

# Unidad 6 - Chatbots y Sistemas de Diálogo

**UNR - TUIA - Procesamiento de Lenguaje Natural**

Docente teoría: Juan Pablo Manson - [jpmanson@gmail.com](mailto:jpmanson@gmail.com) - [LinkedIN](#)

## 1. Chatbots

### Introducción

Los chatbots, asistentes virtuales o sistemas de diálogo son programas informáticos diseñados para simular una conversación con usuarios humanos, especialmente a través de Internet. Estos sistemas combinan tecnologías de inteligencia artificial como el procesamiento de lenguaje natural (NLP) y el aprendizaje automático (ML) para comprender, procesar y responder a las solicitudes verbales o textuales de los usuarios. A menudo se les asigna tareas específicas, como proporcionar información, asistencia al cliente, realizar reservas o controlar dispositivos y sistemas a través de comandos de voz o texto.

Aunque los términos "chatbots", "asistentes virtuales" y "sistemas de diálogo" a menudo se usan indistintamente, existen ciertas diferencias sutiles entre ellos basadas en sus funciones, capacidades y usos previstos. A continuación, se describen las diferencias entre estos términos: (1)

#### 1. Chatbots:

- **Función:** Diseñados principalmente para interactuar mediante texto con los usuarios, ofreciendo respuestas y realizando tareas simples basadas en un

conjunto de entradas predefinidas o mediante la comprensión del lenguaje natural.

- **Capacidad:** Los chatbots varían desde sistemas basados en reglas muy básicos hasta sistemas más avanzados impulsados por inteligencia artificial.
- **Uso:** Comúnmente se implementan en sitios web y aplicaciones de mensajería para asistencia al cliente, FAQs, y para tareas específicas de comercio electrónico o servicios informativos.

## 2. Asistentes Virtuales:

- **Función:** Estos sistemas están diseñados para comprender y ejecutar una amplia gama de tareas, como configurar alarmas, realizar llamadas, enviar mensajes, proporcionar información en tiempo real y controlar dispositivos inteligentes, a menudo a través de comandos de voz.
- **Capacidad:** Suelen incorporar capacidades más sofisticadas de inteligencia artificial, como el aprendizaje automático y la comprensión del contexto, lo que les permite realizar tareas más complejas y personalizadas.
- **Uso:** Son integrados en smartphones, altavoces inteligentes, y sistemas de automatización del hogar, siendo ejemplos prominentes Siri de Apple, Alexa de Amazon y el Asistente de Google. **(2)**

## 3. Sistemas de Diálogo:

- **Función:** Son plataformas o frameworks diseñados para gestionar conversaciones con humanos, permitiendo un diálogo bidireccional que puede adaptarse y cambiar según la entrada del usuario.
- **Capacidad:** Pueden ser tanto sistemas de voz como de texto y generalmente implican una estructura más compleja que puede manejar conversaciones de ida y vuelta, reconocimiento de entidades, comprensión del sentimiento, y pueden tener memoria de contexto a lo largo de una sesión.
- **Uso:** Suelen encontrarse en aplicaciones donde es necesaria una interacción más profunda, como soporte técnico avanzado, diagnósticos médicos o asistentes personales virtuales.



Los tres términos se refieren a tecnologías que permiten la interacción entre humanos y máquinas, los chatbots tienden a ser más específicos y están centrados en tareas, los asistentes virtuales son más amplios y centrados en el usuario, y los sistemas de diálogo proporcionan un marco más robusto para el intercambio conversacional. En la actualidad se habla generalmente de chatbot, como una manera genérica de referirse a los sistemas conversacionales que utilizan IA para un entendimiento avanzado del lenguaje natural. (por ej: ChatGPT, Bard, Claude 2, etc.) **(3)**

## Historia

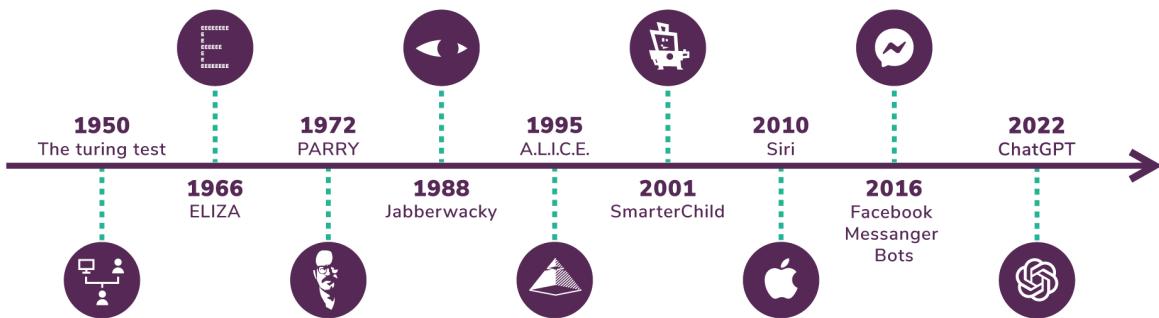
La historia de los chatbots se remonta a mediados de la década de 1960, cuando ELIZA, uno de los primeros chatbots creados, fue inventado por Joseph Weizenbaum en el MIT. ELIZA fue diseñado para imitar las respuestas de un terapeuta con el fin de proporcionar apoyo terapéutico a los pacientes.**(4)**

```
Welcome to
      EEEEEE  LL      IIII    ZZZZZZ  AAAAAA
      EE       LL      II      ZZ      AA      AA
      EEEEEE  LL      II      ZZZ     AAAAAAAA
      EE       LL      II      ZZ      AA      AA
      EEEEEE  LLLLLL  IIII    ZZZZZZ  AA      AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU: Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU: They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU: Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU: He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU: It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

En 1972, PARRY se convirtió en uno de los primeros chatbots diseñados para vencer a un oponente humano en una prueba de Turing. PARRY se basaba en las alucinaciones esquizofrénicas paranoicas de su creador, Kenneth Colby.



Desde entonces, los chatbots han recorrido un largo camino, evolucionando desde programas simples de coincidencia de patrones hasta su adopción masiva a principios de los años 2000 en servicios como AOL y MSN Messenger.

En 2016, la adopción generalizada de chatbots explotó cuando Facebook anunció que comenzaría a permitir bots en su popular plataforma de mensajería. Para el año 2018, había más de 300,000 chatbots activos en Facebook Messenger.

En los últimos años, la tecnología de los chatbots ha avanzado rápidamente debido a los avances en inteligencia artificial y procesamiento del lenguaje natural. Los chatbots de hoy son más sofisticados que nunca y se utilizan en una variedad de aplicaciones que van desde el servicio al cliente hasta el marketing y la atención médica.

La historia de los chatbots y los sistemas de diálogo ha experimentado varios puntos de inflexión significativos, especialmente con la aparición de los modelos de Transformers, que cambiaron la forma en que las máquinas entienden y generan lenguaje humano. Aquí hay un breve recuento desde la aparición de los Transformers hasta el desarrollo de ChatGPT: (5)

### 2017: Aparición de los Transformers

- En junio de 2017, Google publica el artículo “**Attention Is All You Need**”, introduciendo la arquitectura Transformer. Esta nueva arquitectura basada en mecanismos de atención permitía a los modelos prestar atención a diferentes partes de la entrada de texto de manera **más efectiva** que las técnicas anteriores como las redes neuronales recurrentes (**RNN**) y las redes de memoria a largo plazo (**LSTM**).
- Los Transformers marcaron el comienzo de una nueva era en el procesamiento del lenguaje natural (PLN), ofreciendo mejoras significativas en tareas como la traducción automática, el resumen de textos y la comprensión de lectura.

### 2018: BERT y GPT-2

- BERT (Bidirectional Encoder Representations from Transformers) fue presentado por Google en 2018 y demostró ser un punto de referencia en la capacidad de las máquinas para comprender el contexto en el texto.

- OpenAI lanzó GPT-2 en febrero de 2019. Aunque no fue inmediatamente liberado en su versión completa por temores de mal uso, GPT-2 fue un modelo de lenguaje que mejoró la generación de texto coherente y relevante.

### **2019-2020: GPT-3 y su impacto**

- GPT-3 fue anunciado en junio de 2020 por OpenAI. Su modelo de 175 mil millones de parámetros superó significativamente a su predecesor en términos de tamaño y capacidad, generando texto que a menudo era indistinguible de lo escrito por humanos.
- GPT-3 fue ampliamente reconocido por su capacidad para realizar muchas tareas de NLP sin entrenamiento específico de la tarea, simplemente mediante instrucciones en lenguaje natural (prompts).

### **2021: Modelos de Transformers en aplicaciones prácticas**

- Empresas y servicios empezaron a incorporar modelos como BERT y GPT-3 en sus productos, ofreciendo mejoras en chatbots, herramientas de escritura, motores de búsqueda y más.
- Los chatbots mejorados por estos modelos se hicieron más prevalentes en la atención al cliente, e-commerce y en aplicaciones de entretenimiento y educación.

### **2022-2023: ChatGPT y su evolución**

- ChatGPT, basado en la arquitectura de GPT-3.5, fue optimizado específicamente para conversaciones, y lanzado por OpenAI a finales de 2022. Ofrece una experiencia de chat más fluida y contextuada que las iteraciones anteriores.
- GPT-4: Mientras que GPT-3.5 solo acepta peticiones en texto, **GPT-4 es multimodal**, es decir, admite entradas en texto y visuales. O lo que es lo mismo: no tiene por qué ser una imagen con texto escrito (aunque valdría), si no que vale cualquier cosa, desde una fotografía de un paisaje a un problema matemático manuscrito pasando por un meme. GPT-4 es capaz de comprender y describir prácticamente cualquier imagen. Es decir, que GPT-4 ya no es "solo" un **modelo de lenguaje por Inteligencia Artificial, si no también un modelo visual**. Entre sus posibilidades está la de identificar objetos concretos dentro de una foto con muchos elementos visuales. **(6)**

## **Clasificaciones**

Los chatbots o sistemas de diálogo se han vuelto cada vez más populares debido a su capacidad para interactuar con los usuarios y ofrecer soluciones a diversas tareas. Pueden clasificarse según diversos criterios. Aquí vemos algunas de las clasificaciones más comunes:

### **1. Segundo la capacidad de procesamiento:**

- **Chatbots basados en reglas:** Estos chatbots tienen respuestas predefinidas para preguntas específicas. Funcionan con un conjunto de reglas y no pueden

manejar preguntas que no estén programadas en su sistema.

- **Chatbots basados en inteligencia artificial (IA):** Utilizan el procesamiento del lenguaje natural (PLN) y aprendizaje automático para entender y responder a las preguntas de los usuarios. Estos chatbots pueden aprender de las interacciones con los usuarios y mejorar con el tiempo.

## 2. Segundo criterio:

- **Chatbots informativos:** Proporcionan información específica cuando se les pregunta, como el clima, noticias o datos sobre algún tema en particular.
- **Chatbots de servicio al cliente:** Están diseñados para resolver problemas específicos de los usuarios, como problemas con un producto o servicio.
- **Chatbots de ventas y marketing:** Ayudan a guiar a los usuarios a través de un proceso de compra o proporcionan recomendaciones de productos.
- **Chatbots en sitios web:** Están integrados en sitios web y asisten a los visitantes con tareas como navegación, preguntas frecuentes o soporte.
- **Chatbots de entretenimiento:** Son para entretener al usuario, como los chatbots que cuentan chistes o historias. (7)

## 3. Segundo criterio:

- **Chatbots en aplicaciones de mensajería:** Operan dentro de aplicaciones de mensajería populares como WhatsApp, Facebook Messenger, Telegram, etc.
- **Chatbots en aplicaciones independientes:** Son aplicaciones móviles o de escritorio diseñadas específicamente para ofrecer funciones de chatbot.

## 4. Segundo criterio:

- **Chatbots basados en texto:** Interactúan con los usuarios principalmente a través de texto.
- **Chatbots basados en voz:** Como los asistentes virtuales (ej. Alexa, Siri, Google Assistant), interactúan con los usuarios a través de comandos de voz.
- **Chatbots multimodales:** Combinan múltiples formas de interacción, como texto, voz, imágenes y más.(8)

## 5. Segundo criterio:

- **Chatbots formales:** Están diseñados para ser directos y profesionales.
- **Chatbots amigables o casuales:** Tienen un tono más relajado y pueden usar emojis, GIFs y un lenguaje más coloquial.
- **Chatbots con personalidad definida:** Pueden estar diseñados para emular una personalidad específica, como un personaje famoso o un tipo particular de persona.

## Chatbots Basado en Reglas

Como su nombre indica, estos chatbots utilizan una serie de reglas definidas. Con este tipo de bots, la comunicación se realiza a través de reglas predefinidas y un conjunto de preguntas.

Estos chatbots no son capaces de generar sus propias respuestas, pero con un extenso conjunto de respuestas y reglas inteligentemente diseñadas, pueden demostrar ser muy útiles y productivos.

También se les conoce como bots de árbol de decisiones y la razón es que los chatbots basados en reglas se guían por un árbol de decisiones, al cliente o usuario se le da un conjunto de opciones predefinidas que conducen a las respuestas deseadas.

Una limitación de estos chatbots es que no pueden responder preguntas fuera de las reglas definidas. Pero no se puede decir que sea una desventaja ya que el principal mecanismo de estos chatbots es responder preguntas limitadas por las reglas definidas.

**Los chatbots basados en reglas se utilizan en escenarios donde un cliente o usuario desea realizar una tarea determinada, en ese caso, para automatizar ese proceso se prefieren los chatbots basados en reglas en lugar de los chatbots basados en IA.** Esas tareas pueden ser, por ejemplo, reservar un pasaje de avión, reservar una entrada de cine, preguntar sobre horarios de vuelo y muchas más. La mayoría de las empresas utilizan chatbots basados en reglas para aumentar la satisfacción del cliente y responder a sus consultas.

Los chatbots basados en reglas se utilizan como recurso de preguntas frecuentes (FAQ) y no necesitan tener una gran cantidad de conversaciones de ejemplo para alimentar su respuesta.

Si bien estos chatbots son más "seguros" y "responsables", las interacciones con el chatbot se sienten robóticas en lugar de conversacionales. Su rigidez puede producir un rechazo en algunos usuarios, ya que no se percibe una conversación natural. **(9)**

Veamos un ejemplo básico:

```
import re
import random
from datetime import datetime

# Base de conocimiento del chatbot
knowledge_base = {
    'saludos': {
        'patrones': [
            r'\b(hola|hey|buenos días|buenas tardes|buenas noches|hi|qué tal)\b',
        ],
        'respuestas': [
            '¡Hola! ¿Cómo puedo ayudarte?',
        ]
    }
}
```

```

        '¡Bienvenido! ¿En qué puedo asistirte?',
        '¡Hola! Estoy aquí para ayudarte'
    ],
},
'despedidas': {
    'patrones': [
        r'\b(adiós|chau|hasta luego|nos vemos|bye)\b'
    ],
    'respuestas': [
        '¡Hasta luego! Que tengas un buen día',
        '¡Adiós! Gracias por tu visita',
        '¡Nos vemos pronto!'
    ]
},
'agradecimientos': {
    'patrones': [
        r'\b(gracias|te agradezco|muchas gracias)\b'
    ],
    'respuestas': [
        '¡De nada! Estoy para ayudar',
        'Es un placer poder ayudarte',
        'No hay de qué. ¿Necesitas algo más?'
    ]
},
'preguntas_estado': {
    'patrones': [
        r'\b(cómo estás|qué tal estás|cómo te encuentras)\b'
    ],
    'respuestas': [
        'Muy bien, ¡gracias por preguntar! ¿En qué puedo ayudarte?',
        'Excelente y listo para ayudarte. ¿Qué necesitas?',
        '¡Todo bien! ¿Cómo puedo asistirte hoy?'
    ]
},
'productos': {
    'patrones': [
        r'\b(qué productos|qué vendés|qué tienen|qué ofrecen)\b'
    ],
    'respuestas': [
        'Tenemos una amplia variedad de productos: laptops, smartphones, tablets y accesorios',
        'Ofrecemos productos electrónicos como computadoras, celulares y tablets',
        'Vendemos todo tipo de dispositivos electrónicos y sus accesorios'
    ]
}

```

```

    },
    'precios': {
        'patrones': [
            r'\b(cuánto cuesta|precio|valor|cuánto vale)\b'
        ],
        'respuestas': [
            'Los precios varían según el producto específico. ¿Qué producto te interesa?',
            'Para darte el precio exacto, ¿podrías decirme qué producto te interesa?',
            '¿Qué producto específico te gustaría consultar?'
        ]
    }
}

# Productos y sus precios
productos = {
    'laptop': {'precio': 800, 'descripcion': 'Laptop de última generación'},
    'smartphone': {'precio': 500, 'descripcion': 'Smartphone con gran cámara'},
    'tablet': {'precio': 300, 'descripcion': 'Tablet perfecta para entretenimiento'},
    'auriculares': {'precio': 50, 'descripcion': 'Auriculares inalámbricos'}
}

def get_time_based_greeting():
    """Retorna un saludo basado en la hora del día"""
    hora = datetime.now().hour
    if 5 <= hora < 12:
        return "¡Buenos días!"
    elif 12 <= hora < 20:
        return "¡Buenas tardes!"
    else:
        return "¡Buenas noches!"

def buscar_producto(mensaje):
    """Busca menciones de productos en el mensaje"""
    for producto in productos.keys():
        if producto in mensaje.lower():
            return producto
    return None

def procesar_mensaje(mensaje):
    """Procesa el mensaje del usuario y retorna una respuesta apropiada"""
    mensaje = mensaje.lower()

# Buscar producto específico

```

```

producto = buscar_producto(mensaje)
if producto:
    if 'precio' in mensaje or 'cuesta' in mensaje or 'valor' in mensaje:
        return f"El {producto} cuesta ${productos[producto]['precio']}. {productos[producto]['descripcion']}"
    return f"Tenemos {producto}. {productos[producto]['descripcion']}. ¿Te gustaría s"

# Buscar coincidencias en la base de conocimiento
for categoria, datos in knowledge_base.items():
    for patron in datos['patrones']:
        if re.search(patron, mensaje):
            return random.choice(datos['respuestas'])

# Si no se encuentra coincidencia
return "No estoy seguro de entender. ¿Podrías reformular tu pregunta?"

def main():
    """Función principal del chatbot"""
    print(f"{get_time_based_greeting()} Soy tu asistente virtual.")
    print("Puedo ayudarte con información sobre nuestros productos y precios.")
    print("Escribe 'salir' para terminar la conversación.\n")

    historial = []

    while True:
        try:
            # Obtener input del usuario
            entrada_usuario = input("Tú: ").strip()

            # Guardar en historial
            historial.append(("usuario", entrada_usuario))

            # Verificar si el usuario quiere salir
            if entrada_usuario.lower() == 'salir':
                print("Chatbot: ¡Gracias por tu visita! ¡Hasta pronto!")
                break

            # Procesar el mensaje y obtener respuesta
            respuesta = procesar_mensaje(entrada_usuario)
            print("Chatbot:", respuesta)

            # Guardar respuesta en historial
            historial.append(("bot", respuesta))

```

```

except KeyboardInterrupt:
    print("\nChatbot: Programa terminado por el usuario.")
    break
except Exception as e:
    print(f"Chatbot: Lo siento, ha ocurrido un error. Por favor, intenta de nuevo.")
    continue

if __name__ == "__main__":
    main()

```

Y la salida podría ser:

```

¡Buenas noches! Soy tu asistente virtual.
Puedo ayudarte con información sobre nuestros productos y precios.
Escribe 'salir' para terminar la conversación.

Tú: hola
Chatbot: ¡Bienvenido! ¿En qué puedo asistirte?
Tú: quiero saber precio de una tablet
Chatbot: El tablet cuesta $300. Tablet perfecta para entretenimiento
Tú: y la laptop?
Chatbot: Tenemos laptop. Laptop de última generación. ¿Te gustaría saber el precio?
Tú: si
Chatbot: No estoy seguro de entender. ¿Podrías reformular tu pregunta?
Tú: quiero precio de la laptop
Chatbot: El laptop cuesta $800. Laptop de última generación

Chatbot: Programa terminado por el usuario.

```

Al probar nuestro chatbot, notaremos las limitaciones de nuestra implementación, y las pocas posibilidades de conversación. Podríamos intentar otra estrategia, usando algunas capacidades de la librería **Spacy**:

```

# !pip install spacy
# !python -m spacy download es_core_news_md

import spacy
import random
from datetime import datetime
import re

# Cargar el modelo preentrenado de spaCy
nlp = spacy.load('es_core_news_md')

```

```

# Información ampliada sobre productos y promociones
productos = {
    'lavarropas': {
        'precio': 300,
        'descuento': 10,
        'características': ['6kg de capacidad', 'Eficiencia energética A++', 'Multiple programa'],
        'stock': 5
    },
    'aspiradora': {
        'precio': 200,
        'descuento': 5,
        'características': ['Sin bolsa', 'Filtro HEPA', 'Potencia 2000W'],
        'stock': 8
    },
    'heladera': {
        'precio': 1000,
        'descuento': 20,
        'características': ['No frost', '500L de capacidad', 'Dispensador de agua'],
        'stock': 3
    },
    'microondas': {
        'precio': 150,
        'descuento': 15,
        'características': ['30L de capacidad', 'Grill integrado', 'Panel digital'],
        'stock': 10
    },
    'licuadora': {
        'precio': 50,
        'descuento': 5,
        'características': ['Vaso de vidrio', '5 velocidades', '600W de potencia'],
        'stock': 15
    },
    'cafetera': {
        'precio': 40,
        'descuento': 5,
        'características': ['Capacidad 1.5L', 'Sistema antigoteo', 'Filtro permanente'],
        'stock': 12
    }
}

# Promociones con fechas de validez
promociones = [

```

```

{
    "mensaje": "¡Este fin de semana todos los productos con 15% de descuento!",
    "validez": "hasta el domingo",
    "descuento": 15,
    "productos_aplicables": "todos"
},
{
    "mensaje": "Por la compra de una heladera, lleva una licuadora con 50% de descuento",
    "validez": "próximos 30 días",
    "descuento": 50,
    "productos_aplicables": ["heladera", "licuadora"]
},
{
    "mensaje": "Compra una lavadora y secadora en combo y ahorra $100",
    "validez": "stock disponible",
    "descuento_fijo": 100,
    "productos_aplicables": ["lavarropas", "secadora"]
}
]

# Palabras clave para diferentes intenciones
intenciones = {
    'precio': ['precio', 'costo', 'vale', 'valor', 'cuánto'],
    'caracteristicas': ['características', 'especificaciones', 'detalles', 'funciones'],
    'stock': ['stock', 'disponible', 'hay', 'tienen'],
    'promocion': ['promoción', 'descuento', 'oferta', 'rebaja'],
    'comparacion': ['mejor', 'diferencia', 'comparar', 'versus', 'vs']
}

def detectar_intencion(message):
    """Detecta la intención principal del mensaje del usuario"""
    message = message.lower()
    intenciones_detectadas = []

    for intencion, palabras_clave in intenciones.items():
        if any(palabra in message for palabra in palabras_clave):
            intenciones_detectadas.append(intencion)

    return intenciones_detectadas if intenciones_detectadas else ['general']

def calcular_precio_final(producto):
    """Calcula el precio final con descuento"""
    precio_base = productos[producto]['precio']

```

```

descuento = productos[producto]['descuento']
precio_final = precio_base * (1 - descuento/100)
return precio_final

def buscar_producto(message):
    """Busca productos mencionados en el mensaje usando lematización"""
    doc = nlp(message.lower())
    lemas = [token.lemma_ for token in doc]
    productos_encontrados = []

    for producto in productos:
        if producto in lemas:
            productos_encontrados.append(producto)

    return productos_encontrados

def generar_respuesta_producto(producto, intenciones):
    """Genera una respuesta detallada según el producto y las intenciones detectadas"""
    respuesta = []
    datos = productos[producto]

    if 'precio' in intenciones:
        precio_final = calcular_precio_final(producto)
        respuesta.append(f"El {producto} tiene un precio de ${datos['precio']}, con un {d}
f"Precio final: ${precio_final:.2f}")

    if 'caracteristicas' in intenciones:
        respuesta.append(f"Características principales: {', '.join(datos['características'])}")

    if 'stock' in intenciones:
        respuesta.append(f"Actualmente tenemos {datos['stock']} unidades disponibles")

    if 'general' in intenciones:
        precio_final = calcular_precio_final(producto)
        respuesta.append(f"El {producto} está disponible por ${precio_final:.2f} (incluye {
f"Tenemos {datos['stock']} unidades en stock.")

    return " ".join(respuesta)

def comparar_productos(productos_lista):
    """Compara dos o más productos"""
    if len(productos_lista) < 2:
        return "Por favor, menciona al menos dos productos para comparar."

```

```

comparacion = "Comparación de productos:\n\n"
for producto in productos_lista:
    datos = productos[producto]
    precio_final = calcular_precio_final(producto)
    comparacion += f"{producto.capitalize()}\n"
    comparacion += f"- Precio final: ${precio_final:.2f}\n"
    comparacion += f"- Características: {', '.join(datos['características'])}\n"
    comparacion += f"- Stock disponible: {datos['stock']}\n\n"

return comparacion

def process_message(message):
    """Procesa el mensaje del usuario y genera una respuesta apropiada"""
    # Detectar intenciones
    intenciones_detectadas = detectar_intencion(message)

    # Buscar productos mencionados
    productos_encontrados = buscar_producto(message)

    # Si no se encontraron productos
    if not productos_encontrados:
        if 'promocion' in intenciones_detectadas:
            promo = random.choice(promociones)
            return f"¡Tenemos esta promoción especial: {promo['mensaje']} ({promo['valido']})"
        return "No he identificado ningún producto específico. ¿Puedes ser más específico?"

    # Si se encontró más de un producto y se detecta intención de comparación
    if len(productos_encontrados) > 1 and ('comparacion' in intenciones_detectadas):
        return comparar_productos(productos_encontrados)

    # Generar respuesta para cada producto encontrado
    respuestas = []
    for producto in productos_encontrados:
        respuesta = generar_respuesta_producto(producto, intenciones_detectadas)
        respuestas.append(respuesta)

    return "\n".join(respuestas)

def main():
    """Función principal para ejecutar el chatbot"""
    print("¡Hola! Soy el chatbot de ElectroMax. Puedo ayudarte con:")
    print("- Información de productos y precios")

```

```

print("- Características y especificaciones")
print("- Disponibilidad y stock")
print("- Promociones vigentes")
print("- Comparación entre productos")
print("\nEscribe 'salir' para terminar la conversación.")

while True:
    try:
        user_input = input("\nTú: ")
        if user_input.lower() == 'salir':
            print("Chatbot: ¡Gracias por visitar ElectroMax! ¡Hasta pronto!")
            break

        bot_response = process_message(user_input)
        print("\nChatbot:", bot_response)

    except KeyboardInterrupt:
        print("\nChatbot: Sesión terminada por el usuario. ¡Hasta pronto!")
        break
    except Exception as e:
        print(f"\nChatbot: Lo siento, ha ocurrido un error. Por favor, intenta de nuevo.")
        continue

if __name__ == "__main__":
    main()

```

Y la conversación podría ser:

¡Hola! Soy el chatbot de ElectroMax. Puedo ayudarte con:

- Información de productos y precios
- Características y especificaciones
- Disponibilidad y stock
- Promociones vigentes
- Comparación entre productos

Escribe 'salir' para terminar la conversación.

Tú: ¿Qué características tienen el microondas y la aspiradora?

Chatbot: Características principales: Sin bolsa, Filtro HEPA, Potencia 2000W Actualmente  
Características principales: 30L de capacidad, Grill integrado, Panel digital Actualmente

Tú: ¿Cuál es el precio final del microondas con descuento?

Chatbot: El microondas tiene un precio de \$150, con un 15% de descuento. Precio final

Tú: ¿Qué funciones tiene el microondas?

Chatbot: Características principales: 30L de capacidad, Grill integrado, Panel digital

Tú: ¿Tienen heladera en stock?

Chatbot: Actualmente tenemos 3 unidades disponibles

Tú: ¿Tienen ofertas este fin de semana?

Chatbot: ¡Tenemos esta promoción especial: Por la compra de una heladera, lleva una licuadora gratis!

Tú: ¿Qué diferencia hay entre el microondas y la cafetera?

Chatbot: Actualmente tenemos 10 unidades disponibles

Tú: ¿Qué características tiene la heladera y cuánto cuesta?

Chatbot: La heladera tiene un precio de \$1000, con un 20% de descuento. Precio final: \$800.00

Tú: ¿Me podés decir los precios de la licuadora y la cafetera?

Chatbot: La licuadora tiene un precio de \$50, con un 5% de descuento. Precio final: \$47.50

Tú: ¿Hay stock de heladera y lavarropas?

Chatbot: Actualmente tenemos 3 unidades disponibles

Tú: ¿Cuáles son los electrodomésticos más vendidos?

Chatbot: No he identificado ningún producto específico. ¿Puedes ser más específico sobre qué tipo de electrodoméstico te interesa?

Tú: ¿Tienen servicio de entrega?

Chatbot: No he identificado ningún producto específico. ¿Puedes ser más específico sobre qué tipo de servicio de entrega te interesa?

Tú: Hola, estoy buscando una heladera nueva, ¿qué me recomendás?

Chatbot: La heladera está disponible por \$800.00 (incluye 20% de descuento). Tenemos 3 unidades disponibles.

Chatbot: Sesión terminada por el usuario. ¡Hasta pronto!

La primera versión del chatbot que vimos anteriormente, utilizaba un enfoque basado en expresiones regulares. Las expresiones regulares son patrones que permiten buscar coincidencias dentro de cadenas de texto. Esto significa que son muy potentes para buscar patrones específicos, pero tienen algunas limitaciones:

1. **Literalidad:** Las expresiones regulares buscan coincidencias exactas o patrones muy específicos. Si un usuario no escribe exactamente lo que la expresión regular está configurada para encontrar, es posible que el chatbot no reconozca la intención.
2. **Flexibilidad:** Las expresiones regulares no entienden el lenguaje natural; solo siguen las reglas específicas del patrón. Esto significa que no manejan bien las variaciones lingüísticas como las conjugaciones de verbos, los plurales, las palabras compuestas, etc.
3. **Mantenimiento:** A medida que se agregan más funcionalidades y se cubren más variaciones de lenguaje natural, las expresiones regulares se pueden volver muy complejas y difíciles de mantener.

El chatbot mejorado con spaCy ofrece varias mejoras:

1. **Lematización:** Al reducir las palabras a su raíz o forma base (lemas), el bot puede reconocer "aspirar", "aspirando", "aspirado" como relacionados con la palabra "aspiradora". Esto le permite manejar variaciones lingüísticas mucho más naturalmente que las expresiones regulares.
2. **Comprensión de lenguaje natural:** spaCy ofrece un análisis más sofisticado del texto que incluye la identificación de entidades nombradas (NER), categorización de palabras (part-of-speech tagging), y dependencias sintácticas, lo que significa que el chatbot puede tener una comprensión más rica del texto y la intención del usuario.
3. **Escalabilidad y mantenimiento:** Al depender del procesamiento del lenguaje natural (NLP), es más fácil escalar y mantener el chatbot porque se basa en modelos lingüísticos en lugar de una lista siempre creciente de patrones de expresiones regulares.
4. **Contextualización:** Con spaCy, y en general con cualquier librería de NLP, hay un potencial para utilizar el contexto del diálogo anterior para informar respuestas futuras, algo que las expresiones regulares no pueden hacer directamente. **(10)**

## Chatbots basados en IA

Los chatbots basados en IA, específicamente los que utilizan Modelos de Lenguaje a Gran Escala (LLMs, por sus siglas en inglés), son sistemas avanzados de procesamiento

de lenguaje natural que se basan en redes neuronales profundas. Estos son algunos de los puntos clave que definen cómo funcionan y se aplican:

#### **Tecnología de Aprendizaje Profundo:**

- Los LLMs utilizan arquitecturas de aprendizaje profundo como las redes neuronales recurrentes (RNN), las redes neuronales convolucionales (CNN) y, más recientemente, las redes de atención y transformadores.
- Estas redes están diseñadas para procesar secuencias de palabras y entender la estructura y el contexto del lenguaje.

#### **Entrenamiento con Grandes Conjuntos de Datos:**

- Se entrena con enormes corpus de texto que pueden abarcar desde enciclopedias y literatura hasta diálogos y foros en línea.
- Aprenden a predecir la siguiente palabra en una secuencia, lo que les ayuda a generar respuestas coherentes y contextualmente relevantes.

#### **Capacidad de Generalización:**

- A diferencia de los sistemas basados en reglas, los LLMs pueden generalizar a partir de los datos de entrenamiento y manejar preguntas o comandos que nunca han visto antes.
- Esto los hace mucho más flexibles y capaces de mantener conversaciones más fluidas y naturales.

#### **Comprensión y Generación de Lenguaje Natural:**

- Los LLMs utilizan procesamiento de lenguaje natural (NLP) para analizar y entender las consultas de los usuarios.
- Pueden manejar ambigüedades, errores comunes en el lenguaje humano y entender diferentes matices, como el sarcasmo o el humor, dependiendo de su entrenamiento y sofisticación.[\(11\)](#)

#### **Aprendizaje Continuo:**

- Algunos LLMs pueden mejorar con el tiempo a través del aprendizaje continuo, adaptándose a los patrones de habla de los usuarios y refinando sus modelos internos para ofrecer respuestas más precisas.

#### **Aplicaciones de LLMs:**

- Asistencia virtual: Pueden realizar tareas como establecer recordatorios, responder preguntas y ayudar con las tareas diarias.
- Atención al cliente: Proporcionan respuestas automáticas a preguntas frecuentes, resolución de problemas y asistencia en transacciones en línea.

- Terapia y asesoramiento: Algunos chatbots están diseñados para ofrecer apoyo emocional y consejos, aunque con ciertas limitaciones y siempre bajo supervisión humana.

### **Desafíos y Consideraciones:**

- Sesgo y ética: Los LLMs pueden perpetuar sesgos presentes en los datos de entrenamiento, lo que requiere cuidadosa atención y ajuste.
- Privacidad y seguridad: Manejan información sensible del usuario, lo que exige protocolos robustos de seguridad y privacidad.
- Limitaciones contextuales: A pesar de su sofisticación, pueden malinterpretar el contexto o la intención, llevando a respuestas inadecuadas o irrelevantes.[\(12\)](#)

## **Capacidades emergentes de los LLM**

Las **capacidades emergentes de un LLM (Large Language Model)** son habilidades que no están explícitamente programadas pero que aparecen cuando el modelo alcanza cierto tamaño o se entrena con suficiente diversidad de datos. A continuación, se describen algunas de ellas, junto con una explicación de por qué ocurren:

### **Razonamiento aritmético y lógico básico**

A medida que el modelo crece, puede resolver problemas matemáticos sencillos o inferencias lógicas, aunque no haya sido diseñado para ello. Esto ocurre porque el entrenamiento sobre grandes cantidades de texto incluye ejemplos de razonamiento numérico y verbal, y el modelo aprende patrones estadísticos que generalizan a nuevas preguntas. La arquitectura autoatencional también ayuda a seguir relaciones entre tokens que representan pasos lógicos.

### **Traducción entre lenguas**

Incluso sin supervisión directa, un LLM puede traducir textos entre idiomas si ha visto suficiente texto multilingüe. Esto se debe a que aprende una representación semántica compartida: identifica que ciertas estructuras tienen significados equivalentes, aunque se expresen con diferentes palabras. Esta capacidad emerge por el alineamiento estadístico de significados similares en múltiples lenguas

### **Seguimiento de instrucciones complejas**

Los LLM grandes muestran una capacidad sorprendente para seguir instrucciones dadas en lenguaje natural, incluso si son detalladas o de múltiples pasos. Esto emerge porque durante el entrenamiento, el modelo ve muchos ejemplos de diálogos, manuales y patrones de pregunta-respuesta. El tamaño permite que generalice estas estructuras y represente contextos amplios de manera coherente.

### **Comprensión contextual de metas y roles**

En interacciones complejas, los LLM pueden adoptar roles (como médico, abogado o tutor) y responder en consecuencia. Esto ocurre porque el modelo ha visto en el

entrenamiento numerosos ejemplos de interacciones donde se especifica un rol, y aprende a asociar estilos de lenguaje, objetivos y convenciones de cada uno. La capacidad de mantener coherencia de rol es emergente y depende del tamaño del modelo y de la riqueza del corpus.

Google Colab

🔗 <https://colab.research.google.com/drive/1UQqSRUI4mflfwuQfcogpHueSftFKIXsY?usp=sharing>



## 2. Prompts

En el contexto de la Procesamiento del Lenguaje Natural (NLP) y los Modelos de Lenguaje a Gran Escala (LLM, por sus siglas en inglés), un "prompt" se refiere a una instrucción o conjunto de instrucciones dadas a un modelo de lenguaje para que realice una tarea específica. Los prompts actúan como una interfaz entre el usuario y el modelo, indicándole al modelo qué se espera de él.

Por ejemplo, en la generación de texto, un prompt podría ser una frase o un párrafo que el modelo usa como punto de partida para generar texto continuo. En el caso de las preguntas y respuestas, el prompt sería la pregunta formulada al modelo, a la que se espera que responda.

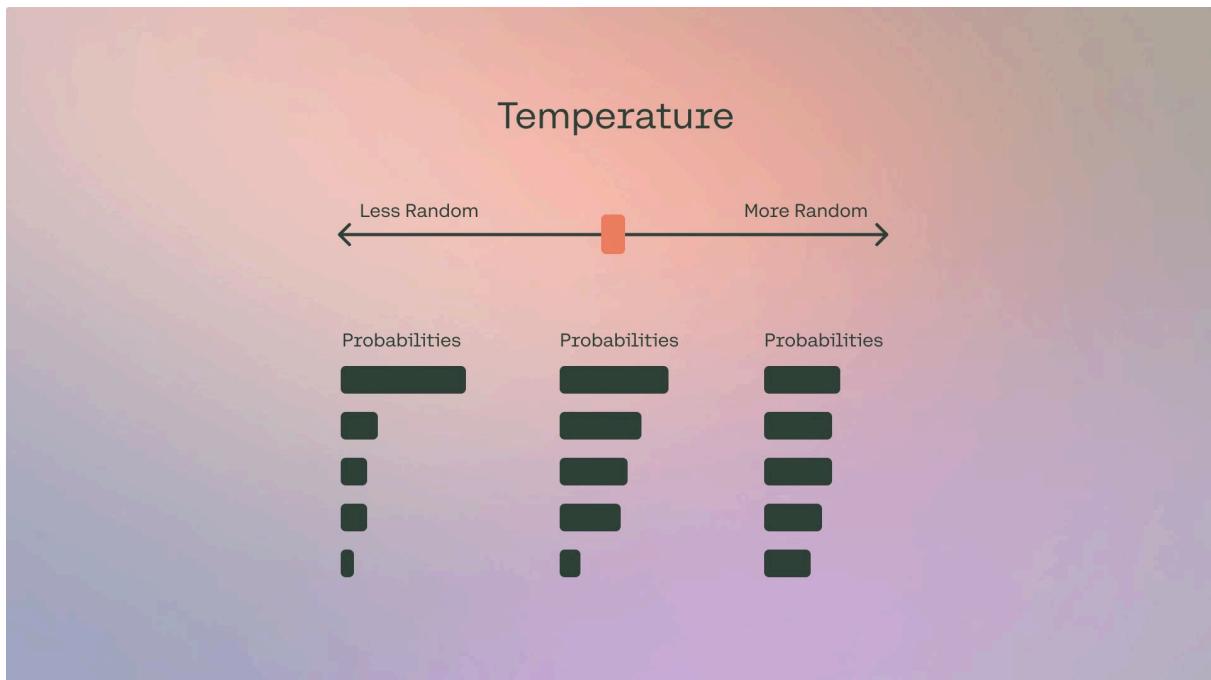
Los prompts son fundamentales en la interacción con los LLMs porque los modelos están diseñados para responder o seguir las instrucciones contenidas en los prompts. La calidad y claridad de un prompt pueden influir significativamente en la calidad de la salida del modelo. Por esta razón, el diseño de prompts efectivos es una habilidad importante en el campo de NLP, ya que permite a los usuarios aprovechar al máximo las capacidades de los LLMs para diversas aplicaciones, desde la automatización de tareas hasta la asistencia creativa y la generación de contenido.

### Configuración de los grandes modelos de lenguaje (LLM Settings)

Al trabajar con *prompts*, interactuamos con el LLM a través de una API o directamente. Podemos configurar algunos parámetros para obtener diferentes resultados a partir de nuestros *prompts*.

**Temperatura:** En resumen, cuanto más baja sea la temperatura, más deterministas serán los resultados en el sentido de que siempre se elige el siguiente token más probable. Aumentar la temperatura podría conducir a más aleatoriedad, lo que fomenta resultados más diversos o creativos. Esencialmente estás aumentando los pesos de los otros tokens posibles. En términos de aplicación, es posible que quieras usar un valor de temperatura más bajo para tareas como preguntas y respuestas basadas en hechos

para fomentar respuestas más factuales y concisas. Para la generación de poemas u otras tareas creativas, podría ser beneficioso aumentar el valor de la temperatura.



<https://txt.cohere.com/llm-parameters-best-outputs-language-ai/>

Consideremos la frase "El cielo es". Cuando lees eso, probablemente piensas que la próxima palabra será "azul" o "el límite". También es improbable que la próxima palabra sea "agua" o "pegajoso". Inconscientemente hemos creado predicciones para palabras que pueden seguir y hemos determinado que "azul" es más probable mientras que "pegajoso" es menos probable.

Y eso es esencialmente lo que hace un modelo generativo de texto. Tiene probabilidades para todas las diferentes palabras que podrían seguir y luego selecciona la próxima palabra para producir. La configuración de Temperatura le indica cuáles de estas palabras puede usar.

Una Temperatura de 0 hace que el modelo sea determinista. Limita al modelo a usar la palabra con la mayor probabilidad. Podemos ejecutarlo una y otra vez y obtener el mismo resultado. A medida que aumentamos la Temperatura, el límite se suaviza, permitiéndole usar palabras con probabilidades más bajas, y podría generar "pegajoso" si lo ejecutamos suficientes veces.

Cuando generamos texto desde un *prompt* "El cielo es" a una Temperatura media y baja, podemos ver resultados como por ejemplo:

El cielo es el límite

El cielo es azul

Y en la configuración más alta, entramos en el reino de la fantasía:

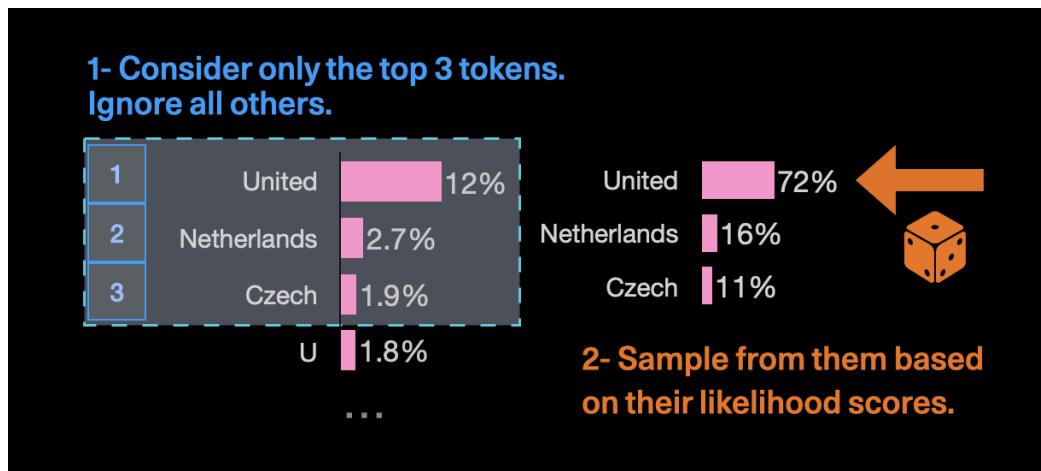
"El cielo está despejado, el agua tranquila, y es un camino inimaginablemente largo antes de que los delfines decidan renunciar a su búsqueda vertical".

### Top\_p y Top\_k:

Además de la Temperatura, Top-k y Top-p son las otras dos formas de elegir el token de salida.

Top-k le indica al modelo que elija el próximo token de los 'k' tokens principales en su lista, ordenados por probabilidad.

Consideremos la frase de entrada: "The name of that country is the" ("El nombre de ese país es el"). El próximo token podría ser "United", "Netherlands", "Czech", y así sucesivamente, con diferentes probabilidades. Puede haber docenas de salidas potenciales con probabilidades decrecientes, pero si establecemos k como 3, le estás diciendo al modelo que solo elija entre las 3 opciones principales.

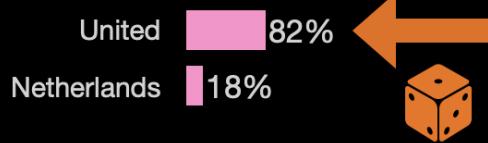
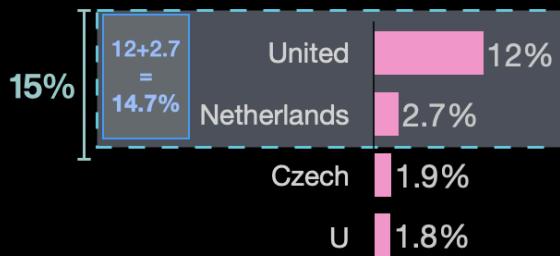


Entonces, si ejecutamos el mismo prompt varias veces, obtendremos "United" muy a menudo, y de vez en cuando "Netherlands" o "Czech", pero nada más.

Si establecemos k en 1, el modelo solo elegirá el token superior ("United", en este caso).

**Top-p** es similar, pero elige entre los tokens superiores basándose en la suma de sus probabilidades. Entonces, para el ejemplo anterior, si establecemos top\_p en 0.15, solo elegirá entre "United" y "Netherlands" ya que sus probabilidades suman el 14.7%.

**1- Consider only the top tokens whose likelihoods add up to 15%. Ignore all others.**



**2- Sample from them based on their likelihood scores.**

Top-p es más dinámico que top-k y se usa a menudo para excluir salidas con probabilidades más bajas. Entonces, si establecemos top\_p en 0.75, excluimos el 25% inferior de las salidas probables.

**Max Length (Máxima longitud):** Podemos gestionar el número de tokens que el modelo genera ajustando la 'máxima longitud'. Especificar una longitud máxima nos ayuda a prevenir respuestas largas o irrelevantes y a controlar costos.

#### Stop Sequences (Secuencias de Parada):

Una secuencia de parada es una cadena que le indica al modelo que deje de generar más contenido. Es otra forma de controlar cuán largo queremos el resultado.

Entonces, por ejemplo, si le damos al modelo el prompt "El cielo es" y escribo un punto (.) como secuencia de parada, el modelo deja de generar texto una vez que alcanza el final de la primera oración, incluso si el límite de tokens es mucho mayor.

Esto combina bien con prompts donde incluimos un par de ejemplos. Digamos que queremos generar texto siguiendo un cierto patrón, agregamos una cierta cadena a los ejemplos y luego usamos esa cadena como secuencia de parada.

Given a post, this program will generate relevant hashtags.

Post: Why are there no country songs about software engineering  
Hashtag: #softwareengineering

Post: Your soulmate is in the WeWork you decided not to go to  
Hashtag: #wework

Post: If she's talking to you once a day I'm sorry bro that's not flirting that standup  
Hashtag: #standup

Post: Going to unmute at the end of the Zoom meeting to say bye and realizing you were actually unmuted the whole call  
Hashtag:

**Parameters**

- Model**: large-20220425 (large)
- Number of Tokens**: 10
- Temperature**: 0.5
- Stop Sequences**: --
- top-k**: 0
- top-p**: 1
- Frequency penalty**: 0

**Frequency Penalty (Penalización de Frecuencia):** La 'penalización de frecuencia' aplica una penalización al siguiente token proporcional a cuántas veces ese token ya apareció en la respuesta y la indicación. Cuanto mayor sea la penalización de frecuencia, menos probable será que una palabra aparezca de nuevo. Esta configuración reduce la repetición de palabras en la respuesta del modelo al dar a los tokens que aparecen más una penalización mayor.

**Presence Penalty (Penalización de Presencia):** La 'penalización de presencia' también aplica una penalización a tokens repetidos pero, a diferencia de la penalización de frecuencia, la penalización es la misma para todos los tokens repetidos. Un token que aparece dos veces y un token que aparece 10 veces son penalizados igual. Esta configuración evita que el modelo repita frases demasiado a menudo en su respuesta. Si queremos que el modelo genere texto diverso o creativo, es posible que queramos usar una penalización de presencia más alta. O, si necesitamos que el modelo se mantenga enfocado, intenta usar una penalización de presencia más baja.

Similar a la temperatura y *top\_p*, la recomendación general es alterar la penalización de frecuencia o presencia, no ambas.

Antes de comenzar con algunos ejemplos básicos, tengamos en cuenta que nuestros resultados pueden variar dependiendo de la versión de LLM que usemos.

## Nociones básicas de prompts

Podemos lograr mucho con *prompts* simples, pero la calidad de los resultados depende de cuánta información le proporcionamos y de lo bien elaborada que esté. Un prompt

puede contener información como la instrucción o pregunta que estamos pasando al modelo, e incluir otros detalles como contexto, entradas o ejemplos. Podemos usar estos elementos para instruir mejor al modelo y, como resultado, obtener mejores resultados.

Comencemos repasando un ejemplo básico de un *prompt* simple:

*Prompt*

El cielo es

*Output:*

El cielo es una región de la atmósfera terrestre que se encuentra sobre nosotros y que generalmente se ve de color azul durante el día debido a la dispersión de la luz solar por las partículas en la atmósfera.

Como podemos ver, el modelo de lenguaje produce una continuación de cadenas que tiene sentido dado el contexto "El cielo es". La salida puede ser inesperada o lejana de la tarea que queremos lograr.

Este ejemplo básico también destaca la necesidad de proporcionar más contexto o instrucciones sobre lo que específicamente queremos lograr.

Intentemos mejorarlo un poco:

*Prompt:*

Completa con una sola frase: El cielo está

*Output:*

El cielo está despejado y azul en un día soleado de verano.

¿Es eso mejor? Bueno, le dijimos al modelo que completara la oración, así que el resultado parece mucho mejor ya que sigue exactamente lo que le dijiste que hiciera ("completa la oración"). Este enfoque de diseñar *prompts* óptimos para instruir al modelo para realizar una tarea, es lo que se conoce como *ingeniería de prompts*.

El ejemplo anterior es una ilustración básica de lo que es posible con los LLM hoy en día. Los LLM de hoy son capaces de realizar todo tipo de tareas avanzadas que van desde la resumir textos hasta el razonamiento matemático y la generación de código.

## Formateo de Prompts

Hemos probado un *prompt* muy simple anteriormente. Un *prompt* estándar tiene el siguiente formato:

<Pregunta>?

ó

<Instrucción>

Podemos formatear esto en un formato de preguntas y respuestas (QA), que es estándar en muchos conjuntos de datos de QA, de la siguiente manera:

Q: <Pregunta>?

A:

Cuando hacemos indicaciones como la anterior, también se le llama indicación zero-shot (cero disparos), es decir, estamos solicitando directamente al modelo una respuesta sin ningún ejemplo o demostración sobre la tarea que queremos que logre. Algunos modelos de lenguaje a gran escala tienen la capacidad de realizar indicaciones zero-shot, pero depende de la complejidad y conocimiento de la tarea en cuestión.

Dado el formato estándar anterior, una técnica popular y efectiva para hacer indicaciones se conoce como prompt few-shot (de pocos disparos), donde proporcionamos ejemplos (es decir, demostraciones). Podemos formatear los *prompts* few-shot de la siguiente manera:

<Question>?

<Answer>

<Question>?

<Answer>

<Question>?

<Answer>

<Question>?

La versión de formato QA se vería así:

Q: <Question>?

A: <Answer>

Q: <Question>?

A: <Answer>

Q: <Question>?

A: <Answer>

Q: <Question>?

A:

Tengamos en cuenta que no es necesario usar el formato QA. El formato del prompt depende de la tarea en cuestión. Por ejemplo, podemos realizar una tarea de clasificación simple y dar ejemplos que demuestren la tarea de la siguiente manera:

*Prompt:*

```
Texto: ¡Esto es malo!
Etiqueta: Positivo
Texto: ¡Vaya, esa película fue genial!
Etiqueta: Negativo
Texto: ¡Qué espectáculo tan horrible!
Etiqueta: Positivo
Texto: ¡Esto es increíble!
Etiqueta:
```

Salida:

```
Negativo
```



Los *prompts few-shot* permiten el aprendizaje en contexto (in-context learning), que es la capacidad de los modelos de lenguaje para aprender tareas dadas unas pocas demostraciones.

## Elementos de los Prompts

A medida que cubrimos más y más ejemplos y aplicaciones con la ingeniería de prompts, notaremos que ciertos elementos constituyen un prompt.

Un prompt contiene cualquiera de los siguientes elementos:

**Instrucción** - una tarea específica o instrucción que quieres que el modelo realice.

**Contexto** - información externa o contexto adicional que puede dirigir al modelo hacia respuestas mejores.

**Datos de Entrada** - la entrada o pregunta para la cual estamos interesados en encontrar una respuesta.

**Indicador de Salida** - el tipo o formato de la salida.

No necesitas todos los cuatro elementos para una indicación y el formato depende de la tarea en cuestión. Tocaremos ejemplos más concretos en guías próximas.

## Diseño de Prompts

Aquí hay algunos consejos a tener en cuenta mientras diseñas tus indicaciones:

### Comenzar Simple

A medida que comiences con el diseño de indicaciones, debes tener en cuenta que realmente es un proceso iterativo que requiere mucha experimentación para obtener resultados óptimos. Usar un entorno de pruebas simple de OpenAI o Cohere es un buen punto de partida.

Podemos comenzar con *prompts* simples y seguir agregando más elementos y contexto a medida que apuntamos a obtener mejores resultados. Cuando tengamos una tarea grande que involucre muchas sub-tareas diferentes, podemos intentar desglosar la tarea en sub-tareas más simples y seguir construyendo a medida que obtenemos mejores resultados. Esto evita añadir demasiada complejidad al proceso de diseño de *prompt* al principio.

### Instrucciones

Podemos diseñar *prompts* efectivos para varias tareas simples usando comandos para instruir al modelo sobre lo que quieres lograr, como "Escribe", "Clasifica", "Resume", "Traduce", "Ordena", etc.

Tengamos en cuenta que también necesitamos experimentar mucho para ver qué funciona mejor. Podemos probar diferentes instrucciones con diferentes palabras clave, contextos y datos y ver qué funciona mejor para cada caso de uso y tarea en particular. Por lo general, cuanto más específico y relevante sea el contexto para la tarea que estamos tratando de realizar, mejor. Tocaremos la importancia del muestreo y la adición de más contexto más adelante.

Algunas recomendaciones indican conveniente colocar las instrucciones al principio del *prompt*. Otra recomendación es usar algún separador claro como "###" para separar la instrucción y el contexto.

Por ejemplo:

*Prompt:*

```
### Instrucción ###
Traduce el texto a continuación al español.
Texto: "hello!"
```

Salida:

```
¡Hola!
```

### Especificidad

Debemos ser específicos sobre la instrucción y la tarea que queremos que el modelo realice. Cuanto más descriptiva y detallada sea la indicación, mejores serán los resultados. Esto es particularmente importante cuando tenemos un resultado deseado o un estilo de generación que estás buscando. No hay tokens o palabras clave específicas que lleven a mejores resultados. Es más importante tener un buen formato y una indicación descriptiva. De hecho, proporcionar ejemplos en el *prompts* es muy efectivo para obtener la salida deseada en formatos específicos.

Al diseñar *prompts*, también debemos tener en cuenta la longitud del *prompt*, ya que hay limitaciones respecto a cuán larga puede ser (cantidad de tokens). Incluir demasiados detalles innecesarios no es necesariamente un buen enfoque. Los detalles deben ser relevantes y contribuir a la tarea en cuestión. Esto es algo con lo que debemos experimentar mucho. Es importante la experimentación e iteración para optimizar los *prompts* para nuestras aplicaciones.

Como ejemplo, intentemos un *prompt* simple para extraer información específica de un texto.

*Prompt:*

Extraer cantidades de conjuntos de datos climáticos realizados en el siguiente texto.

Formato deseado: <cantidad\_de\_lugares>,<ambito\_de\_examen><crlf>

Entrada:

En el análisis se examinaron las temperaturas medias diarias y las olas de calor e incluyó datos de 175 países, 154 estados o provincias y 920 grandes ciudades. El informe se conoció a 20 días de que se celebre la cumbre de cambio climático en Dubai (COP28) en donde los líderes mundiales debatirán los avances en la lucha contra el calentamiento global.

Salida:

175,paises

154,estados/provincias

920,ciudades

En el ejemplo, usamos los caracteres `<crlf>` (Retorno de carro y nueva línea), para generar una salida en diferentes renglones.

## Evitar la imprecisión

Dado los consejos anteriores sobre ser detallado y mejorar el formato, es fácil caer en la trampa de querer ser demasiado astuto con las indicaciones y potencialmente crear descripciones imprecisas. A menudo es mejor ser específico y directo. La analogía aquí es muy similar a la comunicación efectiva: cuanto más directo, más eficazmente se transmite el mensaje.

Por ejemplo, podríamos estar interesados en aprender el concepto de ingeniería de *prompts*. Podemos intentar algo como:

Explicar el concepto de prompt engineering. Realizar una explicación breve, solo unas pocas oraciones, y no seas demasiado descriptivo.

No está claro en el *prompt* anterior cuántas frases usar y qué estilo. Aún podríamos obtener respuestas algo buenas con los *prompts* anteriores, pero un mejor *prompt* sería uno que sea más específico, conciso y al grano. Algo como:

Utilice 2 o 3 oraciones para explicar el concepto de prompt engineering a un estudiante de secundaria.

## ¿Hacer o no hacer?

Otro consejo común al diseñar *prompts*, es evitar decir lo que no se debe hacer, sino decir lo que se debe hacer en su lugar. Esto fomenta más especificidad y se centra en los detalles que conducen a buenas respuestas del modelo.

Aquí hay un ejemplo de un chatbot de recomendación de películas que falla exactamente en lo que no quiero que haga debido a cómo escribimos la instrucción, enfocándonos en lo que no debe hacer.

*Prompt:*

El siguiente es un agente que recomienda películas a un cliente. NO CONSULTAR P OR INTERESES. NO PIDA INFORMACIÓN PERSONAL.

Cliente: Por favor recomiende una película basada en mis intereses.

Agente:

Salida:

Claro, puedo recomendarte una película según tus intereses. ¿Qué tipo de película te gustaría ver? ¿Prefieres acción, comedia, romance o algo más?

Acá hay un prompt mejorado:

*Prompt:*

El siguiente es un agente que recomienda películas a un cliente. El agente es responsable de recomendar una película de las principales tendencias mundiales. Debe abstenerse de preguntar a los usuarios sus preferencias y evitar solicitar información personal. Si el agente no tiene una película para recomendar, debería responder "Lo siento, no pude encontrar una película para recomendar hoy".

Cliente: recomiende una película según mis intereses.

Agente:

Salida:

Lo siento, no tengo ninguna información sobre tus intereses. Sin embargo, aquí hay una lista de las películas de mayor tendencia mundial en este momento: [lista de películas]. ¡Espero que encuentres algo que te guste!

Algunos de los ejemplos anteriores fueron tomados del artículo "[Mejores prácticas para la ingeniería de indicaciones con la API de OpenAI](#)".

## Ejemplos de Prompts

La sección proporciona ejemplos sobre cómo utilizar *prompts* para lograr diferentes tareas e introduce conceptos clave en el camino. A menudo, la mejor manera de aprender conceptos es a través de ejemplos. Los ejemplos a continuación ilustran cómo se pueden usar indicaciones bien elaboradas para realizar diferentes tipos de tareas:

### Resumen de textos

Una de las tareas estándar en la generación de lenguaje natural es el resumen de texto. La síntesis de texto puede incluir muchos enfoques y dominios diferentes. De hecho, una de las aplicaciones más prometedoras de los modelos de lenguaje es la capacidad de resumir artículos y conceptos en resúmenes rápidos y fáciles de leer. Probemos una tarea básica de resumen utilizando indicaciones.

Digamos que estás interesado en aprender sobre antibióticos, podrías intentar una indicación como esta:

*Prompt:*

Explique los antibióticos.

A:

Salida:

Los antibióticos son un tipo de medicamento que se usa para tratar infecciones bacterianas. Actúan matando las bacterias o impidiendo que se reproduzcan, lo que permite que el sistema inmunológico del cuerpo combata la infección. Los antibióticos generalmente se toman por vía oral en forma de píldoras, cápsulas o soluciones líquidas o, a veces, se administran por vía intravenosa. No son eficaces contra las infecciones virales y su uso inadecuado puede provocar resistencia a los antibióticos.

El "A:" es un formato de indicación explícito que se utiliza en las respuestas a preguntas. Aquí se utilizó para decirle al modelo que se espera una respuesta a continuación. En este ejemplo, no está claro cómo esto es útil frente a no usarlo, pero lo dejaremos así para ejemplos posteriores. Supongamos simplemente que esta es demasiada información y quieras resumirla aún más. De hecho, podemos instruir al modelo para que resuma en una sola oración de la siguiente manera:

*Prompt:*

Los antibióticos son un tipo de medicamento que se usa para tratar infecciones bacterianas. Actúan matando las bacterias o impidiendo que se reproduzcan, lo que permite que el sistema inmunológico del cuerpo combata la infección. Los antibióticos generalmente se toman por vía oral en forma de píldoras, cápsulas o soluciones líquidas o, a veces, se administran por vía intravenosa. No son eficaces contra las infecciones virales y su uso inadecuado puede provocar resistencia a los antibióticos.

Resume lo anterior en una sola oración:

*Salida:*

Los antibióticos son medicamentos que se usan para tratar infecciones bacterianas, ya sea matando las bacterias o impidiendo que se reproduzcan, pero no son eficaces contra los virus y el uso excesivo puede provocar resistencia a los antibióticos.

Sin prestar demasiada atención a la precisión de la salida anterior, el modelo intentó resumir el párrafo en una sola frase.

## Extracción de información (Information Extraction)

Mientras que los modelos de lenguaje están entrenados para realizar la generación de lenguaje natural y tareas relacionadas, también son muy capaces de realizar clasificación y una variedad de otras tareas de procesamiento de lenguaje natural (NLP).

Aquí hay un ejemplo de una indicación que extrae información de un párrafo dado.

*Prompt:*

Las declaraciones de contribución de los autores y los agradecimientos en los artículos de investigación deben indicar clara y específicamente si los autores utilizaron tecnologías de inteligencia artificial como ChatGPT en la preparación de su manuscrito y análisis, y en qué medida. También deben indicar qué LLM se utilizaron. Esto alertará a los editores y revisores para que examinen los manuscritos con más atención en busca de posibles sesgos, inexactitudes y acreditación inadecuada de las fuentes. Del mismo modo, las revistas científicas deben ser transparentes sobre el uso de los LLM, por ejemplo, al seleccionar los manuscritos enviados.

Mencione el producto basado en modelos de lenguaje grande mencionado en el párrafo anterior:

Salida:

El producto basado en modelos de lenguaje grande mencionado en el párrafo anterior es ChatGPT.

Hay muchas maneras en las que podemos mejorar los resultados anteriores, pero esto ya es muy útil. A estas alturas nos damos cuenta que podemos pedirle al modelo que realice diferentes tareas simplemente instruyéndolo sobre qué hacer. Esa es una capacidad poderosa que los desarrolladores de productos de inteligencia artificial pueden utilizar para construir productos y experiencias poderosas.

Fuente del párrafo: [ChatGPT: cinco prioridades para la investigación \(se abre en una nueva pestaña\)](#).

### Extracción de información estructurada:

Supongamos que queremos extraer información de un texto, pero luego queremos aplicar herramientas de programación para procesar la salida de forma automática. Podríamos trabajar el prompt de manera tal que seamos mucho más específicos en cuanto al formato de salida. Veamos el siguiente ejemplo, para la extracción de tríadas:

Actúa como un experto en extracción de conocimiento y análisis semántico. Tu tarea es

\*\*Instrucciones detalladas:\*\*

1. \*\*Identifica las entidades principales:\*\* Estas serán los Sujetos y Objetos de tus tríadas.
2. \*\*Identifica las relaciones:\*\* Estas son las acciones, estados o propiedades que conectan las entidades.
3. \*\*Formato de la tríada (Conceptual):\*\* Cada tríada debe seguir la estructura conceptual: `("ENTIDAD\_SUJETO")-[RELACION]→("ENTIDAD\_OBJETO")`
4. \*\*Precisión y Concisión:\*\*
  - \* Sé lo más específico posible al identificar cada componente de la tríada.
  - \* Evita información superflua. La relación debe ser el núcleo del vínculo.
  - \* Si una oración contiene múltiples ideas conectadas, puedes extraer múltiples tríadas.
5. \*\*Contexto:\*\* Considera el contexto de la frase para desambiguar y extraer la relación adecuada.
6. \*\*Formato de Salida Obligatorio:\*\* Devuelve cada tríada extraída en una \*\*nueva línea\*\*:  
`("ENTIDAD\_SUJETO")-[RELACION]→("ENTIDAD\_OBJETO")`
  - \* `ENTIDAD\_SUJETO` y `ENTIDAD\_OBJETO` deben ser los nombres literales de las entidades.
  - \* `TIPO\_RELACION` debe ser una representación concisa y descriptiva de la relación.
  - \* Si no se encuentran tríadas claras o relevantes en el texto, devuelve la cadena: `No se encontraron tríadas relevantes`.

**\*\*Ejemplo de formato de salida para una tríada:\*\***

Si el texto fuera "Ana es amiga de Pedro y Pedro trabaja en Consultora X.", la salida es:

`("Ana")-[:ES\_AMIGA\_DE]→("Pedro")`

`("Pedro")-[:TRABAJA\_EN]→("Consultora X")`

**\*\*Texto a procesar:\*\***

"El aire en la mansión Blackwood se podía cortar con un cuchillo tras el inesperado anu-

El detective Harding, llamado para investigar no un crimen aún, sino las veladas amenazas.

La respuesta del LLM, podría ser la siguiente:

("Lord Alistair")-[:HIZO\_UN\_ANUNCIO\_INESPERADO]→("aire en la mansión Blackwood")  
("Clara")-[:ES\_SOBRINA\_DE]→("Lord Alistair")  
("Clara")-[:OBSERVABA\_REACCION\_DE]→("todos")  
("Clara")-[:OBSERVABA\_ESPECIALMENTE\_REACCION\_DE]→("Julian")  
("Clara")-[:SE\_RUMOREABA\_COMPARTIA\_ALGO\_MAS\_QUE\_LAZO\_FAMILIAR\_CON]→("")  
("Clara")-[:SE\_RUMOREABA\_COMPARTIA\_INVERSION\_FALLIDA\_CON]→("Julian")  
("Inversión fallida")-[:FUE\_HEREDADA\_DE]→("abuelo de ambos (Clara y Julian)")  
("Beatrice")-[:ES\_MATRIARCA\_DE\_LA\_FAMILIA]→("Blackwood")  
("Beatrice")-[:ES\_HERMANA\_DE]→("Lord Alistair")  
("Beatrice")-[:SIEMPRE\_HABIA\_DESCONFIADO\_DE\_INTENCIONES\_DE]→("Julian")  
("Julian")-[:TENIA\_INTENCIONES\_HACIA]→("fortuna familiar")  
("Beatrice")-[:ESPERABA\_ADMINISTRAR]→("fortuna familiar")  
("Beatrice")-[:ESPERABA\_ADMINISTRAR\_A\_TRAVES\_DE]→("Edward")  
("Edward")-[:ES\_HIJO\_DE]→("Beatrice")  
("Edward")-[:MANTENIA\_SECRETA\_CORRESPONDENCIA\_CON]→("Seraphina")  
("Seraphina")-[:ES FIGURA\_MISTERIOSA\_QUE\_NADIE ADMITE CONOCER\_EN]→("familia")  
("Nombre de Seraphina")-[:HABIA\_SIDO\_SUSURRADO\_EN\_CONEXION\_CON]→("negocios")  
("Coronel Montgomery")-[:FUE\_DIFUNTO\_PRIMER\_ESPOSO\_DE]→("Beatrice")  
("Detective Harding")-[:FUE\_LLAMADO\_PARA\_INVESTIGAR]→("veladas amenazas")  
("Lord Alistair")-[:DECIA\_RECIBIR]→("veladas amenazas")  
("Lealtad de Thomas")-[:HACIA]→("Lord Alistair")  
("Lealtad de Thomas")-[:PARECIA\_INQUEBRANTABLE]→("Lord Alistair")  
("Thomas")-[:ES\_MAYORDOMO\_DE\_TODA\_LA\_VIDA\_DE]→("Lord Alistair")  
("Detective Harding")-[:HABIA\_INTERCEPTADO\_NOTA\_DE]→("Thomas")  
("Thomas")-[:PEDIA\_CONSEJO\_A]→("Penélope")  
("Penélope")-[:ES\_HERMANA\_DE]→("Thomas")  
("Penélope")-[:TRABAJABA\_COMO\_AMA\_DE\_LLAVES\_PARA]→("Sr. Croft")  
("Sr. Croft")-[:ES\_PRINCIPAL\_RIVAL\_DE\_NEGOCIOS\_DE]→("Lord Alistair")  
("Sr. Croft")-[:NEGABA]→("cualquier animosidad")  
("Eleonor Croft")-[:ES\_HIJA\_DE]→("Sr. Croft")  
("Eleonor Croft")-[:NO\_OCULTABA\_DESDEN\_POR]→("los Blackwood")  
("Eleonor Croft")-[:NO\_OCULTABA\_DESDEN\_ESPECIAL\_POR]→("Edward")

```
("Eleonor Croft")-[:CONSIDERABA_OBSTACULO_A]→("Edward")
("Edward")-[:ES_OBSTACULO_PARA_PLANES_DE_EXPANSION_DE]→("Eleonor Croft")
("Eleonor Croft")-[:HABIA_DISCUTIDO_PLANES_CON]→("Julian")
("Clara")-[:HABIA_SIDO_VISTA_ENTREGANDO_PAQUETE_A]→("un desconocido")
("Desconocido")-[:ESTABA_CERCA_DE]→("el muelle")
("Detective Harding")-[:RECONOCIO_VAGAMENTE_COMO_ANTIGUO_ASOCIADO_DEL_
("Un desconocido")-[:ES_ANTIGUO_ASOCIADO_DE]→("Coronel Montgomery")
```

Podríamos trabajar aún más el *prompt* para nuestro caso de uso, e incluso puede ser necesario aplicar filtros posteriores en nuestra aplicación, pero vemos la potencia que tienen los modelos para crear salidas estructuradas a partir de escribir la instrucción de una manera clara y asertiva, y especificando el resultado que esperamos obtener.

## Responder preguntas (Question Answering)

Una de las mejores maneras de hacer que el modelo responda a respuestas específicas es mejorar el formato del *prompt*. Como se mencionó antes, un *prompt* puede combinar instrucciones, contexto, entrada y señales de salida para obtener resultados mejorados. Aunque estos componentes no son necesarios, se convierte en una buena práctica ya que cuanto más específico seamos con la instrucción, mejores resultados obtendremos. A continuación hay un ejemplo de cómo se vería esto siguiendo un *prompt* más estructurado.

*Prompt:*

Responda la pregunta basándose en el contexto siguiente. Mantenga la respuesta breve. Responda "No estoy seguro de la respuesta" si no está seguro de la respuesta.

Contexto: El teplizumab tiene sus raíces en una compañía farmacéutica de Nueva Jersey llamada Ortho Pharmaceutical. Allí, los científicos generaron una versión temprana del anticuerpo, denominada OKT3. Originaria de ratones, la molécula pudo unirse a la superficie de las células T y limitar su potencial de destrucción celular. En 1986, fue aprobado para ayudar a prevenir el rechazo de órganos después de trasplantes de riñón, lo que lo convirtió en el primer anticuerpo terapéutico permitido para uso humano.

Pregunta: ¿De qué se obtuvo originalmente OKT3?

Respuesta:

Salida:

OKT3 se obtuvo originalmente de ratones.

Contexto obtenido de: [Nature](#)(opens in a new tab).

---

## Clasificación de texto (Text Classification)

Hasta ahora, hemos utilizado instrucciones simples para realizar una tarea. En casos más difíciles, proporcionar solo instrucciones no será suficiente. Aquí es donde necesitamos pensar más en el contexto y los diferentes elementos que podemos usar en un *prompt*. Otros elementos que podemos proporcionar son datos de entrada o ejemplos.

Intentemos demostrar esto proporcionando un ejemplo de clasificación de texto.

*Prompt:*

Clasifica el texto en neutro, negativo o positivo.

Texto: Creo que la comida estuvo bien.

Sentimiento:

Salida:

Neutro

Le dimos la instrucción de clasificar el texto y el modelo respondió con 'Neutro', que es correcto. No hay nada malo con esto, pero digamos que lo que realmente necesitamos es que el modelo dé la etiqueta en el formato exacto que deseamos. Entonces, en lugar de *Neutro*, queremos que devuelva *neutro*. ¿Cómo logramos esto? Hay diferentes maneras de hacerlo. Aquí nos importa la especificidad, por lo que cuanto más información podamos proporcionar al *prompt*, mejores serán los resultados. Podemos intentar proporcionar ejemplos para especificar el comportamiento correcto. Intentemos de nuevo:

*Prompt:*

Clasifica el texto en neutro, negativo o positivo.

Texto: Creo que las vacaciones están bien.

Sentimiento: neutro

Texto: Creo que la comida estuvo bien.

Sentimiento:

Salida:

neutro

¡Perfecto! Esta vez el modelo devolvió **neutro**, que es la etiqueta específica que buscábamos. Parece que el ejemplo proporcionado en el prompt ayudó al modelo a ser específico en su salida.

Para resaltar por qué a veces ser específico es importante, revisemos el ejemplo a continuación e identifiquemos el problema:

*Prompt:*

Clasifica el texto en nutral, negativo o positivo.

Texto: Creo que las vacaciones están bien.

Sentimiento:

Salida:

Neutral

¿Cuál es el problema aquí? Como pista, la etiqueta inventada "nutral" es completamente ignorada por el modelo. En cambio, el modelo emite "Neutral" ya que tiene cierto sesgo hacia esa etiqueta. Pero supongamos que lo que realmente quieras es "nutral". ¿Cómo arreglaríamos esto? ¿Quizá podamos intentar agregar descripciones a las etiquetas o agregar más ejemplos al *prompt*? Si no estamos seguros, discutiremos algunas ideas en las próximas secciones.

## Conversación (Conversation)

Quizás una de las cosas más interesantes que podamos lograr con la ingeniería de *prompts* es instruir al sistema LLM sobre cómo comportarse, su intención y su identidad. Esto es particularmente útil cuando estamos construyendo sistemas conversacionales como chatbots de servicio al cliente.

Por ejemplo, pensemos un sistema conversacional que sea capaz de generar respuestas más técnicas y científicas a las preguntas. Observemos cómo le estamos diciendo explícitamente cómo comportarse a través de la instrucción. Esto a veces se conoce como *indicación de rol*.

*Prompt:*

La siguiente es una conversación con un asistente de investigación de IA. El tono de I asistente debe ser técnico y científico.

Humano: Hola, ¿quién eres?

AI: ¡Saludo! Soy asistente de investigación de IA. ¿Cómo puedo ayudarte hoy?

Humano: ¿Puedes hablarme sobre la creación de agujeros negros?

AI:

Salida:

¡Seguro! Los agujeros negros son regiones del espacio-tiempo donde la fuerza gravitacional es tan fuerte que nada, ni siquiera la luz, puede escapar de ellos. Se crean cuando una estrella muy masiva muere y su núcleo colapsa sobre sí mismo, formando una singularidad de densidad infinita. La intensa gravedad de la singularidad atrae toda la materia y radiación a su alrededor, creando el agujero negro.

Nuestro asistente de investigación de IA suena un poco demasiado técnico, ¿verdad? Bien, cambiemos este comportamiento e instruyamos al sistema para que dé respuestas más accesibles.

*Prompt:*

La siguiente es una conversación con un asistente de investigación de IA. Las respuestas del asistente deben ser fáciles de entender incluso para los estudiantes de primaria.

Humano: Hola, ¿quién eres?

AI: ¡Saludo! Soy asistente de investigación de IA. ¿Cómo puedo ayudarte hoy?

Humano: ¿Puedes hablarme sobre la creación de agujeros negros?

AI:

Salida:

¡Seguro! Los agujeros negros se crean cuando una estrella muy masiva se queda sin combustible y colapsa sobre sí misma. La gravedad de la estrella es tan fuerte que nada, ni siquiera la luz, puede escapar de su atracción. Esto crea un punto en el espacio-tiempo con una intensa fuerza gravitacional llamado agujero negro.

## Code Generation

Una aplicación en la que los Modelos de Lenguaje a Gran Escala (LLMs) son bastante efectivos es la generación de código. [Copilot](#) es un gran ejemplo de esto. Hay un gran número de tareas de generación de código que podemos realizar con *prompts*. Veamos algunos ejemplos a continuación.

Primero, intentemos un programa simple que salude al usuario.

*Prompt:*

```
/*
Programa que pregunta al usuario su nombre y dice "Hola"
*/
```

Salida:

```
// Programa que pregunta al usuario su nombre y dice "Hola"
const prompt = require('prompt-sync')();

// Pregunta al usuario su nombre
const name = prompt('Por favor, introduce tu nombre: ');

// Saluda al usuario
console.log(`Hola, ${name}!`);
```

Observemos que ni siquiera necesitamos especificar el lenguaje a utilizar.

Cambiemos un poco de nivel. El ejemplo a continuación muestra cuán poderosos pueden ser los LLMs, con un poco más de esfuerzo en el diseño de los *prompts*.

*Prompt:*

```
"""
Table departments, columns = [DepartmentId, DepartmentName]
Table students, columns = [DepartmentId, StudentId, StudentName]
Crear una consulta MySQL para todos los estudiantes del Departamento de Informática.
"""


```

Salida:

```
SELECT StudentId, StudentName
FROM students
WHERE DepartmentId IN (SELECT DepartmentId FROM departments WHERE DepartmentName = 'Departamento de Informática');
```

En este caso, proporcionamos datos sobre el esquema de la base de datos y le pedimos que generara una consulta MySQL válida.

### 3. Prompts desde código

Hasta ahora hemos visto diferentes técnicas para dialogar con una sistema de chat, de forma de poder obtener respuestas más específicas, a través de estructurar los *prompts*

que usamos en nuestras interacciones con los LLMs. Los frontend que nos permiten interactuar con los chatbots, están preparados para ser usados por humanos. Sin embargo, nuestro interés es poder dialogar con estos sistemas desde código de programación. Para ello, es importante conocer cómo llamar a los modelos de manera adecuada, y cómo indicar los parámetros de la conversación que deseamos.

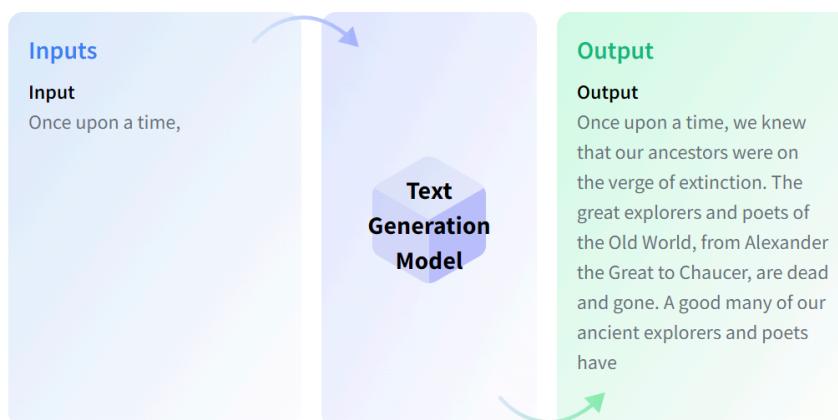
Es fundamental poder aprovechar los LLMs desde una aplicación, lo que nos permite automatizar tareas y utilizar a los modelos de lenguaje como si fueran una función o un servicio web.

## Tipos de modelos

Los grandes modelos de lenguaje, típicamente se diferencian modelos generativos de texto y modelos conversacionales. Son dos categorías de modelos de lenguaje con diferentes enfoques y especializaciones, aunque con algunas superposiciones en sus capacidades:

### Modelos Generativos de Texto

Los modelos de generación de texto están esencialmente entrenados con el objetivo de completar textos. También se los conoce como *modelos causales*.



<https://huggingface.co/tasks/text-generation>

Los desafíos anteriores al trabajar con estos modelos, eran controlar tanto la coherencia como la diversidad del texto a través de parámetros de inferencia y los sesgos discriminatorios. Los resultados que sonaban más coherentes eran menos creativos y más cercanos a los datos de entrenamiento originales, y no parecían algo que sería dicho por un humano. Los desarrollos recientes superaron estos desafíos, y las interfaces de usuario amigables permitieron que todos probaran estos modelos.

Tener más variedad de modelos de generación de texto de código abierto permite a las empresas mantener la privacidad de sus datos (propiedad intelectual), la capacidad de adaptar modelos a sus dominios más rápidamente y reducir costos de inferencia en lugar de depender de APIs pagas cerradas.

The screenshot shows the Hugging Face Hosted inference API interface. At the top, there's a header with a lightning bolt icon and the text "Hosted inference API". Below the header, there are two tabs: "Text Generation" (selected) and "Examples". In the main area, there's a text input box containing the prompt: "My name is Merve and my favorite is the one I'm wearing. I'm wearing a black".

Plataformas como Hugging Face, nos permiten probar diferentes modelos de generación de texto ([https://huggingface.co/models?pipeline\\_tag=text-generation](https://huggingface.co/models?pipeline_tag=text-generation))

El modelo GPT-3 en sí es un modelo de lenguaje de texto generativo, cuyas características principales son:

1. **Amplio Alcance en Generación de Texto:** Están diseñados para generar texto en una variedad de formatos y estilos, incluyendo pero no limitado a la conversación. Pueden crear historias, artículos, poemas, código, y más.
2. **Versatilidad en Aplicaciones:** Pueden ser utilizados para tareas como resumir textos, traducir idiomas, responder preguntas de conocimiento general, y generar contenido creativo.
3. **Menor Especialización en Diálogos:** Aunque capaces de manejar diálogos hasta cierto punto, no están específicamente optimizados para mantener conversaciones coherentes y contextualizadas a lo largo de múltiples intercambios.

## Casos de Uso

- **Modelos de Instrucción**

Un modelo entrenado para la generación de texto puede adaptarse posteriormente para seguir instrucciones. Uno de los modelos de código abierto más utilizados para instrucciones es OpenAssistant (<https://github.com/LAION-AI/Open-Assistant>).

- **Generación de Código**

Un modelo de Generación de Texto, también conocido como modelo causal de lenguaje, puede ser entrenado desde cero con código para ayudar a los programadores en sus tareas repetitivas de codificación. Uno de los modelos de código abierto más populares para la generación de código es [StarCoder](#), que puede generar código en más de 80 lenguajes.

- **Generación de Historias**

Un modelo de generación de historias puede recibir una entrada como "Érase una

vez" y proceder a crear un texto similar a una historia basado en esas primeras palabras.

### Variantes según tareas

- **Modelos de Generación de terminación (Completion Generation Models)**

Una variante popular de los modelos de Generación de Texto predice la siguiente palabra dada un conjunto de palabras. Palabra por palabra, se genera un texto más largo que resulta en, por ejemplo: "Dada una oración incompleta, complétala", "Continuar una historia dadas las primeras oraciones.", "Proporcionado una descripción de código, generar el código.", etc.

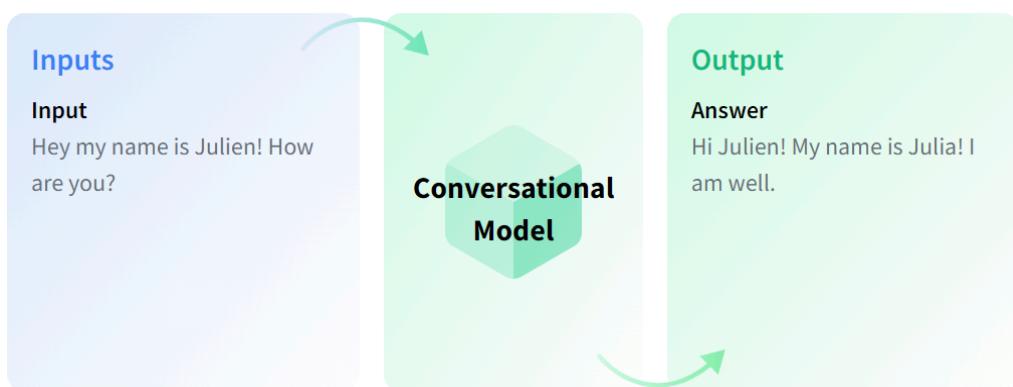
Los modelos más populares para esta tarea son los modelos basados en GPT o la serie Llama. Estos modelos están entrenados con datos que no tienen etiquetas, por lo que solo necesitamos texto plano para entrenar los modelos. Podemos entrenar modelos de generación de texto para generar una amplia variedad de documentos, desde código hasta historias.

- **Modelos de Generación de Texto-a-Texto (Text-to-Text Generation Models)**

Estos modelos están entrenados para aprender el mapeo entre un par de textos (por ejemplo, la traducción de un idioma a otro). Las variantes más populares de estos modelos son FLAN-T5 y BART. Los modelos de Texto-a-Texto están entrenados con capacidades de multitarea, pueden realizar una amplia gama de tareas, incluyendo resumen, traducción y clasificación de textos.

### Modelos Conversacionales

El modelado de respuestas conversacionales es la tarea de generar texto conversacional que sea relevante, coherente y con conocimiento, dado un prompt. Los modelos como el GPT-3.5 o GPT-4, están entrenados (fine-tuning) en prompts que consisten en conversaciones o instrucciones a través de RLHF (Reinforcement Learning from Human Feedback). De ese modo, se entrena un modelo de base generativo, para dotarlo de capacidades conversacionales. Estos modelos tienen aplicaciones en chatbots y como parte de asistentes de voz.



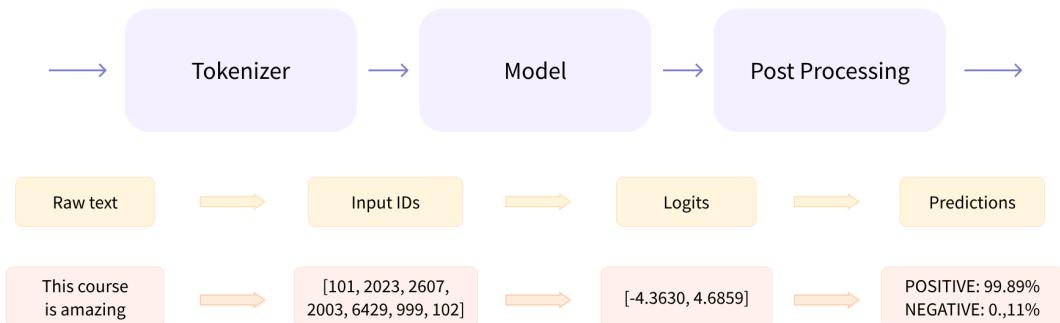
<https://huggingface.co/tasks/conversation>

## Características principales:

1. **Especialización en Diálogo:** Están específicamente diseñados para simular conversaciones humanas. Su arquitectura y entrenamiento se centran en mantener diálogos coherentes y contextuales.
2. **Uso en Chatbots y Asistentes Virtuales:** Son ideales para aplicaciones de chatbot y asistentes virtuales donde se requiere interacción conversacional fluida y natural. ChatGPT, DialoGPT y BlenderBot son ejemplos de modelos conversacionales, optimizados para responder de manera coherente y relevante dentro del contexto de una conversación.
3. **Manejo del Contexto Conversacional:** Tienen una mejor capacidad para manejar el contexto y mantener la coherencia en una serie de intercambios conversacionales.
4. **Limitaciones en Generación de Contenido No Conversacional:** Aunque excelentes en diálogos, pueden no ser tan flexibles o efectivos en la generación de otro tipo de contenido textual que no sea conversacional.

## Pipelines

Para utilizar un modelo de lenguaje y realizar *inferencia*, es necesario realizar una serie de procesos para preparar el texto, de modo que el modelo de base, reciba la información en el formato adecuado para funcionar correctamente. Estos procesos forman parte de lo que se llama “tubería” (pipeline en inglés), donde se realizan tareas como aplicación de templates, tokenización, embeddings, limpieza de texto, etc. Un pipeline puede realizar tareas tanto antes (pre), como después de utilizar el modelo (post), por ejemplo:



Los modelos de Hugging Face, tienen disponibles esos pipelines, que están asociados a cada modelo específico, y que nos facilitan la utilización tanto de los modelos generativos de texto, como los modelos conversacionales, entre otros. Los pipelines de la librería `transformers` de Hugging Face son herramientas de alto nivel diseñadas para facilitar el uso de modelos de procesamiento de lenguaje natural (NLP) pre-entrenados.

Están pensados para usuarios que no necesariamente tienen un profundo conocimiento en modelado de IA o NLP, pero que desean implementar funcionalidades avanzadas de manera rápida y sencilla.

## Pipeline de generación de texto

Veamos un ejemplo con un modelo generativo de texto:

```
from transformers import pipeline, GPT2LMHeadModel, GPT2Tokenizer

# Cargar el modelo y el tokenizador de GPT-2
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

# Crear un pipeline de generación de texto
text_generator = pipeline("text-generation", model=model, tokenizer=tokenizer)

# Generar texto a partir de un prompt
prompt = "In a distance future, the humand and robots"
generated_text = text_generator(prompt, max_length=50, num_return_sequences=2, te

# Mostrar el texto generado
for idx, txt in enumerate(generated_text):
    print(f"Texto generado {[idx]}:\n{txt['generated_text']}")
```

Y la salida será por ejemplo:

Texto generado [0]:

In a distance future, the humand and robots would do the same.

What happens next?

The robotic revolution is so far behind today's artificial intelligence. But even though th  
Texto generado [1]:

In a distance future, the humand and robots will be able to learn to work together in or

The future robots may not be human,

En este ejemplo, se carga el modelo `gpt2`, que es adecuado para la generación de texto (pipeline de tipo `text-generation`). Después de crear el pipeline, se proporciona un prompt inicial y se genera texto basado en ese prompt. El `max_length` controla la longitud máxima del texto generado, `temperature` controla la “creatividad” del modelo, y `num_return_sequences`

especifica cuántas secuencias de texto diferentes queremos generar a partir del prompt.

## Pipeline conversacional

Del mismo modo, podemos utilizar modelos conversacionales usando pipelines de tipo **conversational**. Veamos un ejemplo:

```
from transformers import pipeline, AutoModelForCausalLM, AutoTokenizer, Conversatio
# Cargar el modelo y el tokenizador
model_name = "microsoft/DialoGPT-medium"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Crear un pipeline de conversación
conversation_pipeline = pipeline("conversational", model=model, tokenizer=tokenizer)

# Crear un objeto de conversación
conversation = Conversation()

# Agregar el mensaje inicial de la conversación
conversation.add_user_input("Hello, how are you?")

# Generar una respuesta
response = conversation_pipeline(conversation)

# Agregar más mensajes a la conversación
conversation.add_user_input("What can you tell me about AI?")
response = conversation_pipeline(conversation)

# Mostrar la conversación completa
print(conversation)
```

Y obtendremos una respuesta como la siguiente:

```
Conversation id: 2b5990a2-ac64-4db1-8658-8ba3efe752e1
user: Hello, how are you?
assistant: I'm good, how are you?
user: What can you tell me about AI?
assistant: AI is a great game, I've played it for years.
```

En el ejemplo, primero se agrega un mensaje inicial al objeto `Conversation`. Luego, se pasa este objeto al pipeline para generar una respuesta. Después, se agrega otro mensaje del usuario a la misma conversación, y se vuelve a pasar al pipeline. Esto continúa el diálogo, manteniendo el contexto de la conversación anterior.

El objeto `Conversation` mantiene un registro de toda la interacción, permitiendo al modelo contextualizar sus respuestas en función de la conversación completa. Podemos seguir añadiendo más mensajes de usuario y generando respuestas para crear una conversación más larga y contextualizada.

### Pipeline de generación de texto a texto (text2text-generation)

Los modelos de generación de texto a texto tienen un pipeline específico llamado `text2text-generation`. Este pipeline toma una entrada que contiene la oración, incluida la tarea, y devuelve la salida de la tarea realizada:

```
from transformers import pipeline

text2text_generator = pipeline("text2text-generation", model='t5-base')

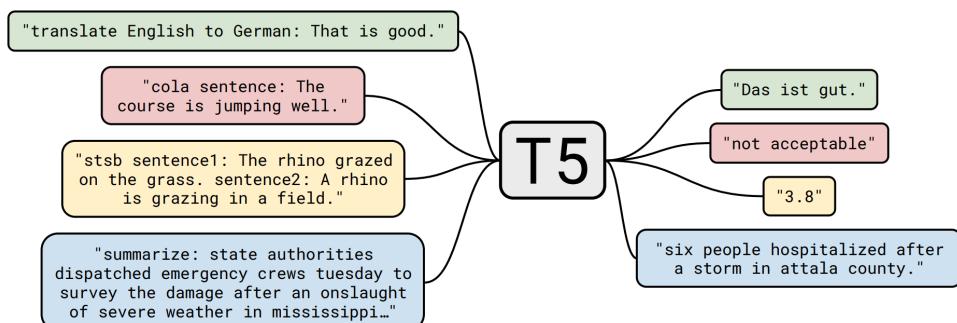
text2text_generator("question: What is 42 ? context: 42 is the answer to life, the universe and everything")

text2text_generator("traducir del inglés al francés: Estoy muy feliz")
```

Y obtendremos algo como:

```
[{'generated_text': 'the answer to life, the universe and everything'}]
[{'generated_text': 'Je suis très heureux'}]
```

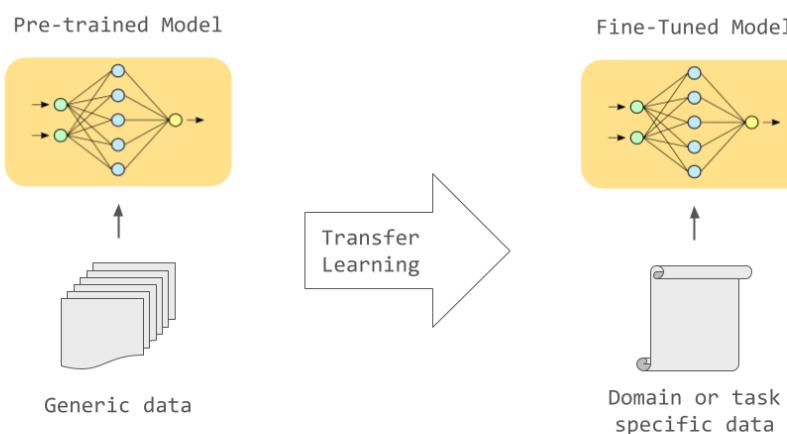
En este caso, hemos utilizado el modelo T5-Base el cual es multipropósito, pero no soporta español:



Para utilizar pipelines de tipo [text2text-generation](#), existen muchos otros modelos disponibles en Hugging Face que podemos utilizar ([https://huggingface.co/models?pipeline\\_tag=text2text-generation](https://huggingface.co/models?pipeline_tag=text2text-generation)).

## Plantillas de chat o instrucciones (chat/instruction templates)

Muchos modelos se basan en LLM fundacionales o pre-entrenados, que luego se vuelven a entrenar (fine-tuning) con datasets de instrucciones especialmente diseñados para mejorar y refinar las habilidades de estos modelos en tareas específicas:

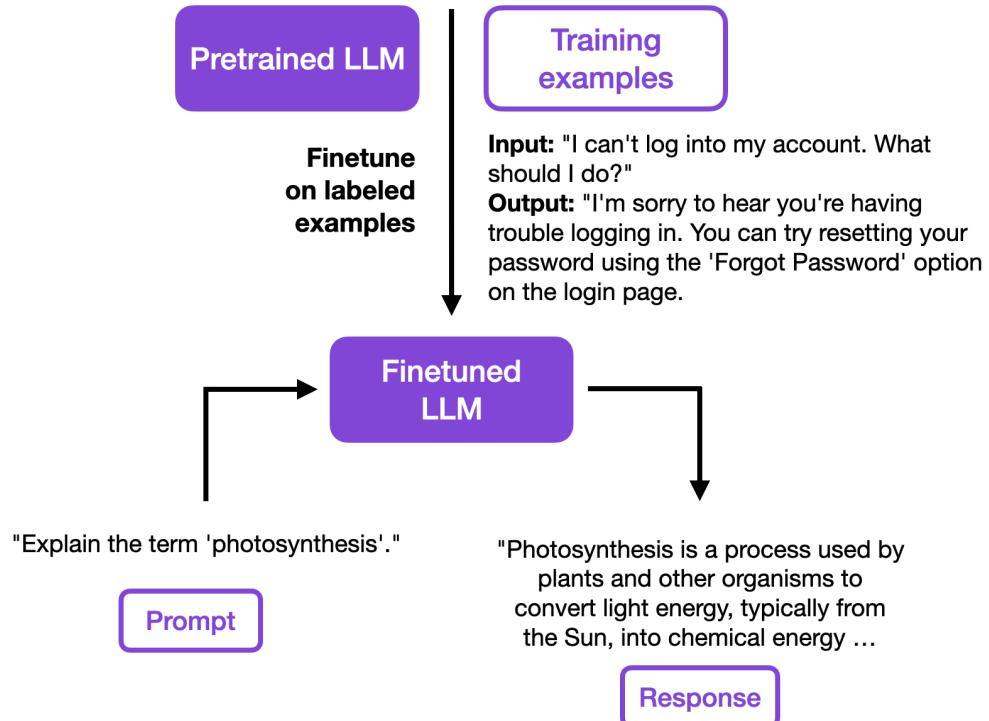


Estos conjuntos de datos suelen incluir una variedad de ejemplos de texto, que pueden abarcar desde preguntas y respuestas hasta instrucciones y respuestas. El propósito principal es enseñar al modelo cómo seguir instrucciones o cómo responder adecuadamente a ciertos tipos de solicitudes.

Cuando a un modelo de lenguaje como GPT-3 o GPT-4 se le realiza el "ajuste fino" con estos conjuntos de datos, aprende a comprender y ejecutar mejor las tareas que se le presentan. Por ejemplo, puede mejorar en la comprensión de instrucciones complejas, en la generación de respuestas más relevantes y precisas, o en la adaptación a estilos de comunicación específicos. Esto es particularmente útil para aplicaciones especializadas, donde se requiere que el modelo comprenda y responda de manera adecuada a un conjunto específico de instrucciones o preguntas relacionadas con un campo o tema particular, o para usarlos en sistemas de diálogo.

El proceso de adaptación de los LLMs de base, hacia modelos entrenados en seguir instrucciones (instruction-following), se llama alineación.

<https://openai.com/research/instruction-following>:



Los datasets de instrucciones, se utilizan para el ajuste fino de los Modelos de Lenguaje de Gran Escala (LLMs). Este ajuste fino, típicamente utiliza el aprendizaje supervisado e incluye tanto una cadena de entrada como una cadena de salida esperada. La cadena de entrada y salida siguen una plantilla conocida como formato de conjunto de datos de instrucciones (por ejemplo, `[INST] <>[SYS]>`). ChatML de OpenAI y Alpaca de Stanford son ejemplos de datasets de Instrucciones. A continuación, se muestra el formato de datos de instrucción utilizado por Alpaca para el ajuste fino que incluye información de contexto (el campo de entrada a continuación):

Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.

### Instruction:

{instruction}

### Input:

{input}

### Response:

Debido a que los modelos han sido ajustados (entrenados) para generar texto en contextos de diálogo o consultas, es que al momento de la inferencia necesitaremos formatear nuestros *prompts* del mismo modo, para no deteriorar la calidad de nuestras consultas o diálogos.

Las conversaciones o en la solicitud de instrucciones, los mensajes tienen un rol y un contenido, siendo este último el texto real del mensaje. Comúnmente, los roles son "user" para los mensajes enviados por el usuario, "assistant" para las respuestas escritas por el modelo y, opcionalmente, "system" para directivas de alto nivel dadas al inicio de la conversación.

Si todo esto parece un poco abstracto, aquí hay un ejemplo de chat para hacerlo más concreto:

```
[  
{"role": "user", "content": "¡Hola!"},  
 {"role": "assistant", "content": "¡Encantado de conocerte!"}  
]
```

Esta secuencia de mensajes necesita ser convertida en un string de texto antes de que pueda ser tokenizada y utilizada como entrada para un modelo. El problema, sin embargo, es que hay muchas maneras de hacer esta conversión. Podríamos, por ejemplo, convertir la lista de mensajes en un formato de Instant Messenger:

```
Usuario: ¡Hola!  
Bot: ¡Encantado de conocerte!
```

O podríamos agregar tokens especiales para indicar los roles:

```
[USER] ¡Hola! [/USER]  
[ASST] ¡Encantado de conocerte! [/ASST]
```

O también podríamos agregar tokens para indicar los límites entre mensajes, pero insertar la información del rol como un string:

```
<|im_start|>user  
¡Hola!<|im_end|>  
<|im_start|>assistant  
¡Encantado de conocerte!<|im_end|>
```

Hay muchas maneras de hacer esto, y ninguna de ellas es la mejor o la correcta. La manera, depende de como los diferentes modelos han sido entrenados. Los ejemplos anteriores, no son inventados; sino que son reales y se usan en algunos modelos populares. Una vez que un modelo ha sido entrenado con un cierto formato, queremos asegurarnos de que las futuras entradas usen el mismo formato, de lo contrario podemos deteriorar el rendimiento del modelo.

### **Plantillas: Una forma de guardar información de formato**

En general el formato que necesitamos está correctamente documentado junto al

modelo. Si no lo está, puede dificultarse el uso del modelo, incluso al extremo de resultarnos inútil.



Usar un formato incorrecto de plantilla es un problema peligroso, porque usar el formato de chat incorrecto es un error silencioso: no obtendremos un fallo evidente o una excepción de Python que nos diga que algo está mal, el modelo simplemente funcionará peor de lo que lo haría con el formato correcto, y será muy difícil depurar la causa.

Las plantillas de chat, generalmente son cadenas de plantillas [Jinja](#) que se guardan y cargan con el tokenizador, y que contienen toda la información necesaria para convertir una lista de mensajes de chat en una entrada correctamente formateada para nuestro modelo. Aquí hay tres cadenas de plantillas de chat, correspondientes a los tres formatos de mensaje anteriores:

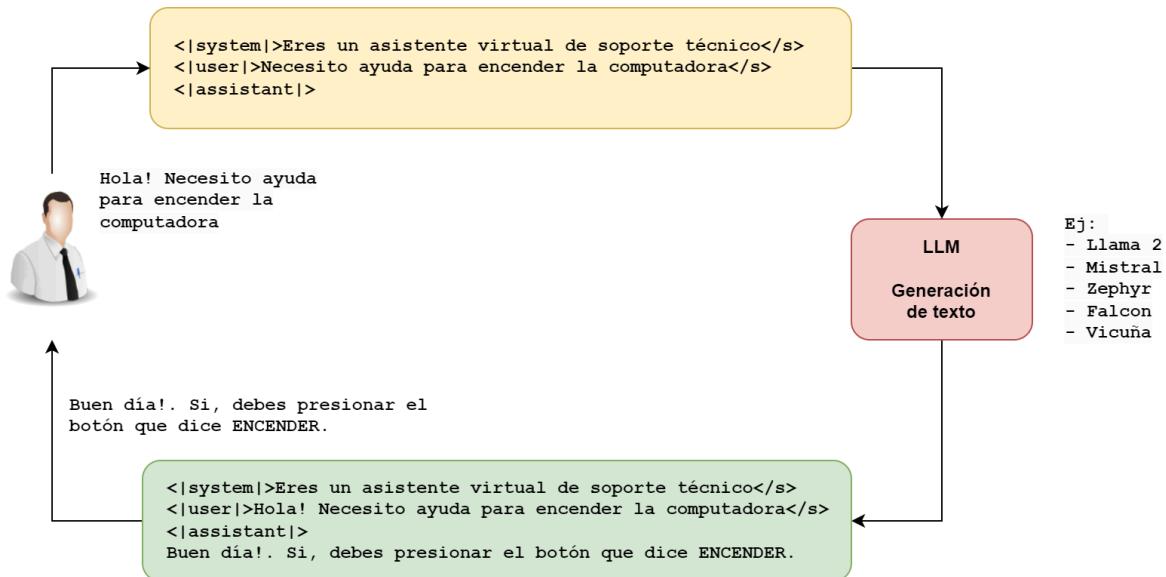
```
{% for message in messages %}  
{% if message['role'] == 'user' %}  
{{ "Usuario : " }}  
{% else %}  
{{ "Bot : " }}  
{{ message['content'] + '\n' }}  
{% endfor %}
```

```
{% for message in messages %}  
{% if message['role'] == 'user' %}  
{{ "[USER] " + message['content'] + " [/USER]" }}  
{% else %}  
{{ "[ASST] " + message['content'] + " [/ASST]" }}  
{{ message['content'] + '\n' }}  
{% endfor %}
```

```
"{% for message in messages %}"  
"{{' + message['role'] + '\n' + message['content'] + " + '\n'}}"  
"{% endfor %}"
```

Aunque [Jinja](#) puede ser confuso al principio si no estamos familiarizados con él, en la práctica los programadores de Python pueden aprenderlo rápidamente. Sería ideal que los desarrolladores de modelos, adopten un formato 'estándar', como el [ChatML](#) creado por OpenAI. El formato tiene la ventaja de ser muy flexible con los roles, ya que el rol se inserta simplemente como una cadena en lugar de tener tokens de rol específicos.

Para desarrollar un sistema de diálogo de tipo chat con motores de generación de texto, debemos formatear los mensajes de manera adecuada, teniendo en cuenta las características del modelo y del servicio que estamos utilizando.



Veamos como a partir de un mismo ejemplo, nos encontraremos con diferentes formas de formatear el mensaje, según algunos modelos Open Source conocidos:

```

chat_messages = [
    {"role": "system", "content": "Eres un escritor de historias de misterio."},
    {"role": "user", "content": "Completar la frase: El cielo está"},
]

# Zephyr template:
<|system|>Eres un escritor de historias de misterio.</s>
<|user|>Completar la frase: El cielo está</s>
<|assistant|>

-----
# Llama2 template:
<s>[INST] <<SYS>>
Eres un escritor de historias de misterio.
<</SYS>>

Completar la frase: El cielo está [/INST]
-----
# Vicuna template:
Eres un escritor de historias de misterio.

```

```

USER: Completar la frase: El cielo está
ASSISTANT:
-----
# Falcon template:
System: Eres un escritor de historias de misterio.
User: Completar la frase: El cielo está
Falcon:
-----
# Open Assistant template:
<|system|>Eres un escritor de historias de misterio.</s><|prompter|>Completar la fras

```

Una forma sencilla de aplicar la plantilla adecuada para generar el prompt para nuestro modelo, es usando la librería `llm-templates` ([https://github.com/jpmanson/llm\\_templates](https://github.com/jpmanson/llm_templates)). Veamos un ejemplo:

```

!pip install llm-templates
from llm_templates import Formatter, Conversation

# Nuestra conversación
messages = [
    {"role": "system", "content": "Eres un escritor de historias de misterio."},
    {"role": "user", "content": "Completar la frase: El cielo está"}
]

formatter = Formatter()
conversation = Conversation(model='zephyr', messages=messages)
print(formatter.render(conversation, add_assistant_prompt=True))

```

Y la salida será como la siguiente:

```

<|system|>Eres un escritor de historias de misterio.</s>
<|user|>Completar la frase: El cielo está</s>
<|assistant|>

```

## Uso de modelos via API

Un modelo de lenguaje puede ser utilizado tanto de forma local como remota. Veamos las diferencias principales:

### Ejecución Local

- Recursos y Hardware:** Requiere hardware significativo, generalmente incluyendo GPUs de alto rendimiento. Esto puede ser costoso y complejo de configurar y

mantener. Sin embargo, los LLM cada vez se optimizan más para poder correr en CPUs tradicionales.

2. **Mayor control:** Mayor control sobre el modelo, lo que incluye la capacidad de ajustar más sus parámetros, y también mantener la privacidad de los datos procesados.
3. **Escalabilidad:** Escalar puede ser difícil y costoso, ya que requiere inversión adicional en hardware y gestión de infraestructura.
4. **Mantenimiento y Actualizaciones:** El mantenimiento, incluidas las actualizaciones del modelo y la gestión de dependencias, corre a cargo del usuario.
5. **Latencia:** La latencia de procesamiento puede ser menor, especialmente si el hardware es potente y está bien configurado.
6. **Costos Iniciales:** Puede haber un costo inicial significativo debido a la necesidad de hardware y configuración.

### Ejecución Remota vía API

1. **Accesibilidad y Facilidad de Uso:** Generalmente, es más fácil de usar. La configuración inicial y el mantenimiento técnico lo maneja el proveedor del servicio.
2. **Costos Basados en Uso:** Los costos suelen basarse en el uso, lo que puede ser más económico para usuarios con necesidades de cómputo moderadas o irregulares.
3. **Escalabilidad:** Las API ofrecidas por proveedores como OpenAI o Hugging Face son altamente escalables, pudiendo manejar un gran número de solicitudes sin esfuerzo adicional por parte del usuario.
4. **Sin Necesidad de Hardware Específico:** No hay necesidad de invertir en hardware costoso, ya que el procesamiento se realiza en los servidores del proveedor.
5. **Latencia:** La latencia puede ser mayor debido al tiempo de transmisión de datos hacia y desde el servidor remoto.
6. **Dependencia de la Conexión a Internet:** Requiere una conexión a Internet confiable y puede verse afectada por problemas de conectividad.
7. **Privacidad de Datos:** La privacidad de los datos puede ser una consideración, ya que los datos se procesan en servidores externos.
8. **Actualizaciones y Mantenimiento:** El proveedor se encarga de las actualizaciones y el mantenimiento del modelo.

Veamos un ejemplo de utilización de un **modelo de generación de texto** como chat, a través de su API en Hugging Face. Utilizaremos el modelo "[HuggingFaceH4/zephyr-7b-beta](#)":

```
import requests
import json
from google.colab import userdata
```

```

from jinja2 import Template

def zephyr_chat_template(messages, add_generation_prompt=True):
    # Definir la plantilla Jinja
    template_str = "{% for message in messages %}"
    template_str += "{% if message['role'] == 'user' %}"
    template_str += "<|user|>{{ message['content'] }}</s>\n"
    template_str += "{% elif message['role'] == 'assistant' %}"
    template_str += "<|assistant|>{{ message['content'] }}</s>\n"
    template_str += "{% elif message['role'] == 'system' %}"
    template_str += "<|system|>{{ message['content'] }}</s>\n"
    template_str += "{% else %}"
    template_str += "<|unknown|>{{ message['content'] }}</s>\n"
    template_str += "{% endif %}"
    template_str += "{% endfor %}"
    template_str += "{% if add_generation_prompt %}"
    template_str += "<|assistant|>\n"
    template_str += "{% endif %}"

    # Crear un objeto de plantilla con la cadena de plantilla
    template = Template(template_str)

    # Renderizar la plantilla con los mensajes proporcionados
    return template.render(messages=messages, add_generation_prompt=add_generation_prompt)

# Tu clave API de Hugging Face
api_key = userdata.get('HUGGINGFACE_TOKEN') # Aquí va la clave hfxxxxxxxxxxxxx

# URL de la API de Hugging Face para la generación de texto
api_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7b-be"

# Cabeceras para la solicitud
headers = {"Authorization": f"Bearer {api_key}"}

# Preparamos el prompt incluyendo el rol del sistema y el texto inicial
chat_prompt = [{"role": "system", "content": "Eres un escritor de historias de misterio."}

prompt_with_template = zephyr_chat_template(chat_prompt)

print(f"Nuestro prompt:\n{prompt_with_template}")

# Datos para enviar en la solicitud POST
# Sobre los parámetros: https://huggingface.co/docs/transformers/main_classes/text_

```

```

data = {
    "inputs": prompt_with_template,
    "parameters": {
        "max_new_tokens": 256,
        "temperature": 0.7,
        "top_k": 50,
        "top_p": 0.95
    }
}

# Realizamos la solicitud POST
response = requests.post(api_url, headers=headers, json=data)

print("\nRespuesta del modelo:")
for idx, txt in enumerate(response.json()):
    print(txt['generated_text'])

```

Y obtendremos un resultado como el siguiente:

Nuestro prompt:

```

<|system|>Eres un escritor de historias de misterio.</s>
<|user|>Completar la frase: El cielo está</s>
<|assistant|>
```

Respuesta del modelo:

```

<|system|>Eres un escritor de historias de misterio.</s>
<|user|>Completar la frase: El cielo está</s>
<|assistant|>
```

El cielo está lleno de estrellas" o "El cielo está nublado hoy" dependiendo de la situación.

Más recientemente, se incorpora la posibilidad de llamar a la API de Hugging Face de una manera más limpia con la librería `huggingface_hub`, usando el objeto `InferenceClient`, que maneja automáticamente la aplicación del template específico de cada modelo, sin tener que preocuparnos por construir el mensaje con `jinja`, como en el caso anterior:

```

from huggingface_hub import InferenceClient
from google.colab import userdata

# Obtener el token desde los secretos de Colab
api_key = userdata.get('HUGGINGFACE_TOKEN')

# Inicializar el cliente
client = InferenceClient()

```

```

        model="HuggingFaceH4/zephyr-7b-beta",
        token=api_key
    )

# Preparar los mensajes
messages = [
    {"role": "system", "content": "Eres un escritor de historias de misterio."},
    {"role": "user", "content": "Completar la frase: El cielo está"}
]

# Generar respuesta directamente
response = client.chat_completion(
    messages=messages,
    max_tokens=256,
    temperature=0.7,
    top_p=0.95
)

print(response.choices[0].message.content)

```

## 4. RAG (Retrieval Augmented Generation)

La Generación Aumentada por Recuperación (RAG) es el concepto de proporcionar a los Modelos de Lenguaje de Gran Escala (LLMs) información adicional proveniente de una fuente de conocimiento externa. Esto les permite generar respuestas más precisas y contextuales, reduciendo las alucinaciones. El siguiente es el paper original donde se introduce la técnica:

Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks

Large pre-trained language models have been shown to store factual knowledge in their parameters, and achieve state-of-the-art results when fine-tuned on downstream NLP tasks. However, their...

 <https://arxiv.org/abs/2005.11401>

Los LLMs de última generación son entrenados con grandes cantidades de datos para lograr un amplio espectro de conocimiento general almacenado en los pesos de la red neuronal (memoria paramétrica). Sin embargo, al solicitar a un LLM que genere una respuesta que requiere conocimientos que no estaban incluidos en sus datos de entrenamiento, como información más reciente, propietaria o específica de un dominio, puede llevar a inexactitudes fácticas (alucinaciones), o bien puede no conocer la respuesta, como se ilustra en la siguiente captura de pantalla:

You  
Quién ganó el mundial de Qatar 2022?

ChatGPT  
Lamento no poder proporcionarte información actualizada sobre el resultado de la Copa Mundial de la FIFA en Qatar 2022, ya que mi conocimiento se detiene en enero de 2022 y no tengo acceso a datos en tiempo real. Te recomendaría verificar las fuentes de noticias deportivas actuales o realizar una búsqueda en línea para obtener información actualizada sobre el resultado del torneo.

🔗 🤝 🤗 🎁

Por lo tanto, es importante cerrar la brecha entre el conocimiento general del LLM y cualquier contexto adicional para ayudar al LLM a generar respuestas más precisas y contextualizadas, al tiempo que se reducen las alucinaciones.

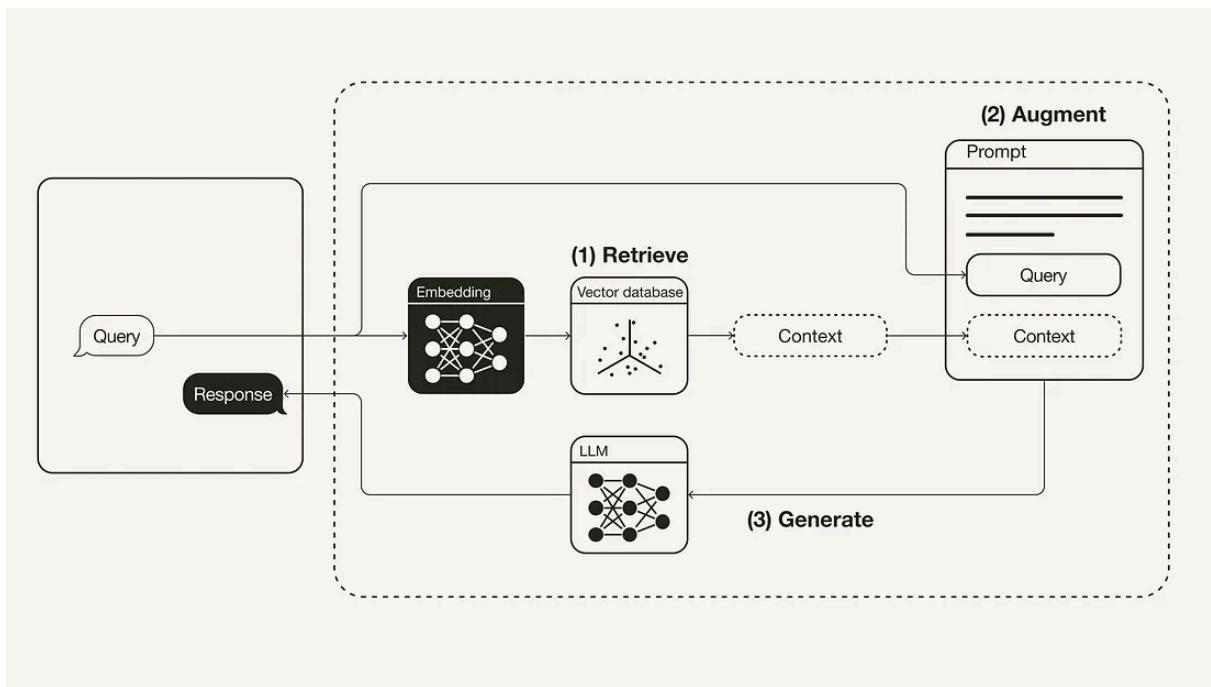
Tradicionalmente, las redes neuronales se adaptan a información específica de dominio o propietaria mediante el ajuste fino del modelo. Aunque esta técnica es efectiva, también es intensiva en cómputo, costosa y requiere experiencia técnica, lo que la hace menos ágil para adaptarse a la información en constante evolución.

En 2020, Lewis et al. propusieron una técnica más flexible llamada Generación con Recuperación Aumentada (RAG, por sus siglas en inglés) en el paper "[Generación con Recuperación Aumentada para Tareas de Procesamiento de Lenguaje Natural Intensivas en Conocimiento](#)". En ese artículo, los investigadores combinaron un modelo generativo con un módulo recuperador para proporcionar información adicional de una fuente de conocimiento externa que puede actualizarse más fácilmente.

En términos simples, RAG es para LLMs lo que un examen con libro abierto es para los seres humanos. En un examen con libro abierto, se permite a los estudiantes llevar materiales de referencia, como libros de texto o apuntes, que pueden utilizar para buscar información relevante y responder a una pregunta. La idea detrás de un examen con libro abierto es que el examen se centra en las habilidades de razonamiento de los estudiantes en lugar de su capacidad para memorizar información específica.

De manera similar, el conocimiento factual se separa de la capacidad de razonamiento del LLM y se almacena en una fuente de conocimiento externa, que puede ser fácilmente accesible y actualizada:

- **Conocimiento paramétrico:** Aprendido durante el entrenamiento y almacenado implícitamente en los pesos de la red neuronal.
- **Conocimiento no paramétrico:** Almacenado en una fuente de conocimiento externa, como una base de datos vectorial.



Allí vemos las tres etapas del RAG:

- **Recuperar:** La consulta del usuario se utiliza para obtener contexto relevante de una fuente de conocimiento externa. Para ello, la consulta del usuario se incrusta con un modelo de embeddings en el mismo espacio vectorial que el contexto adicional en la base de datos vectorial. Esto permite realizar una búsqueda de similitud y se devuelven los k objetos de datos más cercanos de la base de datos vectorial.
- **Aumentar:** La consulta del usuario y el contexto adicional recuperado se introducen en una plantilla de prompt.
- **Generar:** Finalmente, la solicitud aumentada con la recuperación se alimenta al LLM.

Más información:

<https://weaviate.io/blog/introduction-to-rag?s=08>

## Implementación de RAG

Google Colab

🔗 [https://colab.research.google.com/drive/1tOy3QxzJphp\\_4dAyygsAObHgYm5Z-gOL?usp=sharing](https://colab.research.google.com/drive/1tOy3QxzJphp_4dAyygsAObHgYm5Z-gOL?usp=sharing)



A continuación veremos los pasos para implementar un asistente basado en RAG, para crear un sistema de consulta de historia Argentina, a través del lenguaje natural. Nos basaremos en la librería [llama-index](#) y en el modelo [Zephyr de Hugging Face](#) para construir nuestro asistente.

Primero debemos preparar la información de conocimiento, y para ello usaremos documentos PDFs con contenidos de historia Argentina:

```
!pip install gdown

# Obtenemos los PDFs que usaremos como fuente de conocimiento
import gdown
import os
import shutil

# Link con archivos sobre historia Argentina
url = 'https://drive.google.com/drive/folders/10-Q9O6M5uk3kehXHZjdWCoNUO2RMz7

# Descarga carpeta 'Historia Argentina'
gdown.download_folder(url, quiet=True, output='Historia_Argentina')

# Crear la carpeta 'llamaindex_data' si no existe
carpeta_destino = 'llamaindex_data'
if not os.path.exists(carpeta_destino):
    os.makedirs(carpeta_destino)

# Mover todos los archivos de 'Historia Argentina' a 'llamaindex_data'
carpeta_origen = 'Historia_Argentina'
for filename in os.listdir(carpeta_origen):
    ruta_origen = os.path.join(carpeta_origen, filename)
    ruta_destino = os.path.join(carpeta_destino, filename)
    shutil.move(ruta_origen, ruta_destino)

# Eliminar la carpeta 'Historia_Argentina'
shutil.rmtree(carpeta_origen)

print("Archivos movidos con éxito.")
```

Con el script anterior, obtendremos una carpeta con 3 documentos que usaremos como nuestra fuente de datos de conocimiento. Luego debemos extraer el texto de esos documentos, crear los embeddings y almacenarlos en una base de datos vectorial.

Antes de correr el script con nuestra implementación de RAG, además debemos preparar un archivo [.env](#):

```
HUGGINGFACE_TOKEN=xxxxxx_nuestro_token_xxxxxxx
```

De ese modo, guardamos de forma segura nuestra clave de Hugging Face, y que luego podemos recuperar usando `config('HUGGINGFACE_TOKEN')` gracias a la librería `python-decouple`

Ahora corramos nuestro sistema RAG (No olvidemos instalar todas las librerías necesarias, que se mencionan abajo):

```
# !pip install llama-index-embeddings-huggingface==0.1.1 sentence-transformers==2.3
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
import requests
from decouple import config
from llama_index.core import Settings
from google.colab import userdata
from llm_templates import Formatter, Conversation

# Función para aplicar el template de chat usando la librería llm-templates
def zephyr_chat_template(messages, add_generation_prompt=True):
    formatter = Formatter()
    conversation = Conversation(model='zephyr', messages=messages)
    # Renderizar la plantilla con los mensajes proporcionados
    return formatter.render(conversation, add_assistant_prompt=add_generation_prompt)

# Aquí hacemos la llamada al modelo
def generate_answer(prompt: str, max_new_tokens: int = 768) → None:
    try:
        # Tu clave API de Hugging Face. Busca en el .env, luego en los secretos de Google
        api_key = config('HUGGINGFACE_TOKEN', userdata.get('HUGGINGFACE_TOKEN'))

        # URL de la API de Hugging Face para la generación de texto
        api_url = "https://api-inference.huggingface.co/models/HuggingFaceH4/zephyr-7

        # Cabeceras para la solicitud
        headers = {"Authorization": f"Bearer {api_key}"}

        # Datos para enviar en la solicitud POST
        # Sobre los parámetros: https://huggingface.co/docs/transformers/main_classes/t
        data = {
            "inputs": prompt,
            "parameters": {
```

```

        "max_new_tokens": max_new_tokens,
        "temperature": 0.7,
        "top_k": 50,
        "top_p": 0.95
    }
}

# Realizamos la solicitud POST
response = requests.post(api_url, headers=headers, json=data)

# Extraer respuesta
respuesta = response.json()[0]["generated_text"][len(prompt):]
return respuesta

except Exception as e:
    print(f"An error occurred: {e}")

# Esta función prepara el prompt en estilo QA
def prepare_prompt(query_str: str, context_str: str):
    TEXT_QA_PROMPT_TMPL = (
        "La información de contexto es la siguiente:\n"
        "-----\n"
        "{context_str}\n"
        "-----\n"
        "Dada la información de contexto anterior, y sin utilizar conocimiento previo, respon"
        "Pregunta: {query_str}\n"
        "Respuesta: "
    )

    messages = [
        {
            "role": "system",
            "content": "Eres un asistente útil que siempre responde con respuestas veraces,"
        },
        {"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(context_str=context_str)}
    ]

    final_prompt = zephyr_chat_template(messages)
    return final_prompt

# Cargamos nuestro modelo de embeddings
print('Cargando modelo de embeddings...')
model_name = "sentence-transformers/distiluse-base-multilingual-cased-v2"

```

```

#model_name = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
#model_name = "sentence-transformers/LaBSE"
embed_model = HuggingFaceEmbedding(model_name=model_name)

print('Indexando documentos...')
# Carga los documentos de la carpeta y luego genera e indexa los embeddings
documents = SimpleDirectoryReader(input_dir="llamaindex_data").load_data()
# Construimos un índice de documentos a partir de los datos de la carpeta llamaindex_
index = VectorStoreIndex.from_documents(documents, show_progress=True, embed_r

# Construimos un retriever a partir del índice, para realizar la búsqueda vectorial de do
retriever = index.as_retriever(similarity_top_k=2)

print('Realizando llamada a HuggingFace para generar respuestas...\n')

queries = ['¿Que pasó en la crisis del 29?',
           '¿Cuándo se redactó la constitución Argentina?',
           '¿Cuándo fué la revolución de Mayo?',
           '¿Cuándo se declaró la independencia?',
           '¿Cuándo asumió Raúl Alfonsín como presidente?']

for query_str in queries:
    # Traemos los documentos más relevantes para la consulta
    nodes = retriever.retrieve(query_str)

    # Construimos el contexto para usar con el LLM
    context_str = ""
    for node in nodes:
        page_label = node.metadata["page_label"]
        file_path = node.metadata["file_path"]
        context_str += f"\npage_label: {page_label}\n"
        context_str += f"file_path: {file_path}\n\n"
        context_str += f"{node.text}\n"
    #print(f'context_str: {context_str}')

    final_prompt = prepare_prompt(query_str, context_str)
    print('Pregunta:', query_str)
    print('Respuesta:')
    print(generate_answer(final_prompt))
    print('-----')

```

Y aquí vemos el resultado de la ejecución del script:

Cargando modelo de embeddings...

Indexando documentos...

LLM is explicitly disabled. Using MockLLM.

Parsing nodes: 100% |██████████| 1157/1157 [00:07<00:00, 156.84it/s]

Generating embeddings: 100% |██████████| 1663/1663 [07:18<00:00, 3.79it/s]

Realizando llamada a HuggingFace para generar respuestas...

Pregunta: ¿Que pasó en la crisis del 29?

Respuesta:

La crisis mencionada en la información de contexto se refiere a la crisis económica que se produjo en 1929, especialmente después de la caída del presidente Hipólito Yrigoyen. En esa crisis, los sectores empresariales redujeron los salarios, desconocieron las leyes sociales y despidieron a muchos trabajadores, especialmente en la zona de Puerto Nuevo en Buenos Aires, que se conocía como Villa Desocupación y fue eliminada por el municipio en 1932. En ese año, había en la Argentina 333 997 desocupados (Rapoport, 2000).

---

Pregunta: ¿Cuándo se redactó la constitución Argentina?

Respuesta:

La constitución Argentina se sancionó el 1 de mayo de 1853 en la Convención Constituyente que comenzó a sesionar el 20 de noviembre de 1852 en Santa Fe, bajo la presidencia del salteño Facundo Zuviría. Esta información se encuentra en la sección de referencias bibliográficas de la página 241 del documento PDF "Historia-Argentina.pdf".

---

Pregunta: ¿Cuándo fué la revolución de Mayo?

Respuesta:

La pregunta pide identificar el momento en que se produjo la Revolución de Mayo, y se proporciona información de contexto que indica que se inició en 1810. Por lo tanto, la respuesta correcta es: La Revolución de Mayo comenzó en 1810.

---

Pregunta: ¿Cuándo se declaró la independencia?

Respuesta:

La independencia se declaró el 9 de julio de 1816 en San Miguel de Tucumán, según se indica en el texto de contexto proporcionado. Anteriormente, en 1810, se había depuesto al virrey español en Buenos Aires y se formó un gobierno autónomo, pero se seguía ac

---

Pregunta: ¿Cuándo asumió Raúl Alfonsín como presidente?

Respuesta:

El presidente Raúl Alfonsín asumió en 1983. Mencionado eso en la información de contexto en la página 8 y se refiere específicamente a que "El nuevo presidente fue Raúl Alfonsín de la Unión Cívica Radical, quien dispuso investigar los crímenes de lesa humanidad de

dictadura creando la Conadep, ente que produjo un decisivo informe titulado Nunca más.

Como vemos, hemos dotado a nuestro asistente, con conocimiento específico sobre historia Argentina. Del mismo modo, podríamos suministrar información actual o de "tiempo real", incorporando información disponible en servicios web (por ejemplo cotización de monedas, o noticias de último momento).

`VectorStoreIndex` de Llama-Index es una parte importante de la librería, y es una herramienta de indexación y búsqueda eficiente para grandes colecciones de vectores de alta dimensión, y actúa como un contenedor para almacenar vectores. Estos vectores pueden ser, por ejemplo, representaciones vectoriales (embeddings) de textos, imágenes o cualquier otro dato que se pueda convertir en un vector en un espacio de alta dimensión. La herramienta indexa estos vectores para facilitar una búsqueda rápida y eficiente. La indexación implica organizar los vectores de una manera que acelere la recuperación de información.

Cuando se realiza una consulta de búsqueda (que también se representa como un vector), `VectorStoreIndex` busca en su índice los vectores más cercanos o más relevantes a la consulta. Esto generalmente se hace utilizando la similitud del coseno.

Llama-Index, nos facilita el splitting de los documentos y la posterior búsqueda semántica de los trozos de texto relevantes (`nodes`).

Como método para el cálculo de embeddings, hemos utilizado el modelo '[sentence-transformers/distiluse-base-multilingual-cased-v2](#)', que soporta español, pero podríamos haber seleccionado cualquier otro modelo disponible.

## Otros modelos

A continuación, mejoramos el código para probar otro modelo de lenguaje alternativo [mistralai/Mixtral-8x7B-Instruct-v0.1](#). Como podemos ver en ese caso, no utilizamos el rol del sistema, ya que el template de chat de ese modelo no lo soporta:

```
# !pip install llama-index-embeddings-huggingface==0.1.1 sentence-transformers==2.3.1
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
import requests
from decouple import config
from llama_index.core import Settings
from google.colab import userdata
from llm_templates import Formatter, Conversation

# Modelos LLM para testear
# LLM_MODEL = {'model': 'HuggingFaceH4/zephyr-7b-beta', 'system_role': True, 'tem
```

```

LLM_MODEL = {'model': 'mistralai/Mixtral-8x7B-Instruct-v0.1', 'system_role': False, 'temperature': 0.7, 'max_new_tokens': 768}

HUGGINGFACE_TOKEN = config('HUGGINGFACE_TOKEN', userdata.get('HUGGINGFACE_TOKEN'))

# Función para aplicar el template de chat usando la librería l1m-templates
def chat_template(messages, add_generation_prompt=True):
    formatter = Formatter(huggingface_api_key=HUGGINGFACE_TOKEN)
    conversation = Conversation(model=LLM_MODEL['template'], messages=messages)
    # Renderizar la plantilla con los mensajes proporcionados
    return formatter.render(conversation, add_assistant_prompt=add_generation_prompt)

# Aquí hacemos la llamada el modelo
def generate_answer(prompt: str, max_new_tokens: int = 768) → None:
    try:
        # URL de la API de Hugging Face para la generación de texto
        api_url = f"https://api-inference.huggingface.co/models/{LLM_MODEL['model']}"

        # Cabeceras para la solicitud
        headers = {"Authorization": f"Bearer {HUGGINGFACE_TOKEN}"}

        # Datos para enviar en la solicitud POST
        # Sobre los parámetros: https://huggingface.co/docs/transformers/main_classes/text_generation
        data = {
            "inputs": prompt,
            "parameters": {
                "max_new_tokens": max_new_tokens,
                "temperature": 0.7,
                "top_k": 50,
                "top_p": 0.95
            }
        }

        # Realizamos la solicitud POST
        response = None
        response = requests.post(api_url, headers=headers, json=data)

        # Extraer respuesta
        respuesta = response.json()[0]["generated_text"][len(prompt):]
        return respuesta
    except Exception as e:
        print(f"An error occurred: {e}: {response}")

```

```

# Esta función prepara el prompt en estilo QA
def prepare_prompt(query_str: str, context_str: str):
    TEXT_QA_PROMPT_TMPL = (
        "La información de contexto es la siguiente:\n"
        "-----\n"
        "{context_str}\n"
        "-----\n"
        "Dada la información de contexto anterior, y sin utilizar conocimiento previo, respon"
        "Pregunta: {query_str}\n"
        "Respuesta: "
    )

    messages = []
    # Si el modelo soporta el rol del sistema, agregamos el mensaje
    if LLM_MODEL['system_role']:
        messages = [
            {
                "role": "system",
                "content": "Eres un asistente útil que siempre responde con respuestas veraces"
            },
        ]
    # Agregamos el mensaje del usuario con el contexto
    messages.append({"role": "user", "content": TEXT_QA_PROMPT_TMPL.format(context=context_str, query=query_str)})

    final_prompt = chat_template(messages)
    return final_prompt

# Cargamos nuestro modelo de embeddings
print('Cargando modelo de embeddings...')
model_name = "sentence-transformers/distiluse-base-multilingual-cased-v2"
#model_name = "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
#model_name = "sentence-transformers/LaBSE"
embed_model = HuggingFaceEmbedding(model_name=model_name)

# Construimos un retriever a partir del índice, para realizar la búsqueda vectorial de documentos
retriever = index.as_retriever(similarity_top_k=2)

print('Realizando llamada a HuggingFace para generar respuestas...\n')

queries = ['¿Que pasó en la crisis del 29?',
           '¿Cuándo se redactó la constitución Argentina?',
           '¿Cuándo fué la revolución de Mayo?']

```

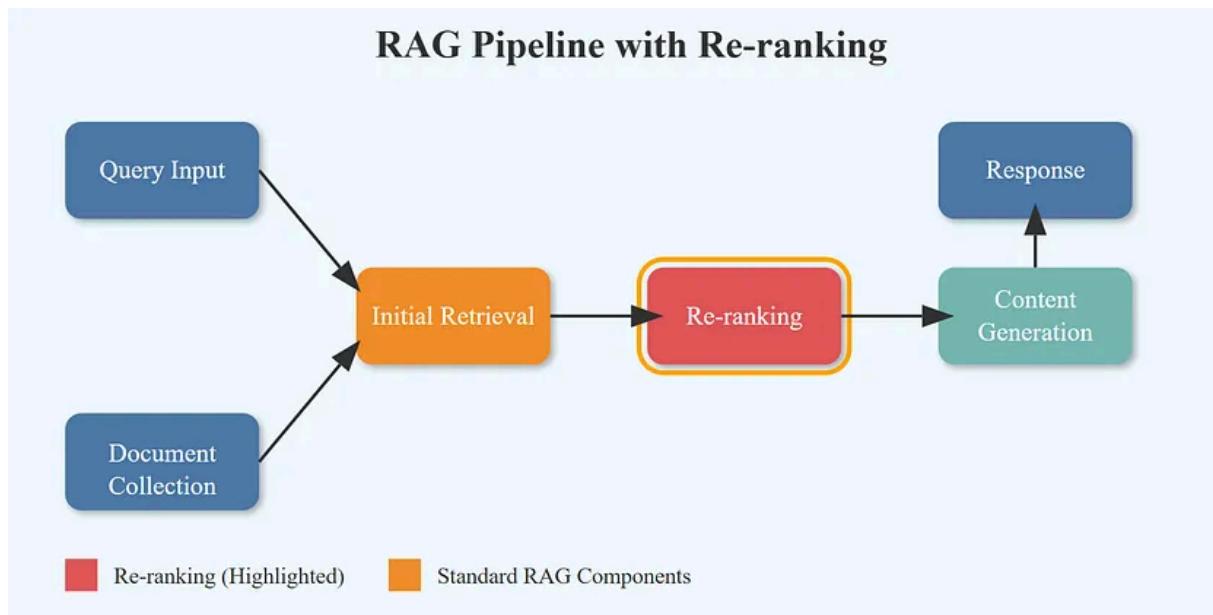
```
'¿Cuándo se declaró la independencia?',  
'¿Cuándo asumió Raúl Alfonsín como presidente?']
```

```
for query_str in queries:  
    # Traemos los documentos más relevantes para la consulta  
    nodes = retriever.retrieve(query_str)  
  
    # Construimos el contexto para usar con el LLM  
    context_str = ""  
    for node in nodes:  
        page_label = node.metadata["page_label"]  
        file_path = node.metadata["file_path"]  
        context_str += f"\npage_label: {page_label}\n"  
        context_str += f"file_path: {file_path}\n\n"  
        context_str += f"{node.text}\n"  
    #print(f'context_str: {context_str}')  
  
    final_prompt = prepare_prompt(query_str, context_str)  
    print('Pregunta:', query_str)  
    print('Respuesta:')  
    print(generate_answer(final_prompt))  
    print('-----')
```

## 5. Re-ranking

El uso de sistemas de Recuperación de Información Generativa (RAG) ha revolucionado la manera en que se combina la generación de texto con la recuperación de documentos relevantes. Estos sistemas integran modelos de lenguaje preentrenados con motores de búsqueda, permitiendo respuestas más informadas y contextualizadas en tareas como preguntas y respuestas, soporte técnico y generación de contenido. Sin embargo, a pesar de su potencial, los sistemas RAG enfrentan desafíos inherentes que limitan su eficacia en ciertas aplicaciones.

Entre estas limitaciones destacan problemas relacionados con la relevancia de los documentos recuperados, la calidad de las respuestas generadas, y la coherencia entre los datos recuperados y el texto generado. Es en este contexto donde el proceso de **re-ranking** emerge como una técnica complementaria que busca optimizar la selección de documentos relevantes antes de la etapa de generación.

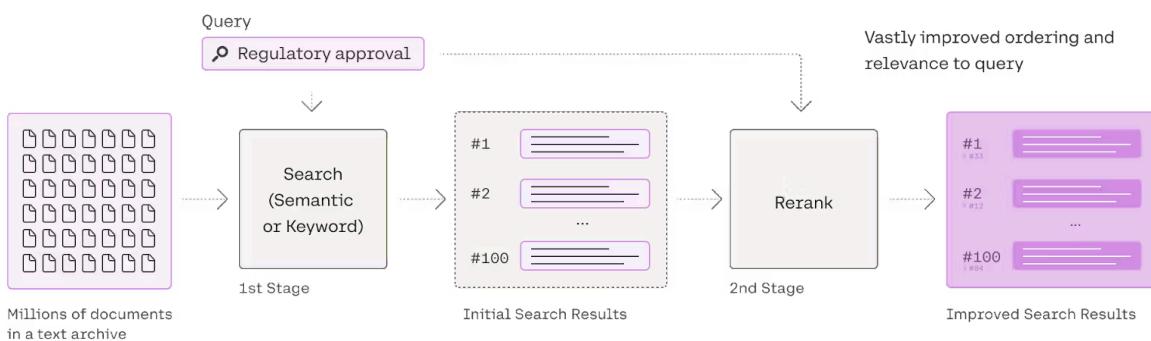


Mediante el uso de modelos más avanzados o especializados para reordenar los documentos inicialmente recuperados, el re-ranking mejora la calidad del input que recibe el modelo generativo, potenciando así los resultados finales.

#### Entendiendo la magia del re-ranking

Entonces, ¿qué es exactamente el re-ranking y por qué debería importarte? Piensa en ello como darle una segunda mirada más exhaustiva a tus resultados de búsqueda. Así es como funciona típicamente:

1. **Recuperación inicial:** Tu sistema obtiene un grupo de documentos potencialmente relevantes utilizando métodos tradicionales rápidos.
2. **Re-ranking:** Aquí es donde ocurre la magia. Un modelo más específico para la tarea, examina más de cerca estos documentos, considerando aspectos como el contexto, la intención y las relaciones matizadas.
3. **Resultados nuevos y mejorados:** Basándose en este análisis más profundo, los documentos se reordenan, llevando los más relevantes a la parte superior.



Es como tener un asistente inteligente que no solo encuentra información para nosotros, sino que también la filtra y organiza según lo que necesitamos.

### **Por qué los métodos tradicionales se quedan cortos**

Ahora bien, podrías preguntarte: "¿Por qué no podemos seguir usando los métodos tradicionales?" Los métodos tradicionales de recuperación, aunque rápidos y eficientes, tienen sus limitaciones. A menudo se basan en coincidencias simples de palabras clave o en similitudes semánticas básicas. Es como tratar de resolver un rompecabezas complejo con solo la mitad de las piezas: puedes tener una idea general, pero faltan detalles cruciales.

Estos métodos enfrentan dificultades como:

- Comprender el contexto y la intención del usuario.
- Manejar consultas complejas y multifacéticas.
- Identificar relaciones sutiles entre conceptos.

Y seamos sinceros, en el mundo real, las preguntas rara vez son tan simples como "¿Cuál es la capital de Francia?".

### **Re-ranking al rescate: un ejemplo práctico**

Para hacerlo más concreto, imaginemos que estás desarrollando un asistente de investigación médica que necesita responder consultas complejas sobre estudios recientes. Un investigador pregunta:

"¿Cuáles son los hallazgos más recientes sobre los efectos a largo plazo del ayuno intermitente en la salud cardiovascular?"

Un sistema tradicional podría devolver:

1. Un artículo general sobre el ayuno intermitente.
2. Un estudio sobre los efectos a corto plazo del ayuno.
3. Un resumen general sobre la salud cardiovascular.

Pero un sistema de re-ranking iría un paso más allá:

1. Primero, recupera un conjunto más amplio de estudios y artículos potencialmente relevantes.
2. Luego, analiza estos documentos con mayor profundidad, buscando conexiones entre el ayuno intermitente, los efectos a largo plazo y la salud cardiovascular.
3. Finalmente, reordena los resultados, priorizando los estudios recientes que aborden específicamente los efectos a largo plazo del ayuno intermitente en la salud cardiovascular.

¿El resultado? Una respuesta mucho más precisa y útil que aborda directamente la consulta específica del investigador, destacando estudios que un sistema tradicional podría haber relegado en los resultados.

## Tipos de re-rankers

Los rerankers se han utilizado durante años, pero el campo está evolucionando rápidamente. Examinemos las opciones actuales y en qué se diferencian.

Modelo	Tipo	Rendimiento	Costo	Ejemplo
Cross-encoder	Código abierto	Excelente	Medio	BGE, sentence, transformers, Mixedbread
Multi-vector	Código abierto	Bueno	Bajo	COLBERT
LLM	Código abierto	Excelente	Alto	RankZephyr, RankT5
LLM API	Privado	El mejor	Muy alto	GPT, Claude
Rerank API	Privado	Excelente	Medio	Cohere, Mixedbread, Jina

### Cross-Encoders

Los modelos Cross-Encoder redefinen el enfoque convencional al emplear un mecanismo de clasificación para pares de datos. El modelo toma un par de datos, como dos oraciones, como entrada y produce un valor entre 0 y 1 que indica la similitud entre ambos elementos. Este enfoque, que se aleja de las incrustaciones vectoriales, permite una comprensión más matizada de las relaciones entre los puntos de datos.

Es importante destacar que los cross-encoders requieren un par de "elementos" para cada entrada, lo que los hace inadecuados para manejar oraciones individuales de forma independiente. En el contexto de búsqueda, un cross-encoder se utiliza con cada elemento de datos y la consulta de búsqueda para calcular la similitud entre la consulta y el objeto de datos.

A continuación, se muestra un fragmento de código sobre cómo utilizar rerankers de última generación como BGE.

```
# Primero instalamos la biblioteca necesaria
!pip install FlagEmbedding

# Importamos la librería
from FlagEmbedding import FlagReranker

# Inicializamos el reranker
reranker = FlagReranker('BAAI/bge-reranker-v2-m3', use_fp16=True)

# Ejemplo simple con una consulta y un pasaje
consulta = "¿Qué es un oso panda?"
```

```

pasaje = "El oso panda gigante es un animal que vive en China."
score = reranker.compute_score([consulta, pasaje])
print(f"Puntuación para consulta simple: {score}")

# Ejemplo con múltiples pares de consulta-pasaje
consultas_y_pasajes = [
    ['¿Qué es un oso panda?', 'Hola, ¿cómo estás?'],
    ['¿Qué es un oso panda?', 'El oso panda gigante (Ailuropoda melanoleuca), también']
]

puntuaciones = reranker.compute_score(consultas_y_pasajes)
print("\nPuntuaciones para múltiples pares:")
for i, score in enumerate(puntuaciones):
    print(f"Par {i+1}: {score}")

```

## Re-rankers Multi-Vector

Los cross-encoders ofrecen un rendimiento excelente, pero ¿qué pasa con las opciones alternativas que requieren menos capacidad de cómputo?

Modelos de incrustación multi-vector como ColBERT utilizan una interacción tardía, donde la interacción entre las representaciones de la consulta y del documento ocurre en etapas finales del proceso, después de que ambas se han codificado de manera independiente. Este enfoque contrasta con los modelos de interacción temprana, como los cross-encoders, en los que las incrustaciones de la consulta y del documento interactúan en etapas anteriores, lo que puede aumentar la complejidad computacional. Por otro lado, en el caso de la similitud coseno de las incrustaciones para recuperación, ¡no hay interacción en absoluto!

El diseño de interacción tardía permite la precomputación de las representaciones de los documentos, lo que contribuye a tiempos de recuperación más rápidos y a una reducción de las demandas computacionales, haciéndolo adecuado para procesar grandes colecciones de documentos.

ColBERT ofrece lo mejor de ambos mundos. En el link [blog de Jina](#), se puede ampliar sobre esta técnica y código para implementarla.

## LLMs para Reranking

A medida que los Modelos de Lenguaje Extensos (LLMs) crecen en tamaño, superando los 10 mil millones de parámetros, el ajuste fino del modelo de reranking se vuelve cada vez más desafiante. Los esfuerzos recientes han buscado abordar este problema utilizando estrategias de *prompting* que permiten a los LLMs mejorar el reranking de documentos de manera autónoma. En términos generales, estas estrategias de *prompting* se dividen en tres categorías: métodos *pointwise*, *listwise* y *pairwise*.

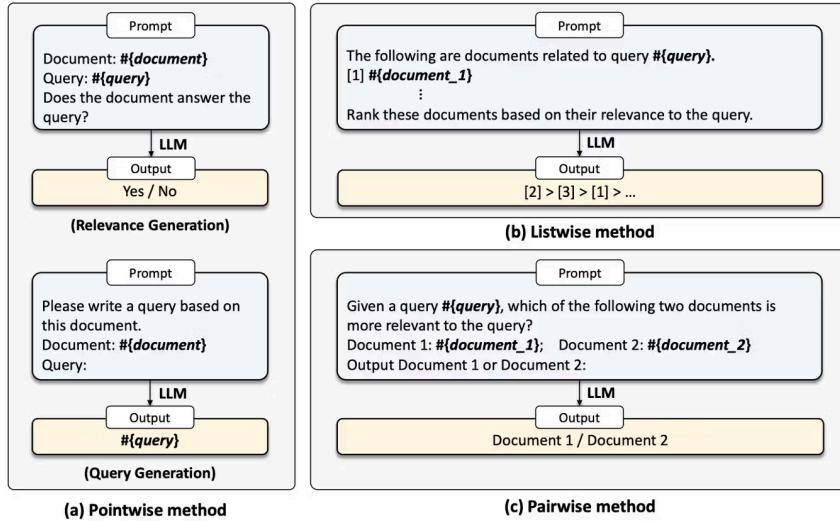


Fig. 6. Three types of unsupervised reranking methods: (a) pointwise methods that consist of relevance generation (upper) and query generation (lower), (b) listwise methods, and (c) pairwise methods.

Fuente: <https://arxiv.org/abs/2308.07107>

- **Métodos Pointwise:** Los métodos *pointwise* miden la relevancia entre una consulta (query) y un único documento. Las subcategorías incluyen la generación de relevancia y la generación de consultas, ambas efectivas para el reranking de documentos en escenarios de *zero-shot*.
- **Métodos Listwise:** Los métodos *listwise* clasifican directamente una lista de documentos insertando la consulta y la lista de documentos en el *prompt* y solicitando a los LLMs que generen los identificadores de los documentos rerankeados. Debido a la longitud limitada de entrada de los LLMs, insertar todos los documentos candidatos en el *prompt* no es factible. Para abordar este problema, estos métodos emplean una estrategia de ventana deslizante (*sliding window*) para rerankear un subconjunto de documentos candidatos en cada iteración. Esto implica clasificar de atrás hacia adelante utilizando una ventana deslizante y rerankear solo los documentos dentro de la ventana en cada paso.
- **Métodos Pairwise:** En los métodos *pairwise*, los LLMs reciben un *prompt* que contiene una consulta y un par de documentos. Luego, se les indica que generen el identificador del documento que consideren más relevante. Métodos de agregación como *AllPairs* se emplean para rerankear todos los documentos candidatos. *AllPairs* genera inicialmente todos los pares de documentos posibles y consolida una puntuación final de relevancia para cada documento. Algoritmos de ordenamiento eficientes como *heap sort* y *bubble sort* se utilizan típicamente para acelerar el proceso de clasificación. Esto puede lograrse con la ayuda de GPT-3.5, GPT-4 y GPT-4-Turbo, siendo este último el que ofrece el mejor equilibrio entre costo y rendimiento.

## Rerankers Supervisados con LLMs

El ajuste fino supervisado es un paso crucial al aplicar LLMs preentrenados a tareas de reranking. Debido a la falta de conciencia sobre el ranking durante el preentrenamiento, los LLMs no pueden medir con precisión la relevancia entre consultas y documentos. El ajuste fino en conjuntos de datos específicos para ranking, como el conjunto de datos de *MS MARCO passage ranking*, permite que los LLMs ajusten sus parámetros para mejorar el rendimiento en reranking. Los rerankers supervisados pueden categorizarse en dos tipos según la estructura del modelo base.

Los modelos *encoder-only* son la opción más común y adecuada para tareas de reranking debido a las siguientes razones:

- **Atención bidireccional mejorada:** Permite que cada token de la consulta y el documento se relacionen completamente entre sí, lo que es esencial para comprender las relaciones contextuales en tareas de clasificación/ranking.
- **Eficiencia para el cálculo de scoring:** Los modelos *encoder-only* suelen ser más eficientes en términos computacionales, especialmente al procesar múltiples pares consulta-documento.
- **Adaptación para clasificación/ranking:** Están específicamente diseñados para tareas que requieren analizar y clasificar entradas con base en su relevancia.

La categorización de los modelos de reranking basados en LLMs son las siguientes:

- *Encoder-only* (la opción principal y más eficiente).
- *Encoder-Decoder* (para casos específicos, como tareas que requieren generación además de clasificación).
- *Decoder-only* (menos común y utilizado en casos especiales). Es mucho más precisa y refleja mejor el uso práctico de estas arquitecturas en sistemas de reranking. Si bien los modelos *decoder-only* (como GPT) son extremadamente potentes, su diseño los hace menos eficientes para reranking puro debido a que están optimizados para tareas de generación de texto. Su uso en reranking se limita generalmente a enfoques experimentales o casos muy específicos donde se requiere generación y clasificación combinadas.

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from typing import List, Dict
import pandas as pd
from IPython.display import display, HTML

class MultilingualRankLLM:
    """
    Implementación de RankLLM usando un modelo multilingüe.
    """

```

```

def __init__(self, model_name: str = "intfloat/multilingual-e5-large"):
    """
    Inicializa el modelo con un checkpoint multilingüe.
    Alternativas posibles:
    - 'intfloat/multilingual-e5-large'
    - 'sentence-transformers paraphrase-multilingual-mpnet-base-v2'
    - 'microsoft/mdeberta-v3-base'
    """

    self.tokenizer = AutoTokenizer.from_pretrained(model_name)
    self.model = AutoModelForSequenceClassification.from_pretrained(
        model_name,
        num_labels=1,
        torch_dtype=torch.float16
    )
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Usando dispositivo: {self.device}")
    self.model.to(self.device)

def prepare_input(self, query: str, document: str) → Dict:
    """
    Prepara el input para el modelo multilingüe.
    """

    # Formato específico para E5: "query: [consulta] document: [documento]"
    texto_combinado = f"query: {query} document: {document}"
    return self.tokenizer(
        texto_combinado,
        truncation=True,
        max_length=512,
        padding="max_length",
        return_tensors="pt"
    )

def rank_documents(self,
                  query: str,
                  documents: List[str],
                  batch_size: int = 8) → List[Dict]:
    """Rankear una lista de documentos para una consulta dada."""
    self.model.eval()
    results = []

    # Procesar documentos en batches
    for i in range(0, len(documents), batch_size):
        batch_docs = documents[i:i + batch_size]

```

```

batch_inputs = [
    self.prepare_input(query, doc) for doc in batch_docs
]

# Concatenar inputs del batch
batch_combined = {
    key: torch.cat([inp[key] for inp in batch_inputs])
    for key in batch_inputs[0].keys()
}

# Mover a GPU/CPU según corresponda
batch_combined = {
    k: v.to(self.device) for k, v in batch_combined.items()
}

# Obtener predicciones
with torch.no_grad():
    outputs = self.model(**batch_combined)
    scores = outputs.logits.squeeze(-1).cpu().numpy()

# Guardar resultados
for doc, score in zip(batch_docs, scores):
    results.append({
        'documento': doc,
        'puntuación': float(score)
    })

# Ordenar por puntuación
results.sort(key=lambda x: x['puntuación'], reverse=True)
return results

def visualizar_resultados(resultados: List[Dict]):
    """Visualiza los resultados del ranking."""
    df = pd.DataFrame(resultados)

    # Crear resumen de documentos
    df['documento_resumen'] = df['documento'].str[:100] + '...'

    # Normalizar puntuaciones para mejor visualización
    max_score = df['puntuación'].max()
    min_score = df['puntuación'].min()
    df['puntuación_norm'] = (df['puntuación'] - min_score) / (max_score - min_score)

```

```

# DataFrame para visualización
df_display = pd.DataFrame({
    'Documento': df['documento_resumen'],
    'Puntuación': df['puntuación'].round(4),
    'Relevancia': (df['puntuación_norm'] * 100).round(2).astype(str) + '%'
})

display(HTML("<h3>Resultados del Ranking</h3>"))
display(df_display.style
    .background_gradient(subset=['Puntuación'], cmap='YlOrRd')
    .format({'Puntuación': '{:.4f}'})
)

print("\nDocumentos completos por orden de relevancia:")
for i, (doc, score, rel) in enumerate(zip(df['documento'],
                                            df['puntuación'],
                                            df['puntuación_norm']), 1):
    print(f"\n{i}. Puntuación: {score:.4f} (Relevancia: {rel*100:.1f}%)")
    print(f"Documento: {doc}")

def ejemplo_uso():
    """Ejemplo de uso con textos en español."""
    print("Inicializando RankLLM multilingüe...")
    ranker = MultilingualRankLLM("intfloat/multilingual-e5-large")
    # Alternativas:
    # ranker = MultilingualRankLLM("sentence-transformers/paraphrase-multilingual-mmpc")
    # ranker = MultilingualRankLLM("microsoft/mdeberta-v3-base")

    # Ejemplo con consulta y documentos en español
    query = "¿Cuáles son los beneficios de la energía solar?"
    documents = [
        "La energía solar es una fuente renovable que reduce significativamente las emisiones de CO2.",
        "Los pingüinos emperadores viven exclusivamente en la Antártida y pueden bucear hasta 500 metros de profundidad.",
        "Los paneles solares pueden reducir considerablemente la factura eléctrica y proporcionar una fuente de energía limpia.",
        "La receta tradicional de paella valenciana incluye arroz, azafrán, conejo y garrofón.",
        "La instalación de sistemas solares genera empleos locales y contribuye al desarrollo económico de las comunidades rurales."
    ]

    print("\nProcesando consulta y documentos en español...")
    resultados = ranker.rank_documents(query, documents)

    print("\nVisualizando resultados...")
    visualizar_resultados(resultados)

```

```
if __name__ == "__main__":
    ejemplo_uso()
```