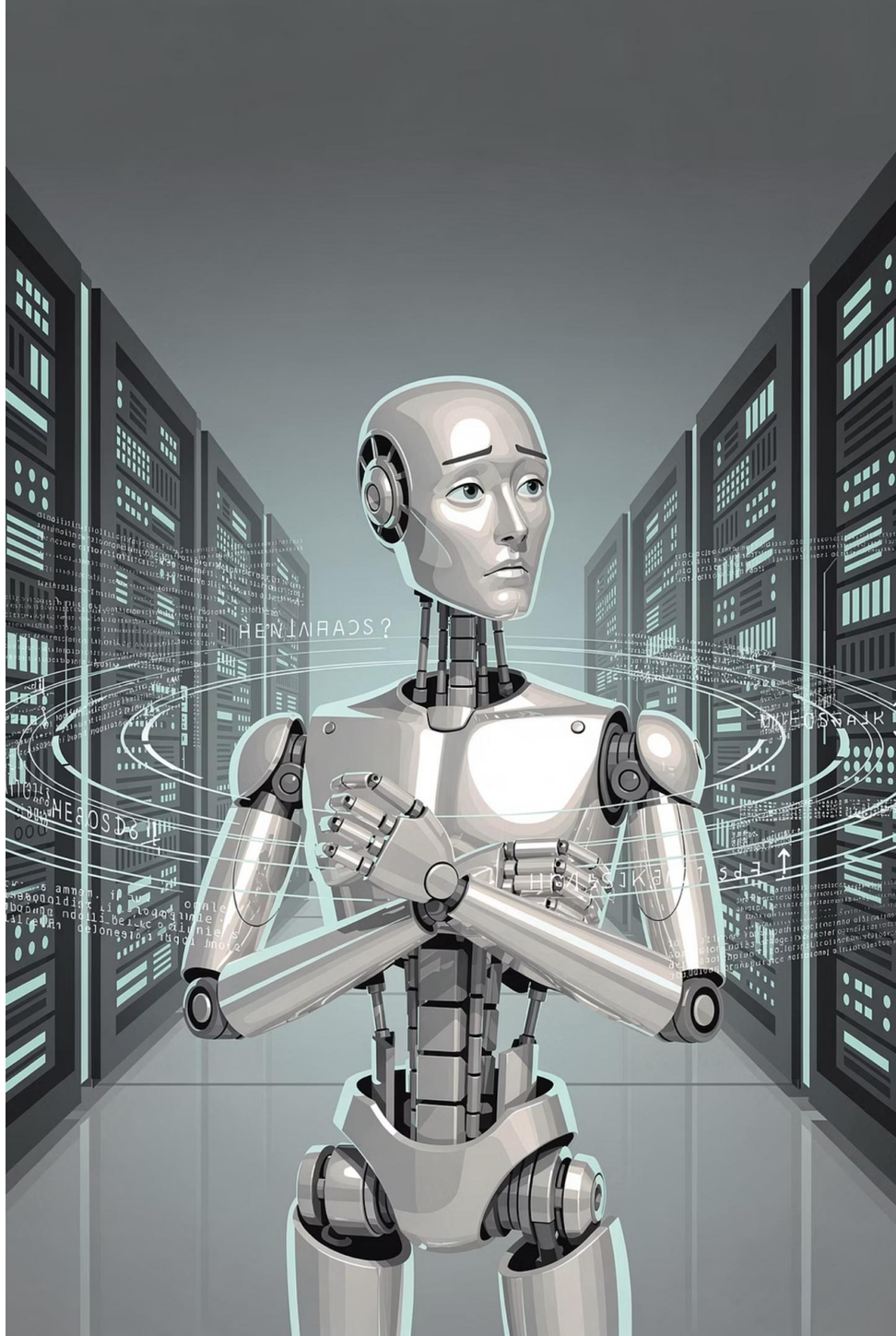
Ventanas de 4k-128k tokens. ¿Qué hacemos cuando el manual del usuario tiene 500 páginas y 2 millones de tokens?

Desactualización

Los modelos congelan su conocimiento en la fecha de entrenamiento. No saben nada sobre eventos actuales o datos internos de tu empresa.

La solución: Arquitecturas compuestas

No podemos depender del modelo aislado. Necesitamos sistemas que combinen el LLM con fuentes de datos externas actualizadas, validación de hechos, y recuperación de información contextual. RAG (Retrieval Augmented Generation) emerge como la arquitectura estándar para aplicaciones confiables.



La Nueva Stack Tecnológica (LLM Ops)

1

Capa de Modelo

Opciones: Llama 3.1 (open source, local), GPT-4 (API comercial, potente), Claude (antropic, seguro)

Criterios de selección: costo, latencia, capacidad, privacidad

2

Capa de Orquestación

Frameworks: LangChain (ecosistema completo), LlamaIndex (foco en RAG), Semantic Kernel (Microsoft)

Gestionan prompts, cadenas, memoria y agentes

3

Capa de Datos Vectoriales

Bases de datos: Chroma (embebida), Pinecone (cloud), Milvus (escalable), Weaviate (GraphQL)

Almacenan y buscan embeddings de manera eficiente

4

Capa de Infraestructura

Hosting: Google Colab (prototipado), AWS Bedrock (empresarial), HuggingFace Inference (community), Modal (serverless)

Proveen cómputo GPU y escalabilidad

Introducción al Entorno de Laboratorio

Plataforma: Google Colab

Utilizaremos Colab como nuestro entorno de desarrollo por su accesibilidad y recursos GPU gratuitos. Permite experimentación rápida sin configuración local compleja.

Hardware disponible

- **T4 GPU (Gratuita):** 16GB VRAM, suficiente para modelos 7B-8B cuantizados
- **A100 GPU (Colab Pro):** 40GB VRAM, permite modelos más grandes y batch processing
- **TPU v2:** Alternativa especializada para ciertos workloads

Stack de librerías esenciales

```
pip install langchain
pip install langchain-community
pip install transformers
pip install accelerate
pip install bitsandbytes
pip install sentence-transformers
pip install chromadb
pip install pypdf
```

Cada librería cumple un rol específico: **LangChain** orquesta, **Transformers** ejecuta modelos, **Accelerate** optimiza GPU, **BitsAndBytes** cuantiza, **ChromaDB** vectoriza.

Consejo práctico: Guarda tus notebooks en Google Drive y monta el drive en Colab para persistencia. La sesión gratuita se desconecta después de inactividad.

¿Qué es LangChain?

Framework para aplicaciones impulsadas por modelos de lenguaje

LangChain nace de una observación fundamental: los LLMs son increíblemente potentes, pero completamente aislados. No pueden acceder a internet, consultar bases de datos, ejecutar código, o recordar conversaciones pasadas. Son cerebros sin cuerpo ni memoria.

El problema del aislamiento

Un modelo pre-entrenado solo sabe lo que vio durante el entrenamiento. No puede responder "¿cuál es el clima hoy?" o "¿qué proyectos están activos en mi empresa?" porque esa información no existe en sus pesos neuronales.

La solución de LangChain

Conecta el LLM con el mundo exterior mediante dos pilares: **(1) Fuentes de datos** externas actualizadas, y **(2) Capacidad de interacción** con herramientas y APIs. El modelo se convierte en el "cerebro" de un sistema más amplio.

Componentes Core de LangChain

Models

Interface estandarizada para interactuar con cualquier LLM (GPT, Llama, Claude, Cohere) usando la misma API. Abstrae las diferencias entre proveedores.

Prompts

Sistema de gestión y optimización de instrucciones. Templates reutilizables, variables dinámicas, y ejemplo few-shot integrados. Versionado de prompts.

Chains

Secuencias de operaciones donde la salida de un paso se convierte en entrada del siguiente. Permite workflows complejos como "resumir → traducir → enviar email".

Indexes

Estructuras optimizadas para almacenar y recuperar documentos. Soporta chunking inteligente, embeddings vectoriales, y búsqueda semántica eficiente.

Memory

Mecanismos de persistencia del estado conversacional entre múltiples llamadas. Desde buffers simples hasta resúmenes comprimidos de conversaciones largas.

Ejemplo Python: Invocación Básica

Tu primer LLM con LangChain

El código más simple para ejecutar Llama 3.1 localmente en Colab con optimizaciones de memoria:

```
from langchain_community.llms import HuggingFacePipeline

# Configuración minimalista pero funcional
llm = HuggingFacePipeline.from_model_id(
    model_id="meta-llama/Meta-Llama-3.1-8B-Instruct",
    task="text-generation",
    pipeline_kwargs={
        "max_new_tokens": 100,
        "temperature": 0.7,
        "top_p": 0.9
    }
)

# Invocar el modelo
respuesta = llm.invoke("¿Por qué el cielo es azul?")
print(respuesta)
```

Parámetros clave explicados

- **max_new_tokens:** límite de tokens generados (evita bucles infinitos)
- **temperature:** aleatoriedad (0=determinista, 1=creativo)
- **top_p:** sampling probabilístico (nucleus sampling)

Lo que sucede internamente

- Descarga del modelo desde HuggingFace Hub
- Carga en memoria GPU (si disponible)
- Tokenización del input
- Inferencia autoresiva
- Decodificación de tokens a texto

Prompt Templates

Separación de lógica y contenido

Hardcodear strings directamente en el código es frágil y difícil de mantener. Los templates permiten reutilización, versionado, y colaboración entre equipos técnicos y de contenido.

Enfoque incorrecto ❌

```
respuesta = llm.invoke(
  "Traduce 'Hola mundo' " +
  "del Español al Francés"
)
```

Mezclamos datos (textos, idiomas) con estructura. Cada cambio requiere modificar código.

Enfoque correcto con templates ✅

```
from langchain_core.prompts import PromptTemplate

template = """Traduce el siguiente texto
de {idioma_origen} a {idioma_destino}:

{texto}"""

prompt = PromptTemplate.from_template(template)

# Uso con datos dinámicos
mensaje = prompt.format(
    idioma_origen="Español",
    idioma_destino="Francés",
    texto="Hola mundo"
)
```

Ventaja clave: Los templates se pueden almacenar en archivos JSON, versionar en Git, y modificar sin tocar código Python. Los equipos de contenido pueden iterar independientemente.

Chains (Cadenas)

El pegamento de la aplicación

Las cadenas conectan componentes para crear workflows complejos. LangChain introduce LCEL (LangChain Expression Language), una sintaxis declarativa moderna que reemplaza las viejas clases verbose.

La magia del operador pipe (|)

```
from langchain_core.prompts import PromptTemplate

template = "Traduce de {idioma_origen} a {idioma_destino}: {texto}"
prompt = PromptTemplate.from_template(template)

# Composición elegante con LCEL
chain = prompt | llm

# Ejecución
response = chain.invoke({
    "idioma_origen": "Español",
    "idioma_destino": "Francés",
    "texto": "Hola mundo"
})
```

1

Input Dict

Variables del template

2

PromptTemplate

Formatea el mensaje

3

LLM

Genera respuesta

4

Output String

Traducción final

Sequential Chains

Orquestando flujos multi-paso

Los problemas complejos requieren múltiples llamadas al LLM, donde cada paso depende del anterior. Sequential Chains automatizan este flujo de datos entre operaciones.

Paso 1: Generación

LLM genera un nombre creativo de empresa basado en la industria: "TechNova Solutions"

Paso 2: Copywriting

LLM escribe un eslogan para "TechNova Solutions": "Innovación que transforma el futuro"

Paso 3: Traducción

LLM traduce el eslogan al inglés: "Innovation that transforms the future"

Paso 4: Validación

LLM verifica que la traducción preserva el mensaje de marca y tono profesional

Patrón de diseño: Cada paso produce un output que se inyecta automáticamente como variable en el template del siguiente paso. El desarrollador solo define la estructura, LangChain gestiona el flujo de datos.

Model I/O: Parsers de Salida

De texto libre a datos estructurados

Los LLMs generan texto natural, pero las aplicaciones necesitan objetos estructurados (JSON, Python dicts, Pydantic models). Los Output Parsers resuelven esta transformación automáticamente.

El problema del formato

Pedimos al LLM extraer información de una factura. Responde: *"El cliente es Juan Pérez, la fecha es 15/03/2024 y el monto total es $1,250.50"*. ¿Cómo parseamos esto programáticamente?

La solución con JsonOutputParser

```
from langchain.output_parsers import JsonOutputParser

parser = JsonOutputParser()

# El parser modifica el prompt automáticamente
# para exigir formato JSON válido

chain = prompt | llm | parser

resultado = chain.invoke({"texto": factura})
# Output:
# {
#   "cliente": "Juan Pérez",
#   "fecha": "2024-03-15",
#   "monto": 1250.50
# }
```

Caso de uso real: Extracción de entidades

Una empresa procesa miles de contratos PDF. Necesitan extraer: nombre del contrato, partes involucradas, fechas clave, cláusulas de terminación. JsonOutputParser + template específico + validación Pydantic = pipeline robusto de extracción.

Instalación de Dependencias (Llama 3.1)

Optimizaciones para ejecutar modelos grandes localmente

Llama 3.1 8B tiene aproximadamente 16GB en precisión completa (float32). La T4 GPU de Colab gratuito solo tiene 16GB VRAM total. Sin cuantización, el modelo no cabe en memoria.

bitsandbytes

Librería de cuantización que reduce modelos de 16-bit a 4-bit con pérdida mínima de calidad. Compresión 4x permite cargar modelos más grandes.

```
pip install bitsandbytes
```

accelerate

Gestión inteligente de memoria GPU. Divide modelos entre GPU/CPU/Disco automáticamente. Maneja offloading transparente.

```
pip install accelerate
```

transformers

Librería de HuggingFace para cargar y ejecutar modelos. Abstrae tokenización, generación, y optimizaciones específicas por arquitectura.

```
pip install transformers>=4.40.0
```

Verificación de instalación

```
import torch
print(f"GPU disponible: {torch.cuda.is_available()}")
print(f"Nombre GPU: {torch.cuda.get_device_name(0)}")
print(f"VRAM total: {torch.cuda.get_device_properties(0).total_memory / 1e9:.2f} GB")
```

Configuración de Cuantización (Código)

Comprimiendo el modelo sin perder capacidad

BitsAndBytesConfig controla cómo se comprimen los pesos del modelo. La cuantización a 4-bit es la técnica que hace posible ejecutar Llama 3.1 en hardware consumer.

```
from transformers import BitsAndBytesConfig

# Configuración óptima para T4 GPU
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,              # Activar cuantización 4-bit
    bnb_4bit_quant_type="nf4",      # Normal Float 4 (mejor que int4)
    bnb_4bit_compute_dtype=torch.bfloat16,  # Cómputo en bfloat16
    bnb_4bit_use_double_quant=True  # Doble cuantización (ahorra más memoria)
)
```

Parámetros explicados

- **nf4:** formato normalizado que preserva mejor la distribución de pesos comparado con int4 uniforme
- **bfloat16:** formato de 16 bits optimizado para ML (rango mayor que float16)
- **double_quant:** cuantiza también los valores de cuantización (meta-compresión)

Impacto en recursos

- **Memoria:** 16GB → 4GB (reducción 75%)
- **Velocidad:** ~15% más lento que 16-bit
- **Calidad:** Perplexidad aumenta < 3%
- **Trade-off:** Memoria vs. Latencia (favorable para GPUs limitadas)

Nota técnica: La cuantización es "free lunch" en el sentido de que la pérdida de calidad es minimal comparada con la ganancia en accesibilidad. Estudios muestran que 4-bit NF4 está a 2-3% de perplexidad vs. full precision.

El Pipeline de Hugging Face en LangChain

Envolviendo el modelo en la interfaz de LangChain

Transformers usa su propia API. LangChain necesita una interfaz unificada. HuggingFacePipeline hace de adaptador entre ambos mundos.

```
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from langchain_community.llms import HuggingFacePipeline

# 1. Cargar modelo y tokenizer con cuantización
model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Meta-Llama-3.1-8B-Instruct",
    quantization_config=bnb_config,
    device_map="auto"  # Distribución automática GPU/CPU
)

tokenizer = AutoTokenizer.from_pretrained(
    "meta-llama/Meta-Llama-3.1-8B-Instruct"
)

# 2. Crear pipeline de Transformers
hf_pipeline = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=512,
    temperature=0.1,  # Baja temperatura = respuestas más precisas
    do_sample=True
)

# 3. Envolver en LangChain
llm = HuggingFacePipeline(pipeline=hf_pipeline)
```

Temperatura: creatividad vs. precisión

- **0.1:** Respuestas deterministas, ideal para extracción de datos
- **0.7:** Balance, buen default para chat
- **1.2:** Alta creatividad, para brainstorming o escritura creativa

device_map="auto"

Accelerate analiza la memoria disponible y decide qué capas van a GPU, cuáles a CPU, y cuáles a disco. Es magia automática de optimización.

El Problema de los LLMs "Stateless"

Sin memoria = Sin contexto

Por diseño arquitectónico, los LLMs son funciones puras: dado un input, producen un output. No tienen concepto de "sesión" o "conversación". Cada llamada a la API es completamente independiente de la anterior.

Escenario sin memoria

```
Usuario: "Me llamo Carlos"
LLM: "Encantado de conocerte Carlos"

Usuario: "¿Cuál es mi nombre?"
LLM: "Lo siento, no tengo esa información"
```

El modelo literalmente "olvida" lo que pasó 10 segundos antes. Cada interacción es un lienzo en blanco.

Por qué esto es problemático

- Chatbots que no recuerdan preferencias del usuario
- Asistentes que pierden contexto de la tarea en curso
- Imposibilidad de seguir conversaciones multi-turno
- Usuario debe repetir información constantemente

La solución técnica

Necesitamos inyectar manualmente el historial de la conversación en cada nueva llamada. El modelo no tiene memoria interna, pero podemos simularla incluyendo mensajes anteriores en el prompt actual.

Tipos de Memoria en LangChain

1

ConversationBufferMemory

Estrategia: Almacena literalmente todos los mensajes de la conversación

Ventaja: Contexto completo, nunca pierde información

Desventaja: Crece linealmente con la conversación. Después de 50 mensajes, el prompt puede tener 10k tokens = caro y lento

2

ConversationBufferWindowMemory

Estrategia: Mantiene solo los últimos K intercambios (ej. últimos 5 mensajes)

Ventaja: Memoria constante O(K), costo predecible

Desventaja: Olvida contexto antiguo. No sabe lo que se dijo al inicio de la conversación

3

ConversationSummaryMemory

Estrategia: El LLM resume periódicamente la conversación antigua y guarda solo el resumen

Ventaja: Balance entre contexto y eficiencia. Retiene información clave pero comprimida

Desventaja: Requiere llamadas extra al LLM para generar resúmenes. Potencial pérdida de detalles

Regla práctica: BufferWindow para chatbots cortos (< 10 turnos), Summary para conversaciones largas empresariales, Buffer completo solo para debugging o análisis exhaustivo.

Estructura de un Chatbot

El loop conversacional completo

1. **Input Usuario**
   Mensaje actual en lenguaje natural
2. **Memoria**
   Recupera historial relevante de la conversación
3. **Prompt**
   Combina System + Historia + Input en un solo contexto
4. **LLM**
   Genera respuesta basada en todo el contexto
5. **Output**
   Respuesta se guarda en memoria y se muestra al usuario

Anatomía del prompt completo

```
SYSTEM: Eres un asistente experto en finanzas que habla con tono formal.

HISTORIAL:
Usuario: Quiero invertir $10,000
Asistente: ¿Cuál es tu tolerancia al riesgo?
Usuario: Moderada

MENSAJE ACTUAL:
Usuario: ¿Qué me recomiendas?

[LLM genera respuesta basada en todo lo anterior]
```

El LLM ve esto como un único string continuo. La "memoria" es simplemente incluir mensajes anteriores en el prompt.

Implementación de Memoria (Código)

De la teoría al código funcional

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

# 1. Inicializar memoria
memory = ConversationBufferMemory(
    return_messages=True  # Devuelve objetos Message en lugar de strings
)

# 2. Crear cadena conversacional
chatbot = ConversationChain(
    llm=llm,
    memory=memory,
    verbose=True  # Imprime el prompt completo para debugging
)

# 3. Primera interacción
respuesta1 = chatbot.predict(input="Hola, me llamo Pablo")
print(respuesta1)
# "Encantado de conocerte Pablo. ¿En qué puedo ayudarte hoy?"

# 4. Segunda interacción - aquí vemos la magia
respuesta2 = chatbot.predict(input="¿Cómo me llamo?")
print(respuesta2)
# "Te llamas Pablo, como me acabas de decir."
```

Qué sucede internamente

1. Memoria recupera historial: `["Usuario: Hola, me llamo Pablo", "Asistente: Encantado..."]`
2. ConversationChain construye prompt con historial + nuevo mensaje
3. LLM procesa TODO el contexto
4. Respuesta se agrega automáticamente a la memoria

verbose=True es tu amigo

Durante desarrollo, activarlo muestra exactamente qué prompt se envía al modelo. Invaluable para debugging cuando el bot "olvida" cosas o responde raro.

Chat Templates en Llama 3

Formatos especiales que el modelo espera

Llama 3.1 fue entrenado con una estructura conversacional específica usando tokens especiales. Respetar este formato es crítico para que el modelo funcione correctamente.

Estructura del chat template

```
<|begin_of_text|>
<|start_header_id|>system<|end_header_id|>

Eres un asistente útil y preciso.
<|eot_id|>

<|start_header_id|>user<|end_header_id|>

¿Cuál es la capital de Francia?
<|eot_id|>

<|start_header_id|>assistant<|end_header_id|>

La capital de Francia es París.
<|eot_id|>
```

Tokens especiales

- **<|begin_of_text|>:** Inicio absoluto
- **<|start_header_id|>:** Quien habla (system/user/assistant)
- **<|end_header_id|>:** Fin del rol
- **<|eot_id|>:** End of turn (fin del mensaje)

Buena noticia: El tokenizer de Llama 3 maneja esto automáticamente con `apply_chat_template()`. No necesitas escribir estos tokens manualmente, pero es importante entender por qué están ahí.

System Prompts

Definiendo la personalidad y comportamiento del asistente

El System Prompt es la instrucción inicial que condiciona TODAS las respuestas subsecuentes. Es como contratar a alguien y darle un manual de estilo que debe seguir siempre.

Asistente Financiero Formal

```
Eres un asesor financiero certificado con 15 años de experiencia. Hablas con tono profesional, siempre pides información adicional antes de hacer recomendaciones, y citas fuentes cuando mencionas datos numéricos. Nunca garantizas rendimientos.
```

Tutor de Programación Amigable

```
Eres un mentor de código paciente y entusiasta. Explicas conceptos con analogías cotidianas, celebras los logros del estudiante, y desglosas problemas complejos en pasos pequeños. Usas emojis ocasionalmente para mantener un tono amigable.
```

Asistente Legal Cauto

```
Eres un asistente legal que proporciona información general, NO asesoramiento legal específico. Siempre recomiendas consultar a un abogado licenciado para casos específicos. Usas lenguaje preciso y evitas ambigüedades.
```

Impacto observable

Mismo input: "¿Debería invertir en Bitcoin?" produce respuestas completamente diferentes según el System Prompt. El financiero analiza riesgo/retorno, el tutor lo explica como concepto educativo, el legal advierte sobre regulaciones.

Gestión de Contexto Largo

¿Qué pasa cuando la conversación crece sin límite?

Los modelos tienen ventanas de contexto finitas: Llama 3.1 (128k tokens), GPT-4 Turbo (128k), Claude 3 (200k). Una conversación de soporte técnico puede superar fácilmente estos límites.

1

Problema: Overflow de contexto

Después de 100 mensajes, el prompt completo tiene 150k tokens. El modelo rechaza la solicitud con error "context length exceeded".

2

Estrategia 1: Recorte por ventana deslizante

ConversationBufferWindowMemory mantiene solo los últimos N turnos. Simple pero efectivo para la mayoría de casos.

3

Estrategia 2: Resumen progresivo

Cada 20 mensajes, el LLM resume los primeros 15 y descarta los originales. El resumen se mantiene en el contexto.

4

Estrategia 3: RAG como memoria externa

Guardar todo el historial en una base de datos vectorial. Recuperar solo los mensajes semánticamente relevantes para el mensaje actual.

Patrón avanzado: Combinar ventana reciente (últimos 5 mensajes) + resumen antiguo + RAG para preguntas sobre contexto lejano. Esto da memoria de corto y largo plazo simultáneamente.

Asistentes Especializados

Personalización extrema para dominios específicos

Asistente de Código

- **System Prompt:** "Eres un senior engineer. Generas código limpio, idiomático, con error handling. Explicas trade-offs."
- **Temperatura:** 0.0 (determinismo absoluto)
- **Herramientas:** Python REPL, linter, unit test runner
- **Formato output:** Markdown con bloques de código
- **Validación:** Ejecución en sandbox antes de mostrar

Asistente Legal

- **System Prompt:** "Proporciona info general, no asesoramiento legal. Cita precedentes cuando aplique."
- **Temperatura:** 0.2 (bajo, prioriza precisión)
- **Herramientas:** Base de datos de leyes, buscador de jurisprudencia
- **Formato output:** Párrafos formales con citas entre paréntesis
- **Validación:** Disclaimer automático en cada respuesta

El dilema temperatura

Para código: temperatura cercana a 0 porque queremos soluciones correctas y reproducibles. Para escritura creativa: temperatura 0.7-1.0 porque buscamos variedad y originalidad. No existe un valor universal óptimo.

Persistencia de Sesiones

Memoria que sobrevive reinicios de servidor

ConversationBufferMemory vive solo en RAM. Si el servidor de Colab se reinicia, toda la conversación desaparece. Para aplicaciones serias, necesitamos persistencia durable.

Almacenamiento en Base de Datos

Guardar cada mensaje en PostgreSQL/MongoDB con timestamp y session_id. Permite auditoría completa y análisis posterior.

Identificadores de Sesión

Cada conversación tiene un UUID único. Permite múltiples sesiones concurrentes del mismo usuario (chat en móvil + web).

Recuperación de Historial

Al inicio de cada sesión, cargar los últimos N mensajes desde DB a memoria. Usuario retoma exactamente donde lo dejó.

Ejemplo de esquema SQL

```
CREATE TABLE chat_messages (
  id SERIAL PRIMARY KEY,
  session_id UUID NOT NULL,
  role VARCHAR(20) NOT NULL,  -- 'user' o 'assistant'
  content TEXT NOT NULL,
  timestamp TIMESTAMPTZ DEFAULT NOW(),
  INDEX idx_session (session_id, timestamp)
);
```

La query de recuperación: SELECT * FROM chat_messages WHERE session_id = ? ORDER BY timestamp DESC LIMIT 20

UX en Chatbots

La diferencia entre un demo técnico y un producto usable

Streaming de respuestas

En lugar de esperar 10 segundos en silencio, mostrar tokens mientras se generan. Reduce la percepción de latencia dramáticamente. Implementar con Server-Sent Events (SSE) o WebSockets.

Indicadores de estado

Mostrar "Pensando..." o "Buscando en documentos..." o "Ejecutando código...". Usuarios toleran esperas largas si entienden qué está pasando.

Manejo de errores graceful

Timeout después de 30 segundos con mensaje amable: "La respuesta está tomando más tiempo de lo esperado. ¿Quieres intentar reformular la pregunta?". Nunca mostrar stack traces.

Botón de regenerar respuesta

Si la primera respuesta no es útil, permitir regenerar con diferente seed/temperatura sin repetir la pregunta.

Dato de UX research: Streaming reduce la tasa de abandono en un 35% comparado con espera estática. Los usuarios perciben el sistema como "más inteligente" aunque la latencia total sea la misma.

¿Qué es RAG?

Retrieval Augmented Generation

El problema fundamental

Los LLMs tienen dos limitaciones críticas que los hacen inadecuados para aplicaciones empresariales serias:

1. **Alucinaciones:** Inventan información falsa con total confianza
2. **Conocimiento congelado:** No saben nada posterior a su fecha de entrenamiento ni sobre datos privados de tu empresa

Pregunta a GPT-4 sobre tu política interna de vacaciones → alucina una respuesta genérica incorrecta.

La solución RAG

En lugar de depender de la memoria del modelo, le damos un "libro abierto" durante el examen:

1. **Retrieval:** Buscar información relevante en tus documentos
2. **Augmented:** Pegar esa información en el prompt
3. **Generation:** El LLM responde basándose en esa información real

El modelo se convierte en un "lector inteligente" en lugar de un "experto que memoriza".

Analogía: Sin RAG = estudiante haciendo examen de memoria cerrado. Con RAG = estudiante con material de referencia permitido, puede buscar la respuesta exacta.

Arquitectura RAG Alto Nivel

1

Documentos

PDFs, TXT, Web, SQL, APIs

2

Splitter

Fragmentar en chunks de 500-1000 tokens

3

Embeddings

Convertir a vectores numéricos

4

Vector Store

Índice para búsqueda rápida

1

Query Usuario

"¿Cuál es la política de vacaciones?"

2

Retriever

Busca top 3 chunks relevantes

3

LLM

Genera respuesta usando chunks como contexto

4

Respuesta

"Según el manual: 20 días al año..."

Dos fases distintas

Fase offline (una vez): Procesar todos los documentos, generar embeddings, construir índice. Puede tomar horas para corpus grandes.

Fase online (cada query): Buscar chunks relevantes (milisegundos), construir prompt con contexto, generar respuesta (segundos).

Paso 1: Ingesta de Documentos

Cargando conocimiento desde múltiples fuentes

LangChain proporciona loaders especializados para cada tipo de documento. Cada uno maneja las peculiaridades del formato (metadata, estructura, encoding).

Cargar PDFs individuales

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("/content/manual.pdf")
docs = loader.load()

# Output: lista de Document objects
# Cada página = 1 Document
print(f"Cargadas {len(docs)} páginas")
print(docs[0].page_content)  # Texto de página 1
print(docs[0].metadata)      # {'source': '...', 'page': 0}
```

Cargar carpetas completas

```
from langchain_community.document_loaders import DirectoryLoader

loader = DirectoryLoader(
    "/content/documentos/",
    glob="**/*.pdf",  # Buscar recursivamente
    loader_cls=PyPDFLoader
)
docs = loader.load()

# Carga TODOS los PDFs de la carpeta
# y subcarpetas automáticamente
```

Otros loaders útiles

- **TextLoader:** archivos .txt planos
- **CSVLoader:** cada fila como documento
- **UnstructuredHTMLLoader:** páginas web parseadas
- **NotionDirectoryLoader:** exportaciones de Notion
- **GitbookLoader:** documentación técnica

Paso 2: Splitting (Fragmentación)

Por qué no podemos enviar documentos completos al LLM

Razón 1: Límites de contexto

Incluso modelos con 128k tokens de contexto no pueden procesar un PDF de 500 páginas (≈ 500k tokens). Necesitamos dividir el contenido en piezas manejables.

Razón 2: Precisión de recuperación

Buscar en un párrafo específico es mucho más preciso que buscar en un capítulo entero. La granularidad fina mejora la relevancia semántica.

RecursiveCharacterTextSplitter

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,        # Tamaño objetivo del fragmento
    chunk_overlap=200,      # Superposición entre chunks
    length_function=len,    # Función para medir longitud
    separators=["\n\n", "\n", " ", ""]  # Prioridad de separación
)

splits = text_splitter.split_documents(docs)
```

¿Por qué "Recursive"?

Intenta dividir primero por párrafos (\\n\\n), si un párrafo es demasiado grande, divide por líneas (\\n), si aún es grande, por palabras ( ), y como último recurso, por caracteres. Esto preserva la estructura semántica lo más posible.

Estrategias de Chunking

El arte de dividir texto sin romper significado

chunk_size: Equilibrando precisión y contexto

- **500 chars:** Muy granular, máxima precisión en búsqueda, pero puede perder contexto de párrafos largos
- **1000 chars:** Balance estándar, bueno para la mayoría de casos, ≈ 250 tokens
- **2000 chars:** Chunks grandes, más contexto pero menor precisión en búsqueda

chunk_overlap: Evitando pérdida de información

Si cortamos justo en medio de una oración importante, podríamos perder contexto crítico. La superposición garantiza que oraciones que cruzan límites de chunks aparecen completas en al menos uno.

Ejemplo: "La empresa fue fundada en 1995. Su misión es..." → La superposición asegura que ambas oraciones estén juntas en un chunk.

Rule of thumb: chunk_overlap debería ser 10-20% de chunk_size. 200 chars de overlap con 1000 chars de chunk es una buena configuración default.

Paso 3: Embeddings

Transformando texto en números que capturan significado

Los embeddings son representaciones numéricas densas donde textos con significados similares están cerca en el espacio vectorial. "Perro" y "cachorro" tendrán vectores cercanos, mientras "perro" y "galaxia" estarán lejos.

Selección del modelo de embeddings

```
from langchain_community.embeddings import HuggingFaceEmbeddings

embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={'device': 'cuda'},  # Usar GPU
    encode_kwargs={'normalize_embeddings': True}
)

# Genera un vector de 384 dimensiones
vector = embeddings.embed_query("Hola mundo")
print(len(vector))  # 384
```

Opciones populares

- **all-MiniLM-L6-v2:** Rápido, 384 dims, bueno para español/inglés
- **multilingual-e5-large:** 1024 dims, excelente para español
- **OpenAI text-embedding-3-small:** Via API, 1536 dims

¿Por qué no usar el LLM directamente?

Los modelos de embeddings están especializados en capturar similitud semántica. Son más pequeños (350MB vs. 15GB), más rápidos (10ms vs. 2s), y producen vectores optimizados para búsqueda.

Paso 4: Vector Store (Base de Datos Vectorial)

Donde vive el conocimiento indexado

Una vez que tenemos vectores, necesitamos una estructura de datos optimizada para búsqueda de similitud. Las bases de datos SQL tradicionales no pueden hacer esto eficientemente.

ChromaDB

Características: Embebida, corre en el mismo proceso de Python, perfecta para Colab y prototipos

Pros: Configuración cero, no requiere servidor separado

Cons: No escalable para millones de vectores

FAISS

Características: Librería de Facebook, extremadamente rápida, optimizada para GPU

Pros: Máximo rendimiento para búsquedas

Cons: Solo índice en memoria, sin persistencia nativa

Pinecone

Características: Servicio cloud managed, API REST

Pros: Escalable a billones de vectores, sin mantenimiento

Cons: Costo mensual, requiere conexión internet

En memoria vs. persistente

ChromaDB puede correr completamente en RAM (rápido, se pierde al cerrar notebook) o persistir a disco (sobrevive reinicios, ligeramente más lento). Para desarrollo: memoria. Para producción: persistente.

Creando el Vector Store (Código)

Indexando nuestros documentos fragmentados

```
from langchain_community.vectorstores import Chroma

# Crear el índice vectorial en un solo paso
vectorstore = Chroma.from_documents(
    documents=splits,              # Los chunks del paso 2
    embedding=embeddings,          # El modelo de embeddings del paso 3
    persist_directory="./chroma_db"  # Persistir a disco
)

# Esto automáticamente:
# 1. Genera embeddings para cada chunk
# 2. Construye el índice HNSW para búsqueda rápida
# 3. Guarda todo en disco para reusabilidad

print(f"Indexados {vectorstore._collection.count()} chunks")
```

¿Qué sucede internamente?

1. Para cada uno de los 150 chunks, llama a `embeddings.embed_documents()`
2. Genera 150 vectores de 384 dimensiones cada uno
3. Construye índice HNSW (Hierarchical Navigable Small World) para búsqueda O(log n)
4. Serializa índice + metadata a SQLite en `./chroma_db`

Carga de índice existente

```
# En sesiones futuras, no re-indexar
vectorstore = Chroma(
    persist_directory="./chroma_db",
    embedding_function=embeddings
)

# Carga instantánea desde disco
```

Guardar el índice es crucial en Colab porque las sesiones gratuitas se desconectan. Sin persistencia, tendrías que re-indexar todo cada vez.

Paso 5: El Retriever

Convirtiendo la base de datos en un buscador inteligente

El Retriever es una abstracción que toma una query de texto y devuelve los documentos más relevantes. Es el puente entre "pregunta del usuario" y "contexto para el LLM".

Similarity Search (búsqueda por similitud)

```
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)

# Buscar chunks relevantes
docs = retriever.get_relevant_documents(
    "¿Cuántos días de vacaciones tengo?"
)

for doc in docs:
    print(doc.page_content)
    print(doc.metadata)
```

Convierte la query a vector, calcula distancia coseno con todos los chunks, devuelve los k más cercanos.

MMR (Maximal Marginal Relevance)

```
retriever = vectorstore.as_retriever(
    search_type="mmr",
    search_kwargs={
        "k": 5,
        "fetch_k": 20,
        "lambda_mult": 0.5
    }
)
```

Objetivo: Balancear relevancia con diversidad. Primero obtiene 20 candidatos, luego selecciona 5 que son relevantes PERO también diversos entre sí. Evita devolver 5 chunks casi idénticos.

Parámetro k: Más chunks = más contexto pero más ruido. Menos chunks = respuestas más precisas pero pueden perder información. k=3-5 es un buen default. Experimentar es clave.

Paso 6: La Cadena RAG (RetrievalQA)

Orquestando todo el flujo en una sola interfaz

Finalmente unimos retriever + LLM en una cadena que automáticamente: (1) busca contexto relevante, (2) lo inyecta en el prompt, (3) genera respuesta basada en ese contexto.

```
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(
    llm=llm,                          # El modelo Llama 3.1 cuantizado
    chain_type="stuff",               # "stuff" = meter todo el contexto en un prompt
    retriever=vectorstore.as_retriever(search_kwargs={"k": 3}),
    return_source_documents=True      # Devolver los chunks usados (citar fuentes)
)

# Hacer una pregunta
resultado = qa_chain({"query": "¿Cuáles son los requisitos del proyecto?"})

print(resultado['result'])            # La respuesta generada
print(resultado['source_documents'])  # Los 3 chunks usados como contexto
```

chain_type opciones

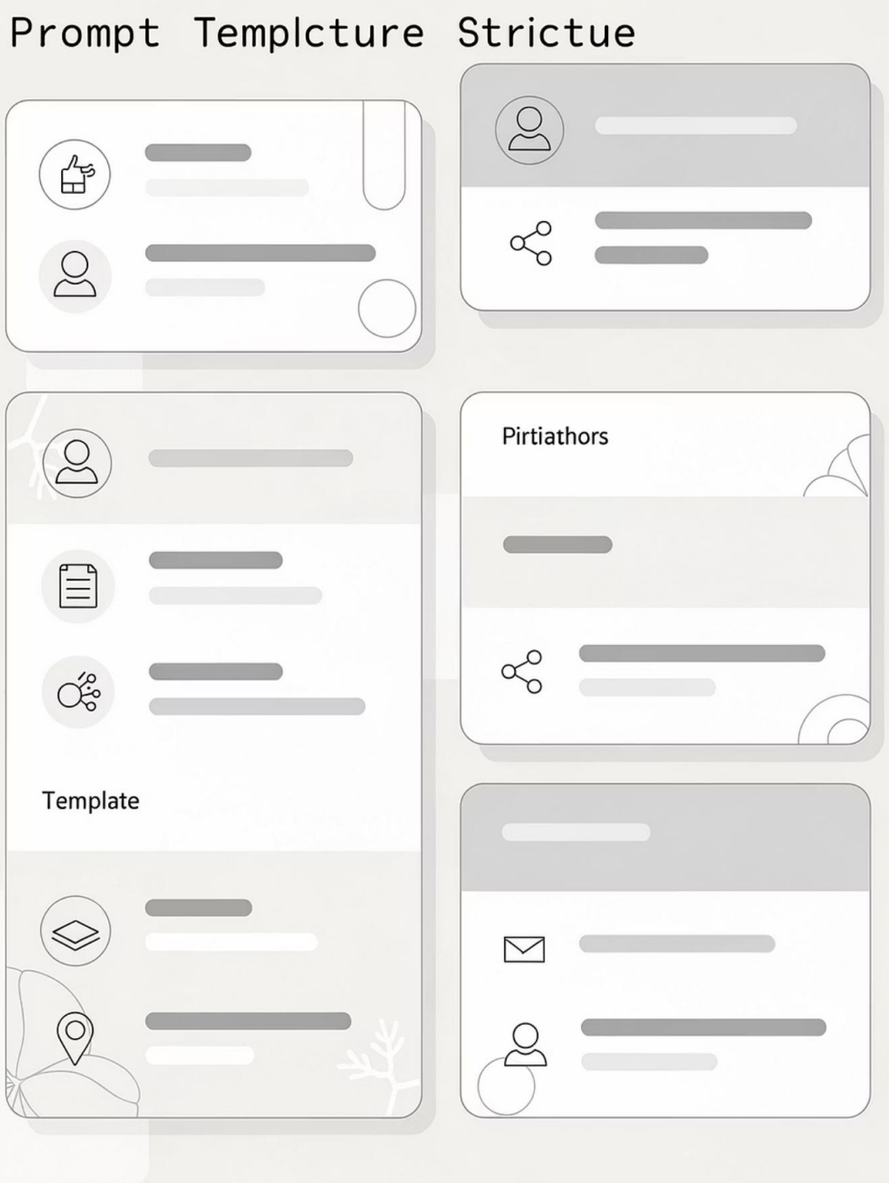
- **"stuff":** Concatena todos los chunks en un solo prompt (simple, funciona si caben en contexto)
- **"map_reduce":** Procesa cada chunk separadamente, luego combina resultados (para muchos chunks)
- **"refine":** Respuesta iterativa que mejora con cada chunk (para respuestas largas)

Código de la Cadena RAG

Configuración completa con template personalizado

```
from langchain.chains import RetrievalQA
from langchain_core.prompts import PromptTemplate

# Template que instruye al LLM a usar el contexto recuperado
template = """Usa el siguiente contexto para responder la pregunta del usuario.
Si no sabes la respuesta basándote en el contexto, di "No tengo suficiente información".
No inventes información.

Contexto: {context}

Pregunta: {question}

Respuesta útil:"""

prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"]
)

qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vectorstore.as_retriever(search_kwargs={"k": 3}),
    return_source_documents=True,
    chain_type_kwargs={"prompt": prompt}  # Inyectar nuestro template
)
```

Clave del template: La instrucción "Si no sabes, di que no sabes" reduce alucinaciones dramáticamente. Forzamos honestidad intelectual en lugar de inventar respuestas.



Prompt Engineering para RAG

Adaptando el formato para Llama 3.1

Llama 3 espera el formato de chat template con sus tokens especiales. Necesitamos adaptar el prompt RAG para respetar esta estructura.

Template genérico (menos efectivo)

```
Contexto: {context}
Pregunta: {question}
Respuesta:
```

Funciona pero no aprovecha el fine-tuning del modelo para instrucciones conversacionales.

Template optimizado para Llama 3

```
<|begin_of_text|>
<|start_header_id|>system<|end_header_id|>

Eres un asistente experto. Responde SOLO
basándote en el contexto proporcionado. Si el
contexto no contiene la respuesta, admite que no
sabes.
<|eot_id|>

<|start_header_id|>user<|end_header_id|>

Contexto:
{context}

Pregunta: {question}
<|eot_id|>

<|start_header_id|>assistant<|end_header_id|>
```

El formato correcto activa el "modo instrucción" del modelo, resultando en respuestas más precisas y obedientes al system prompt.

Ejecución y Prueba

Probando el sistema RAG completo

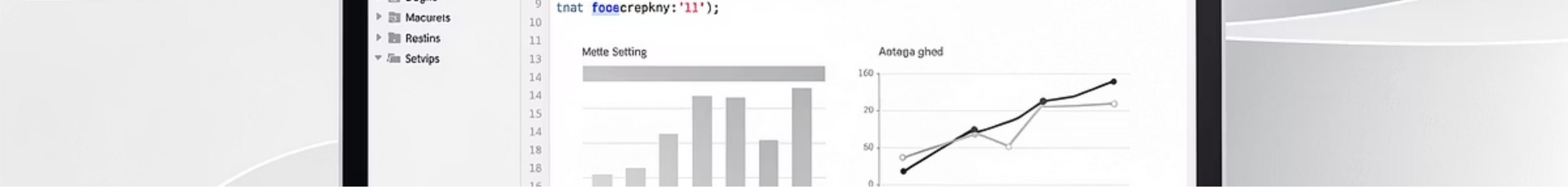
```
resultado = qa_chain({"query": "¿Cuáles son los requisitos del proyecto?"})

print("=== RESPUESTA ===")
print(resultado['result'])

print("\n=== FUENTES CONSULTADAS ===")
for i, doc in enumerate(resultado['source_documents'], 1):
    print(f"\n--- Fuente {i} ---")
    print(f"Archivo: {doc.metadata.get('source', 'N/A')}")
    print(f"Página: {doc.metadata.get('page', 'N/A')}")
    print(f"Contenido: {doc.page_content[:200]}...")  # Primeros 200 chars
```

Output de ejemplo

```
=== RESPUESTA ===
Según el documento del proyecto, los requisitos principales son:
1. Python 3.8 o superior
2. Dependencias listadas en requirements.txt
3. Cuenta activa de AWS para deployment
4. Base de datos PostgreSQL 13+
```

```
=== FUENTES CONSULTADAS ===

--- Fuente 1 ---
Archivo: /content/proyecto_setup.pdf
Página: 3
Contenido: Requisitos técnicos
El proyecto requiere Python 3.8+ y todas las librerías especificadas en el
archivo...
```

Ventaja clave de return_source_documents=True: Permite auditar de dónde vino la información. Crítico para aplicaciones reguladas (legal, médico, financiero).



Análisis del Notebook (Llama 3.1)

Revisión del código del Colab compartido

Celda 1: Instalación optimizada

1

```
!pip install -q transformers accelerate bitsandbytes
!pip install -q langchain langchain-community sentence-transformers chromadb
```

Flag `-q` (quiet) reduce el output verbose. Orden importa: transformers primero por dependencias.

Celda 2: Configuración de cuantización

2

BitsAndBytesConfig con 4-bit, compute_dtype=bfloat16, double_quant=True. Esta configuración específica permite correr el modelo 8B en T4 GPU sin OOM errors.

Celda 3: Carga del modelo con device_map="auto"

3

Accelerate calcula qué capas caben en GPU y automáticamente hace offloading del resto a CPU. Usuario no necesita gestionar memoria manualmente.

Celda 4: Pipeline de HuggingFace wrapeado en LangChain

4

HuggingFacePipeline conecta transformers con LangChain. Temperature=0.1 para respuestas deterministas en RAG (importante para reproducibilidad).

Manejo de Alucinaciones en RAG

RAG reduce pero no elimina alucinaciones

Incluso con contexto recuperado, el LLM puede ignorarlo y generar información falsa si el contexto no responde directamente la pregunta. Necesitamos múltiples capas de validación.

1

Prompt Engineering Defensivo

Instrucción explícita: "Si el contexto no contiene la respuesta, responde 'No tengo información suficiente' en lugar de adivinar". Aumenta honestidad del modelo.

2

Evaluación de Relevancia Pre-Generación

Antes de generar respuesta, hacer que el LLM evalúe si los chunks recuperados son suficientes: "¿Este contexto responde la pregunta? Sí/No". Si No, no generar respuesta.

3

Verificación Post-Generación

Usar un segundo modelo (más pequeño/rápido) para verificar si la respuesta está grounded en el contexto. Si no, rechazar la respuesta y regenerar.

4

Citas Forzadas

Template que exige: "Cita el texto exacto del contexto que respalda tu respuesta entre comillas". Fuerza al modelo a anclarse en el texto recuperado.

Técnica avanzada: Implementar un sistema de "confianza" donde el modelo asigna una probabilidad a su respuesta. Respuestas < 70% de confianza se marcan como "inciertas" en la UI.

RAG sobre Múltiples Documentos

Escalando a corpus grandes y diversos

Ingesta de carpeta completa

```
from langchain_community.document_loaders import DirectoryLoader

loader = DirectoryLoader(
    "/content/drive/MyDrive/docs_empresa/",
    glob="**/*.{pdf,txt,docx}",
    show_progress=True
)

docs = loader.load()
print(f"{len(docs)} documentos cargados")
```

Preservando metadata crucial

Cada Document object tiene un dict `metadata`. Podemos enriquecer con información útil:

```
for doc in docs:
    doc.metadata['departamento'] = extraer_dept(doc.metadata['source'])
    doc.metadata['fecha_modificacion'] = obtener_timestamp(doc)
    doc.metadata['autor'] = extraer_autor(doc)
```

Self-Querying: Filtrado por metadata

```
from langchain.retrievers import SelfQueryRetriever

retriever = SelfQueryRetriever.from_llm(
 llm=llm,
 vectorstore=vectorstore,
 document_contents="Políticas y procedimientos corporativos",
 metadata_field_info=[
 {"name": "departamento", "type": "string"},
 {"name": "fecha_modificacion", "type": "date"}
 ]
)

# Query con filtro implícito
docs = retriever.get_relevant_documents(
 "¿Cuál es la política de RRHH sobre vacaciones actualizada en 2024?"
)
# Automáticamente filtra por departamento='RRHH' y fecha >= 2024
```

Persistencia

Guardando el índice vectorial para reusabilidad

En Google Colab, las sesiones son efímeras. Sin persistencia, cada vez que se desconecta el runtime, perdemos toda la indexación y debemos recomenzar desde cero (costoso en tiempo y cómputo).

Guardar en Google Drive

```
from google.colab import drive
drive.mount('/content/drive')

# Crear índice con persistencia en Drive
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=embeddings,
    persist_directory="/content/drive/MyDrive/chroma_db"
)

# Ahora el índice sobrevive reinicios
```

Cargar índice existente

```
from google.colab import drive
drive.mount('/content/drive')

# Cargar sin re-indexar
vectorstore = Chroma(
    persist_directory="/content/drive/MyDrive/chroma_db",
    embedding_function=embeddings
)

# Instantáneo, solo lee de disco
```

Tamaño del índice

Para 10,000 chunks con embeddings de 384 dims: ~50MB. Para 100,000 chunks: ~500MB. ChromaDB usa SQLite comprimido, muy eficiente en espacio.

Buena práctica: Versionar los índices. Si actualizas documentos, crea un nuevo directorio `chroma_db_v2` en lugar de sobrescribir. Permite rollback si algo sale mal.

Optimización de RAG: Hybrid Search

Combinando búsqueda semántica con keyword matching

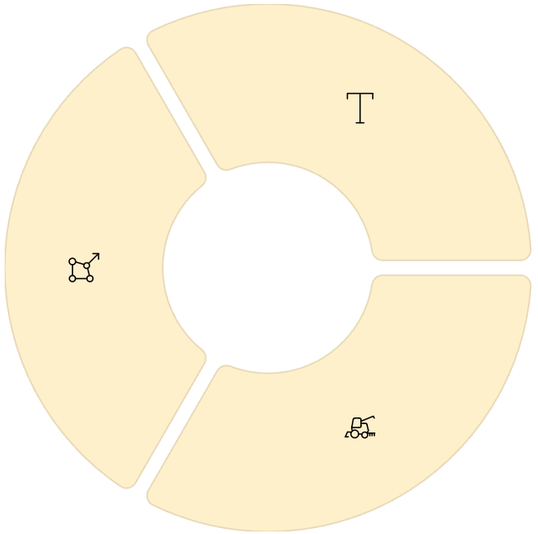
La búsqueda vectorial es excelente para similitud conceptual, pero falla con nombres propios exactos, acrónimos, o códigos (ej: "ISO-27001", "John Smith").

Búsqueda Vectorial

Entiende sinónimos y contexto. "auto" encuentra "coche" y "vehículo".

Fuerte en: Conceptos abstractos

Débil en: Matching exacto



Búsqueda BM25

Algoritmo clásico de keyword matching (TF-IDF mejorado).

Fuerte en: Términos exactos, nombres propios

Débil en: Sinónimos, contexto

Hybrid Search

Ejecuta ambas búsquedas en paralelo y fusiona resultados con pesos.

Ventaja: Lo mejor de ambos mundos

Costo: 2x latencia de búsqueda

```
from langchain.retrievers import EnsembleRetriever
from langchain.retrievers import BM25Retriever

# Retriever vectorial (ya tenemos)
vector_retriever = vectorstore.as_retriever(search_kwargs={"k": 5})

# Retriever BM25 sobre los mismos documentos
bm25_retriever = BM25Retriever.from_documents(splits)
bm25_retriever.k = 5

# Combinar con pesos
hybrid_retriever = EnsembleRetriever(
 retrievers=[vector_retriever, bm25_retriever],
 weights=[0.6, 0.4] # 60% vectorial, 40% keyword
)
```

Optimización de RAG: Re-ranking

Refinando los resultados de búsqueda

El primer retrieval (vectorial/hybrid) es rápido pero impreciso. Re-ranking usa un modelo más sofisticado para reordenar los candidatos, mejorando calidad sin impactar mucho la latencia.

Paso 1: Retrieval inicial

Búsqueda vectorial rápida recupera top 50 candidatos (~50ms)

Paso 2: Re-ranking con Cross-Encoder

Modelo BERT que evalúa la relevancia query-documento conjuntamente (~200ms para 50 docs)

Paso 3: Selección final

Reordenar por score del re-ranker y tomar top 3-5 para el prompt del LLM

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import CrossEncoderReranker
from langchain_community.cross_encoders import HuggingFaceCrossEncoder

# Modelo de re-ranking
model = HuggingFaceCrossEncoder(model_name="cross-encoder/ms-marco-MiniLM-L-6-v2")
compressor = CrossEncoderReranker(model=model, top_n=3)

# Envolver el retriever base
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=vector_retriever
)

# Ahora devuelve solo los 3 MÁS relevantes después de re-ranking
docs = compression_retriever.get_relevant_documents("query")
```

Ganancia típica: 15-25% de mejora en métricas de relevancia (NDCG@5) con solo 150ms de latencia adicional. Trade-off muy favorable.

RAG Conversacional

Añadiendo memoria contextual al sistema RAG

RAG estándar es stateless: cada pregunta se trata independientemente. Pero las conversaciones reales tienen referencias anafóricas ("¿Y cuánto cuesta ESO?" donde "eso" se refiere a algo mencionado antes).

El problema de seguimiento

```
Usuario: "¿Qué políticas tiene la empresa?"
RAG: "Tenemos política de vacaciones, remoto..."

Usuario: "¿Cuántos días da la primera?"
RAG: ???
# "la primera" no hace sentido sin contexto
```

ConversationalRetrievalChain

```
from langchain.chains import ConversationalRetrievalChain

qa = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    memory=ConversationBufferMemory(
        memory_key="chat_history",
        return_messages=True
    )
)
```

Reformulación de preguntas

Antes de recuperar documentos, ConversationalRetrievalChain reformula la pregunta actual usando el historial:

```
Historial:
Usuario: "¿Qué políticas tiene la empresa?"
Bot: "Tenemos política de vacaciones..."

Pregunta actual: "¿Cuántos días da la primera?"

Reformulación automática → "¿Cuántos días de vacaciones proporciona la política de vacaciones de la empresa?"
```

Esta reformulación standalone se usa para recuperar documentos, eliminando ambigüedad.

De Cadenas a Agentes

Cadena ≠ Agente

Cadenas: Flujo determinista

Una cadena es una secuencia fija predefinida de operaciones. El desarrollador decide el flujo:

A → B → C → D

Siempre ejecuta los mismos pasos en el mismo orden. No hay toma de decisiones. Predecible y debuggeable, pero inflexible.

Agentes: Razonamiento dinámico

Un agente observa, razona, y decide qué hacer a continuación. El LLM es el motor de decisión:

A → **¿Necesito más info? Sí → B**

B → **¿Suficiente? No → C**

C → **¿Puedo responder? Sí → Fin**

El flujo emerge del razonamiento, no está hardcoded.

El LLM como cerebro del sistema

En una cadena, el LLM solo genera texto. En un agente, el LLM decide qué herramientas usar, en qué orden, y cuándo detenerse. Es el "cerebro" que controla el cuerpo (herramientas).

Herramientas (Tools)

Dándole "brazos" y "sentidos" al LLM

Por defecto, los LLMs solo pueden generar texto. Las herramientas les permiten interactuar con el mundo: realizar cálculos precisos, buscar información actualizada, ejecutar código, consultar bases de datos.

Calculadora

Permite aritmética exacta en lugar de depender de las capacidades matemáticas imprecisas del LLM. Ejemplo: "¿Cuánto es 1247 * 8934?"

Google Search / SerpAPI

Búsqueda en tiempo real de información actualizada. Rompe la limitación de conocimiento congelado del entrenamiento.

Wikipedia

Consulta la API de Wikipedia para información factual verificable. Menos propenso a sesgo que búsqueda web general.

Python REPL

Ejecuta código Python arbitrario en un sandbox. Permite análisis de datos, visualización, procesamiento complejo.

SQL Database

Traduce preguntas en lenguaje natural a queries SQL y ejecuta contra una base de datos real.

Importante: Herramientas deben tener descripciones claras. El LLM lee esas descripciones para decidir cuándo usarlas. Descripción ambigua = herramienta mal utilizada.

ReAct (Reason + Act)

El paradigma de pensamiento de los agentes

ReAct es un framework que estructura cómo los agentes razonan y actúan. Alterna entre pensamiento interno y acciones externas hasta resolver la tarea.

Thought (Pensamiento)

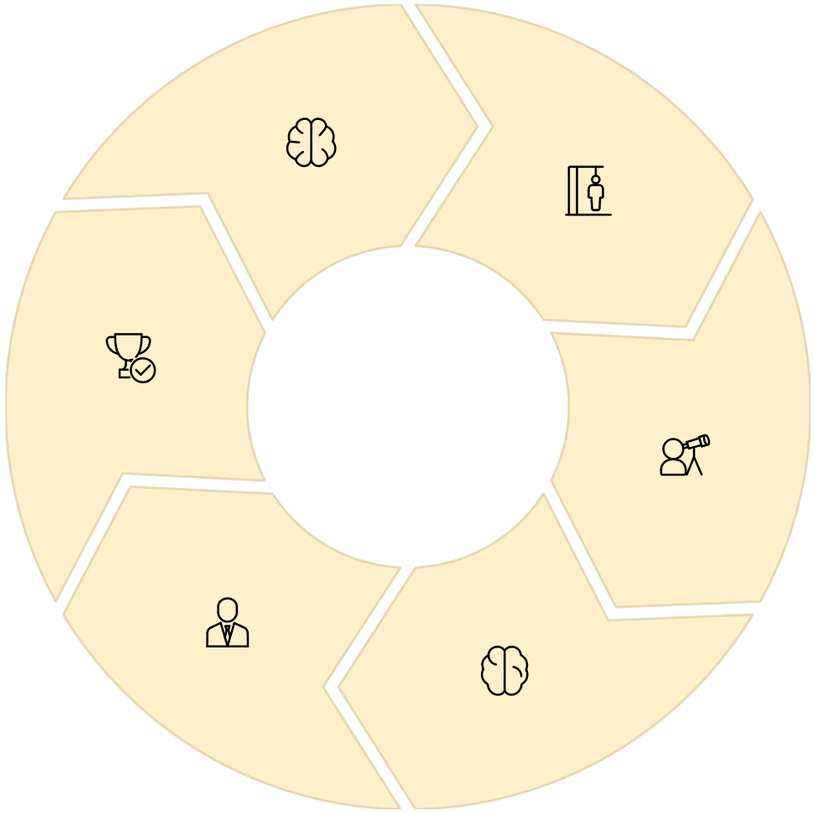
El LLM razona sobre qué hacer a continuación: "Necesito buscar la edad actual de Leo DiCaprio"

Final Answer

Observation: 125000. Thought: "Tengo la respuesta." Answer: "La edad de Leo DiCaprio elevada al cubo es 125,000"

Action (Cálculo)

Usa calculadora: `Tool: Calculator, Input: "50^3"`



Action (Acción)

Decide usar una herramienta: `Tool: Wikipedia, Input: "Leonardo DiCaprio"`

Observation (Observación)

La herramienta devuelve resultado: "Leonardo DiCaprio (born 1974) is an American actor..."

Thought (Evaluación)

Razona sobre el resultado: "Nació en 1974, ahora es 2024, tiene 50 años. Necesito elevar al cubo."

Ejemplo de Agente en LangChain

Código funcional de un agente ReAct

```
from langchain.agents import load_tools, initialize_agent, AgentType

# 1. Cargar herramientas (requiere API keys)
tools = load_tools(
    ["serpapi", "llm-math"],  # Google Search + Calculadora
    llm=llm
)

# 2. Inicializar agente
agent = initialize_agent(
    tools=tools,
    llm=llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,  # ReAct sin ejemplos
    verbose=True,  # Mostrar razonamiento completo
    max_iterations=5  # Límite para evitar loops infinitos
)

# 3. Ejecutar tarea compleja multi-herramienta
pregunta = """¿Quién es la pareja actual de Leonardo DiCaprio y
              cuál es su edad elevada al cubo?"""

respuesta = agent.run(pregunta)
```

Traza de ejecución (verbose=True)

```
Thought: Necesito averiguar quién es la pareja actual de Leo DiCaprio
Action: Search
Action Input: "Leonardo DiCaprio current girlfriend 2024"
Observation: Vittoria Ceretti (born 1998) is dating Leonardo DiCaprio...

Thought: Nació en 1998, tiene 26 años. Ahora necesito calcular 26^3
Action: Calculator
Action Input: 26^3
Observation: 17576

Thought: Tengo toda la información necesaria
Final Answer: La pareja actual de Leonardo DiCaprio es Vittoria Ceretti,
 y su edad elevada al cubo es 17,576.
```

El Futuro: Agentes Autónomos

Hacia la autonomía completa

Los agentes actuales son reactivos: el humano define la tarea, el agente ejecuta. El próximo salto es hacia agentes proactivos que planifican y persiguen objetivos de largo plazo sin supervisión constante.

1

AutoGPT (2023)

Primer intento viral de agente autónomo. Se le da un objetivo ("construir un negocio rentable online") y se auto-asigna sub-tareas, ejecuta herramientas, y evalúa progreso iterativamente.

2

BabyAGI

Framework minimalista de agente que mantiene una lista de tareas, prioriza dinámicamente, y genera nuevas tareas basándose en resultados anteriores. Loop de mejora continua.

3

MetaGPT (2024)

Simula una empresa de software completa con múltiples agentes (PM, arquitecto, developer, QA) que colaboran para completar proyectos end-to-end.

4

El desafío actual

Agentes autónomos tienen éxito limitado por: (1) alucinaciones que propagan errores, (2) falta de memoria de largo plazo efectiva, (3) costo de ejecutar miles de llamadas LLM. Investigación activa.

Debate ético: ¿Qué nivel de autonomía es seguro? Agentes que ejecutan código arbitrario o hacen compras online sin aprobación humana plantean riesgos significativos.

Evaluación de Aplicaciones (RAGAS)

Midiendo la calidad de sistemas RAG

Construir un RAG es relativamente sencillo. Construir un RAG *bueno* requiere evaluación rigurosa. RAGAS (Retrieval Augmented Generation Assessment) es un framework para medir calidad objetivamente.

0.85

Faithfulness (Fidelidad)

¿La respuesta es consistente con el contexto recuperado? Detecta cuando el LLM inventa información no presente en los chunks.

0.92

Answer Relevance (Relevancia)

¿La respuesta realmente aborda la pregunta del usuario? Evita respuestas tangenciales o evasivas.

0.78

Context Precision (Precisión)

¿Los chunks recuperados son realmente relevantes para la pregunta? Mide calidad del retriever.

0.88

Context Recall (Cobertura)

¿Se recuperaron TODOS los chunks relevantes disponibles? Detecta cuando el retriever pierde información importante.

Implementación básica

```
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy

# Dataset de evaluación con ground truth
eval_dataset = [
 {"question": "...", "answer": "...", "contexts": [...], "ground_truth": "..."},
 ...
]

# Ejecutar evaluación
result = evaluate(
 eval_dataset,
 metrics=[faithfulness, answer_relevancy]
)

print(result) # Scores para cada métrica
```

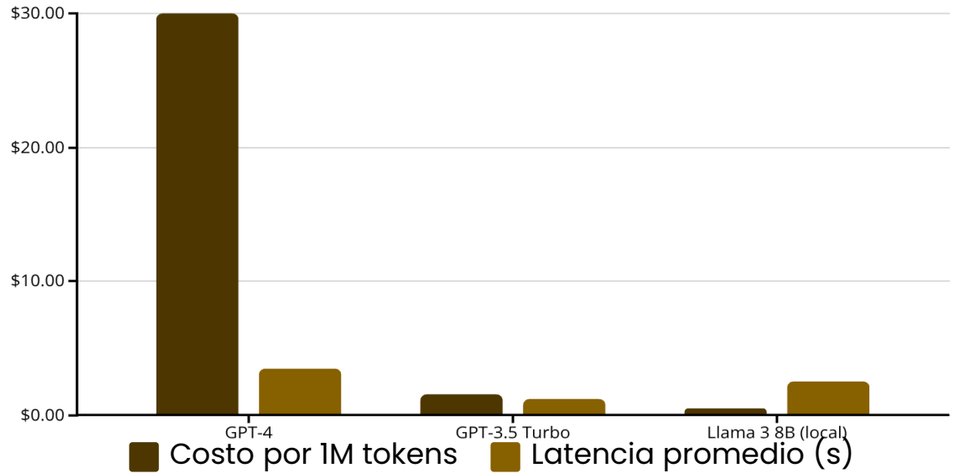
Costos y Latencia

El trade-off económico de aplicaciones LLM

Estructura de costos

- **Tokens de entrada:** Prompt (system + contexto RAG + query) puede ser 2000-5000 tokens
- **Tokens de salida:** Respuesta generada típicamente 100-500 tokens
- **Embeddings:** $0.0001 por 1000 tokens (mucho más barato que generación)
- **Cómputo GPU:** Si self-hosted, $1-3 por hora de GPU A100

Comparación de modelos



Optimización de costos

- Usar modelos pequeños (GPT-3.5/Llama) para tareas simples, grandes solo cuando sea necesario
- Cachear respuestas frecuentes (evitar regenerar la misma respuesta)
- Comprimir contexto RAG (eliminar información redundante)
- Self-host modelos open source para alto volumen

Privacidad y Ética en Apps

Responsabilidad en producción

PII en APIs públicas

Problema: Enviar información personal identificable (nombres, emails, números de identificación) a APIs de OpenAI/Anthropic significa que esa data pasa por servidores de terceros.

Solución: Anonimizar datos antes de enviar, o usar modelos self-hosted locales para datos sensibles.

Retención y logging

Problema: Los prompts completos (incluyendo contexto RAG) se loguean para debugging. Pueden contener información confidencial.

Solución: Implementar políticas de retención (borrar logs después de 30 días), encriptar logs en reposo, limitar acceso.

Sesgos algorítmicos

Problema: LLMs heredan sesgos del training data. Un asistente de RRHH podría discriminar basándose en edad, género, o etnia implícitamente.

Solución: Auditorías de fairness, red-teaming con casos adversarios, prohibir decisiones automatizadas en contextos de alto riesgo.

Ventaja de Llama 3 local

Ejecutar Llama 3 en tu propia infraestructura (Colab, servidor on-prem) significa que los datos nunca salen de tu control. 100% privacidad. Crítico para sectores regulados: salud (HIPAA), finanzas (SOX), gobierno.

Resumen del Módulo

Lo que hemos dominado juntos

Paisaje de aplicaciones generativas

Comprendemos las categorías principales (generación, resumen, transformación, extracción) y sus casos de uso empresariales

Arquitectura con LangChain

Sabemos estructurar aplicaciones usando Models, Prompts, Chains, Memory, y Agents. Entendemos LCEL y composición de flujos

Sistemas conversacionales sofisticados

Implementamos chatbots con memoria persistente, system prompts personalizados, y manejo de contexto largo

RAG end-to-end

Construimos un sistema completo: ingesta, chunking, embeddings, vector store, retrieval, y generación. Optimizamos con hybrid search y re-ranking

Implementación práctica con Llama 3.1

Cargamos y cuantizamos modelos grandes en GPUs limitadas. Ejecutamos inferencia local manteniendo privacidad

Estamos listos para construir prototipos funcionales y llevarlos a producción con conciencia de costos, latencia, y ética.

Cierre y Próximos Pasos

Tu viaje de aprendizaje continúa

Tarea práctica

Modifica el notebook del módulo para trabajar con tus propios documentos:

1. Sube 3-5 PDFs de tu dominio de interés (papers académicos, manuales técnicos, documentación interna)
2. Ajusta los parámetros de chunking (experimenta con diferentes chunk_size)
3. Personaliza el system prompt para tu caso de uso específico
4. Evalúa la calidad de las respuestas y documenta mejoras

Recursos para profundizar

- **LangChain Docs:** langchain.com/docs (documentación oficial exhaustiva)
- **HuggingFace Course:** huggingface.co/learn (módulos gratuitos sobre transformers)
- **Papers fundamentales:** "Retrieval-Augmented Generation" (Lewis et al., 2020), "ReAct" (Yao et al., 2023)
- **Community:** LangChain Discord, r/LocalLLaMA en Reddit

Experimentación

La mejor forma de aprender es construyendo. Prueba, falla, itera.

Comunidad

El ecosistema LLM evoluciona semanalmente. Mantente conectado.

Ética primero

Con gran poder viene gran responsabilidad. Construye con consciencia.

¿Preguntas finales?

Gracias por su atención y participación activa en este módulo. ¡Éxito en sus proyectos!