

eclap.adrformacion.com © ADR Infor SL
JUAN ALBERTO AZEVEDO IBÁÑEZ

Gestión de la conversación y contexto del chatbot © ADR Infor SL

eclap.adrformacion.com © ADR Infor SL
JUAN ALBERTO AZEVEDO IBÁÑEZ

eclap.adrformacion.com © ADR Infor SL
JUAN ALBERTO AZEVEDO IBÁÑEZ

Indice

Competencias y Resultados de Aprendizaje desarrollados en esta unidad	3
Gestión de la conversación y contexto del chatbot	4
Pasos para integrar spaCy en el chatbot	4
Configurar spaCy en tu proyecto Django	4
Modificar logic_chatbot.py para usar spaCy	4
Integrando spaCy para mejorar las respuestas	5
Modificar nlp_utils.py para reconocer saludos y otras intenciones dinámicamente	6
Modificar chatbot_logic.py para usar la detección dinámica	8
Crear el flujo conversacional	8
nlp_utils.py	9
Explicación de las funciones	11
chatbot_logic.py	11
Explicación de la lógica general	13
Y por último el archivo views.py	14
Explicación del código	15
Posibles errores y cómo solventarlos	16
Error de fechas	16
Error de comensales	18
Resumen	19
Actividades prácticas	20

Competencias y Resultados de Aprendizaje desarrollados en esta unidad

Competencia:

Diseñar, implementar y gestionar chatbots inteligentes e interactivos en aplicaciones web utilizando Django y Python, integrando habilidades de procesamiento de lenguaje natural para mejorar la experiencia del usuario y optimizar la interacción entre usuarios y chatbots.

Resultados de Aprendizaje:

- Implementar la lógica del chatbot en views.py y vincularla con la interfaz de usuario a través de plantillas HTML y CSS.
- Integrar la biblioteca spaCy para mejorar el procesamiento de texto y la comprensión del contexto de la conversación.
- Implementar un sistema de estados en el chatbot para gestionar el flujo de la conversación de forma dinámica.

Gestión de la conversación y contexto del chatbot

Ya que tenemos control sobre la lógica del chatbot, integrar un procesador de lenguaje natural como spaCy permitirá mejorar la comprensión de los mensajes, hacer reconocimiento de entidades y aplicar modelos de NLP más avanzados.

Pasos para integrar spaCy en el chatbot

En la unidad anterior vimos como crear un chatbot muy sencillo que nos permitía realizar una serie de preguntas muy concretas, además las preguntas tenían que estar escritas de una forma totalmente correcta tanto ortográfica como gramaticalmente, por lo que vamos a mejorar el chatbot para que use procesamiento de lenguaje natural (NLP) con spaCy y pueda responder de manera más flexible.

Instalación de spaCy y el modelo de idioma

Si aún no lo tienes instalado, en la terminal ejecuta:

```
pip install spacy
python -m spacy download es_core_news_sm # Para español
```

Esto instala el modelo de idioma que spaCy usará para analizar texto.

Configurar spaCy en tu proyecto Django

En una aplicación de Django, puedes crear un archivo `nlp_utils.py` para manejar el procesamiento:

```
import spacy

nlp = spacy.load("es_core_news_sm") # O "en_core_web_sm" si es en inglés

def analizar_texto(texto):
    """Analiza el texto y extrae la información clave"""
    doc = nlp(texto)
    entidades = [(ent.text, ent.label_) for ent in doc.ents]#Entidades nombradas
    palabras_clave = [token.lemma_ for token in doc if not token.is_stop and not token.is_punct]#Palabras clave
    return {"entidades": entidades, "palabras_clave": palabras_clave}
```

Modificar `logic_chatbot.py` para usar spaCy

Ahora importamos `analizar_texto` en `logic_chatbot.py` y lo integramos en `get_response`:

from .nlp_utils import analizar_texto #recuerda poner : para que la búsqueda sea relativa al archivo, los dos deben estar en la misma carpeta

```
def get_response(user_message):

    """Devuelve una respuesta según el mensaje del usuario."""

    user_message = user_message.lower()

    resultado_nlp = analizar_texto(user_message)

    responses = {

        "hola": "¡Hola! ¿Cómo puedo ayudarte?",

        "adiós": "¡Hasta luego! Que tengas un buen día.",

        "cómo estás": "Soy solo un programa, ¡pero gracias por preguntar!",

        "qué puedes hacer": "Puedo responder preguntas básicas. Inténtalo :)",

    }

    if user_message in responses:

        return responses[user_message]

    # Si detectamos entidades, podemos usarlas para mejorar la respuesta

    if resultado_nlp["entidades"]:

        return f"He detectado estas entidades en tu mensaje: {resultado_nlp['entidades']}"

    return "No entiendo tu mensaje, ¿puedes reformularlo?"
```

Integrando spaCy para mejorar las respuestas



Integrado spaCy para mejorar las respuestas

En el modelo anterior, spaCy lo estamos usando principalmente para procesar el texto y obtener información más detallada sobre él, como tokens (palabras), entidades, y parte del discurso. Sin embargo, en ese caso, el modelo no está aprovechando todas las capacidades de spaCy, ya que el código simplemente trata el texto como una cadena de caracteres y lo compara directamente con las respuestas predefinidas. Para otorgar a nuestro chatbot una mayor capacidad de procesamiento de lenguaje natural, debemos integrar spaCy para que no solo entienda los textos tal como son, sino que los interprete en su contexto.

¿Por qué usar spaCy?

La ventaja de spaCy es que puede entender el contexto de las palabras y hacer cosas como:

- Identificar entidades (fechas, lugares, nombres, etc.)
- Detectar la parte del discurso (sustantivos, verbos, etc.)
- Procesar el lenguaje natural de forma más avanzada (reconocimiento de intenciones, similaridad semántica, etc.).

Sin embargo, en el modelo original, simplemente usa un diccionario de respuestas basado en coincidencias exactas, que no aprovecha las capacidades avanzadas de spaCy.

¿Cómo mejora spaCy el modelo?

- Detección de intenciones: Usar spaCy para identificar si el mensaje del usuario es un saludo, una pregunta o cualquier otra intención (en lugar de hacer coincidir el texto exactamente).
- Tratamiento de texto más preciso: Si el usuario escribe con errores, usa sinónimos o variaciones de palabras, spaCy puede procesar y comprender esas diferencias.
- Similitud semántica: Podemos usar spaCy para calcular la similitud semántica entre el mensaje del usuario y las respuestas posibles, lo que permite una respuesta más flexible.

Modificar nlp_utils.py para reconocer saludos y otras intenciones dinámicamente

Usaremos spaCy para detectar diferentes variaciones de saludos, preguntas, etc., y luego usar similitud semántica para determinar qué tipo de mensaje es.

Vamos a agregar un enfoque donde:

- Utilizamos un conjunto de palabras clave relacionadas con saludos, preguntas, y despedidas.
- SpaCy procesará el texto y buscará similitudes o coincidencias con estas intenciones.

```

import spacy

nlp = spacy.load("es_core_news_md")

# Palabras clave para saludos, preguntas y despedidas
INTENTOS = {
    "saludo": ["hola", "buenos días", "buenas tardes", "qué tal", "saludos", "cómo estás", "cómo te va", "qué onda"],
    "despedida": ["adiós", "hasta luego", "nos vemos", "chao", "hasta pronto", "cuídate"],
    "pregunta": ["cómo", "qué", "por qué", "dónde", "quién", "cuándo"],
}

def analizar_texto(texto):
    """Procesa el texto con spaCy y detecta la intención (saludo, pregunta, etc.)."""
    # Procesar el texto con spaCy
    doc = nlp(texto.lower())
    # Analizamos si alguna palabra del texto coincide con las palabras clave para cada intención
    for token in doc:
        for intent, keywords in INTENTOS.items():
            if token.text in keywords:
                return intent

    # Si no encontramos coincidencias exactas, usamos similitud semántica
    for intent, keywords in INTENTOS.items():
        for keyword in keywords:
            keyword_doc = nlp(keyword)
            similarity = doc.similarity(keyword_doc)
            if similarity > 0.7: # Si la similitud es mayor a 0.7, lo consideramos una coincidencia
                return intent
    return "desconocido" # Si no detectamos ninguna intención

```

¿Qué hace este código?

Palabras clave

Define un conjunto de **palabras clave** para diferentes intenciones como "saludo", "despedida", "pregunta", etc. Las puedes ampliar o modificar según necesites.

Procesamiento del texto

Usa **spaCy** para procesar el mensaje del usuario.

Búsqueda de coincidencias

Primero, busca coincidencias exactas con las palabras clave definidas. Si no encuentra coincidencias exactas, **compara la similitud semántica** entre las palabras clave y el mensaje del usuario usando el **modelo de spaCy**.

Similitud semántica

Si una palabra del mensaje tiene una similitud mayor a 0.7 con alguna de las palabras clave, el chatbot lo considera como una coincidencia.

Modificar chatbot_logic.py para usar la detección dinámica

Ahora modificamos el archivo chatbot_logic.py para que use esta lógica y responda de forma dinámica según la intención detectada:

```
from .nlp_utils import analizar_texto

def get_response(user_message):
    """Devuelve una respuesta según la intención detectada dinámicamente."""
    intent = analizar_texto(user_message)
    # Respuestas para diferentes intenciones
    responses = {
        "saludo": "¡Hola! ¿Cómo puedo ayudarte?",
        "despedida": "¡Hasta luego! Que tengas un buen día.",
        "pregunta": "¡Parece que tienes una pregunta! ¿En qué puedo ayudarte?",
        "desconocido": "No entiendo tu mensaje, ¿puedes reformularlo?",
    }
    return responses.get(intent, "No entiendo tu mensaje, ¿puedes reformularlo?")
```

Crear el flujo conversacional

En la siguiente parte de la unidad vamos a crear un flujo conversacional para nuestro chatbot, este flujo puede ser extrapolable a las necesidades particulares de cada chatbot que queramos realizar, al ser solo un ejemplo veremos como crearlo y que posibles problemas podemos afrontar.

El chatbot va realizar la gestión de un restaurante, solo la parte conversacional, es decir no se va a encargar de realizar el guardado de fechas o gestión de mesas, simplemente la parte donde el chat gestiona la conversación del restaurante.

Básicamente la conversación será algo así:



Tú: Hola

Bot: Hola, ¿en qué puedo ayudarte hoy? ¿Menú o reserva?

Tú: ver el menú

Bot: 🍷🍷 Aquí tienes el menú: - Ensalada mixta - Pizza margarita - Pasta carbonara - Tarta de queso. ¿Quieres hacer una reserva?

Tú: si

Bot: Genial. ¿Para qué día quieres la reserva?

Tú: 6 Junio

Bot: ¿Para cuántas personas será la reserva?

Tú: 2

Bot: ¿A nombre de quién estará la reserva?

Tú: Antonio

Bot: ¿Hay algo que debamos tener en cuenta? Alergias, niños, etc.

Tú: no

Bot: Reserva completa para 2 personas el 6 junio, a nombre de antonio. Detalles: no.

¿Deseas hacer otra cosa?

Tú: no

Bot: ¡Reserva anotada! Aquí los detalles: Reserva para 2 personas el 6 junio, a nombre de antonio. Detalles: no.

Tú: Adiós

Bot: Perfecto. Recuerda: Reserva para 2 personas el 6 junio, a nombre de antonio. Detalles: no.

Como puedes ver el flujo conversacional es bastante fluido y natural, así que vamos a trabajar en ello.

Lo que tenemos que hacer es ajustar los tres archivos principales de nuestro chatbot, `nlp_utils.py`, `chatbot_logic.py` y `views.py`

Estos tres archivos junto con el template y los archivos estáticos `.js` y `.css` son los encargados de gestionar el chatbot, el template no lo vamos a modificar pues no es necesario ni el objeto de este curso, así que directamente vamos al trabajo

`nlp_utils.py`

Importación de librerías necesarias:

```
import spacy # Librería para el procesamiento del lenguaje natural
import re # Librería para expresiones regulares, usada para limpiar el texto
import dateparser # Librería para el análisis y parseo de fechas en texto
```

```
# Cargamos el modelo de lenguaje en español de spaCy
nlp = spacy.load("es_core_news_md")
```

```
# Definimos una lista de posibles saludos que el bot puede reconocer
SALUDOS = ["hola", "buenos días", "buenas tardes", "qué tal", "hey", "saludos"]
```

```
# Definimos una lista de posibles afirmaciones que el bot puede reconocer
AFIRMACIONES = ["sí", "claro", "vale", "de acuerdo", "por supuesto", "sí quiero", "ok"]
```

Función para detectar la intención del usuario en base al texto introducido

```
def detectar_intencion(texto):
    """
    Esta función detecta si el texto introducido por el usuario corresponde a un saludo.
    Si se detecta un saludo, devuelve 'saludo'; de lo contrario, devuelve 'desconocida'.
    """
    texto = texto.lower() # Convertimos el texto a minúsculas para normalizar la comparación
    for saludo in SALUDOS:
        if saludo in texto: # Si uno de los saludos está presente en el texto
            return "saludo" # Retornamos 'saludo' si se detecta un saludo
    return "desconocida" # Si no se detecta un saludo, retornamos 'desconocida'
```

Función para determinar si el texto del usuario contiene una afirmación

```
def es_afirmacion(texto):
    """
    Esta función verifica si el texto del usuario contiene alguna afirmación reconocida.
    Si el texto contiene alguna de las palabras en la lista AFIRMACIONES, devuelve True.
    """
    texto = texto.lower() # Convertimos el texto a minúsculas para normalizar la comparación
    return any(af in texto for af in AFIRMACIONES) # Comprobamos si alguna afirmación está en el texto
```

Función para extraer una fecha de un texto dado

```
def extraer_fecha(user_input):
    """
    Esta función intenta extraer una fecha del texto introducido por el usuario.
    Primero elimina palabras irrelevantes como 'el', 'la', 'del', etc.
    Luego usa la librería dateparser para intentar parsear la fecha.
    Si la fecha es válida, se devuelve en formato: 'Día de la semana DD de Mes'.
    Si no se puede extraer una fecha, retorna None.
    """
    # Eliminamos palabras innecesarias del texto usando expresiones regulares
    frase_limpia = re.sub(r"\b\be\b|\b\la\b|\b\lo\b|\b\del\b|\b\de\b", "", user_input.lower())
    # Usamos dateparser para intentar extraer la fecha del texto limpio
    fecha = dateparser.parse(frase_limpia.strip(), languages=["es"])
    if fecha:
        # Si se encontró una fecha, la formateamos y la devolvemos
        return fecha.strftime("%A %d de %B")
    return None # Si no se pudo extraer la fecha, devolvemos None
```

Explicación de las funciones

1. detectar_intencion:

- Esta función toma un texto, lo convierte a minúsculas para realizar una comparación estándar (sin importar mayúsculas/minúsculas), y luego verifica si alguna de las palabras en la lista SALUDOS está presente en el texto. Si detecta un saludo, devuelve "saludo", y si no, retorna "desconocida".

2. es_afirmacion:

- Aquí, la función también convierte el texto a minúsculas y busca si alguna palabra de la lista AFIRMACIONES está presente en el texto. Si encuentra alguna afirmación, devuelve True, indicando que el usuario ha afirmado algo. Si no, devuelve False.

3. extraer_fecha:

- Esta función limpia el texto de palabras irrelevantes que pueden interferir con la extracción de la fecha (por ejemplo, artículos como "el", "la", "de", etc.). Después usa la librería dateparser para intentar interpretar la fecha y la formatea en un estilo legible ("Día de la semana, DD de mes"). Si no puede encontrar una fecha válida, devuelve None.

chatbot_logic.py

Una vez tenemos nuestro archivo de nlp_utils.py vamos a generar/modificar el archivo chatbot_logic.py

```
# Importación de funciones necesarias
from nlp_utils import detectar_intencion, extraer_fecha, es_afirmacion
import random # Importamos la librería random para seleccionar respuestas aleatorias
# Definimos algunas respuestas iniciales para saludar al usuario
saludos_iniciales = [
    "Hola 🎉 ¿Quieres ver el menú o hacer una reserva?",
    "¡Bienvenido! 🎉 ¿Te muestro el menú o prefieres reservar?",
    "Hola, ¿en qué puedo ayudarte hoy? ¿Menú o reserva?",
    "¡Hola! 🎉 ¿Deseas consultar el menú o hacer una reserva?"
]
# Respuestas de despedida cuando se ha realizado una reserva
despedidas_reserva = [
    "¡Gracias por tu visita! 🎉 Que tengas un buen día.",
    "¡Reserva anotada! Disfruta tu jornada 🎉",
    "Perfecto, nos vemos el día acordado. ¡Cuidate!",
    "¡Todo listo! Te esperamos con gusto. 🎉"
]
# Respuestas de despedida cuando no se ha realizado una reserva
despedidas_sin_reserva = [
    "¡Gracias por tu visita! 🎉 Que tengas un buen día.",
    "¡Hasta la próxima! 🎉",
    "Encantado de ayudarte. ¡Nos vemos!",
    "Que tengas un buen día. ¡Adiós!"
]
# Función principal para gestionar las respuestas del chatbot
def get_response(user_input, session_id, memory):
```

```

user_input = user_input.lower().strip() # Normalizamos la entrada del usuario a minúsculas y eliminamos espacios innecesarios
# Inicializamos la memoria de la sesión si es la primera vez que interactúan
if session_id not in memory:
    memory[session_id] = {"estado": "inicio"} # Si no existe la sesión, comenzamos en el estado "inicio"
    estado = memory[session_id]["estado"] # Recuperamos el estado actual de la conversación
# Definimos las posibles afirmaciones y despedidas que el bot puede reconocer
afirmaciones = ["sí", "si", "claro", "por supuesto", "vale", "ok"]
despedidas = ["adiós", "adios", "hasta luego", "nos vemos", "chao", "chau", "bye", "no gracias", "no, gracias", "no"]
# Verificamos si el usuario ha solicitado despedirse
# Sólo permitimos despedidas si no estamos en medio de una reserva
if user_input.lower() in despedidas and estado not in [
    "esperando_dia", "esperando_personas", "esperando_nombre", "esperando_detalle"
]:
    memory[session_id]["estado"] = "despedida" # Cambiamos el estado a "despedida"
    reserva = memory[session_id].get("reserva", {}) # Recuperamos la reserva si existe
    # Si existe una reserva, mostramos un resumen de la misma
    if reserva:
        resumen = (
            f"Reserva para {reserva.get('personas')} personas el {reserva.get('dia')}, "
            f"a nombre de {reserva.get('nombre')}. Detalles: {reserva.get('detalles', 'ninguno')}."
        )
        # Seleccionamos una despedida con el resumen de la reserva
        mensaje = random.choice(despedidas_reserva).format(resumen=resumen)
        return mensaje
    else:
        # Si no hay reserva, mostramos una despedida genérica
        return random.choice(despedidas_sin_reserva)
# --- Manejo de afirmaciones "sí" ---
if user_input in afirmaciones:
    if estado == "ofrecido_menu":
        memory[session_id]["estado"] = "esperando_dia" # Si ya se ofreció el menú, preguntamos por la fecha
        return "Genial. ¿Para qué día quieres la reserva?"
    elif estado == "inicio":
        return "¿Puedes especificar si quieres ver el menú o hacer una reserva?" # Preguntamos si desea menú o reserva
    else:
        return "Vale, dime más detalles por favor."
# --- Flujo principal: Manejo de solicitudes del usuario ---
# Si el usuario menciona "menú", se le ofrece el menú
if "menu" in user_input:
    memory[session_id]["estado"] = "ofrecido_menu" # Actualizamos el estado a "ofrecido_menu"
    return "🍷🍷 Aquí tienes el menú: - Ensalada mixta - Pizza margarita - Pasta carbonara - Tarta de queso. ¿Quieres hacer una reserva?"
# Si el usuario menciona "reserva", comenzamos el flujo de reserva
if "reserva" in user_input:
    memory[session_id]["estado"] = "esperando_dia" # Actualizamos el estado a "esperando_dia"
    return "Perfecto. ¿Para qué día quieres hacer la reserva?"
# --- Flujos pendientes: fecha, personas, nombre, detalles, etc. ---
# Si estamos esperando una fecha, guardamos la fecha y preguntamos por el número de personas
if estado == "esperando_dia":
    memory[session_id]["estado"] = "esperando_personas"
    memory[session_id]["fecha"] = user_input
    return "¿Para cuántas personas será la reserva?"
# Si estamos esperando el número de personas, lo guardamos y preguntamos por el nombre
if estado == "esperando_personas":
    memory[session_id]["estado"] = "esperando_nombre"

```

```

memory[session_id]["personas"] = user_input
return "¿A nombre de quién estará la reserva?"
# Si estamos esperando el nombre, lo guardamos y preguntamos por detalles adicionales
if estado == "esperando_nombre":
    memory[session_id]["estado"] = "esperando_detalle"
    memory[session_id]["nombre"] = user_input
    return "¿Hay algo que debemos tener en cuenta? Alergias, niños, etc."
# Si estamos esperando detalles, completamos la reserva y mostramos un resumen
if estado == "esperando_detalle":
    memory[session_id]["estado"] = "reserva_completa"
    memory[session_id]["detalles"] = user_input
    memory[session_id]["reserva"] = {
        "dia": memory[session_id].get("fecha"),
        "personas": memory[session_id].get("personas"),
        "nombre": memory[session_id].get("nombre"),
        "detalles": memory[session_id].get("detalles")
    }
# Resumen de la reserva y una respuesta final
return (
    f"Reserva completa para {memory[session_id]['personas']} personas el {memory[session_id]['fecha']}, "
    f"a nombre de {memory[session_id]['nombre']}. Detalles: {memory[session_id]['detalles']}.\\n"
    "¿Deseas hacer otra cosa?"
)
# Si no se detecta ninguna de las condiciones anteriores, respondemos con un saludo inicial aleatorio
return random.choice(saludos_iniciales)

```

Explicación de la lógica general

Respuestas iniciales y despedidas

Se definen varias respuestas de saludo y despedida, tanto con reserva como sin reserva. El bot responderá de manera diferente dependiendo de si se ha realizado una reserva.

Flujo de conversación

El flujo de la conversación está determinado por el estado actual de la interacción, que se guarda en `memory[session_id]["estado"]`.

El estado inicial es "inicio", y luego el bot cambiará a otros estados como "ofrecido_menu", "esperando_dia", "esperando_personas", etc., dependiendo de las interacciones del usuario.

Cada vez que el usuario proporciona un dato relevante (por ejemplo, una fecha o el número de personas), el estado de la conversación se actualiza y el bot responde según corresponda.

Manejo de despedidas

Se verifica si el usuario ha usado una palabra de despedida (como "adiós" o "nos vemos") y se maneja la despedida en función de si se ha hecho una reserva o no. Si hay una reserva, se muestra un resumen.

Respuestas a afirmaciones

Si el usuario responde afirmativamente (con "sí", por ejemplo), el bot avanza al siguiente paso, ya sea pidiendo más detalles o esperando la siguiente entrada.

Interacción principal

Si el usuario menciona "menú", se le presenta el menú.

Si el usuario menciona "reserva", se le guía para hacer una reserva (fecha, número de personas, nombre, detalles adicionales).

Este flujo de trabajo permite una conversación interactiva y fluida en un chatbot que gestiona tanto solicitudes de menú como reservas, con respuestas personalizadas según el progreso de la interacción.

Y por último el archivo `views.py`

```
import json # Importa la biblioteca json para trabajar con datos en formato JSON.
from django.http import JsonResponse # Importa JsonResponse para devolver respuestas JSON a las solicitudes HTTP.
from django.shortcuts import render # Importa render para renderizar plantillas HTML en Django.
from django.views.decorators.csrf import csrf_exempt # Importa csrf_exempt para permitir solicitudes POST sin validación CSRF.
from .logic.chatbot_logic import get_response # Importa la función get_response que maneja la lógica del chatbot.

# Memoria por sesión simulada
# Aquí almacenamos el estado de cada sesión para simular la memoria del chatbot.
session_memory = {}

def chatbot_view(request):
    """
    Esta vista se encarga de renderizar la plantilla HTML del chatbot, lo que permite
    a los usuarios interactuar con el chatbot a través de una interfaz web.
    """
    return render(request, "chatbot/chatbot.html") # Renderiza la plantilla "chatbot.html" para el frontend.

@csrf_exempt # Decora la vista para deshabilitar la protección CSRF en esta función específica.
def chatbot_api(request):
    """
    Este endpoint maneja las solicitudes POST del usuario para interactuar con el chatbot.
    Recibe el mensaje del usuario y la session_id, y devuelve la respuesta del chatbot.
    """
    if request.method == "POST": # Verifica que la solicitud sea de tipo POST.
        try:
            data = json.loads(request.body) # Intenta cargar los datos JSON del cuerpo de la solicitud.
            user_message = data.get("message", "").strip() # Obtiene el mensaje del usuario.
            session_id = data.get("session_id", "default") # Obtiene el ID de la sesión o usa "default" si no se proporciona.

            if not user_message: # Si el mensaje está vacío.
                return JsonResponse({"response": "No recibí ningún mensaje."}, status=400) # Responde con un error 400 si no hay mensaje.

            bot_response = get_response(user_message, session_id, session_memory) # Llama a la función get_response para obtener la respuesta del bot.

            return JsonResponse({"response": bot_response}) # Devuelve la respuesta del bot en formato JSON.

        except json.JSONDecodeError: # Si ocurre un error al intentar decodificar los datos JSON.
            return JsonResponse({"response": "Error: JSON no válido."}, status=400) # Devuelve un error 400 si el JSON no es válido.

    return JsonResponse({"response": "Método no permitido"}, status=405) # Si el método HTTP no es POST, devuelve un error 405 (Método no permitido).
```

Explicación del código

Este código maneja las interacciones con el chatbot en Django bien estructurado.

Aquí una descripción breve de lo que hace cada parte del código:

1. **chatbot_view**: Esta vista se encarga de renderizar la plantilla HTML del chatbot, lo que permite que los usuarios interactúen con el chatbot desde una interfaz de usuario en el frontend.
2. **chatbot_api**: Este endpoint gestiona las solicitudes de interacción del usuario a través de una API. Recibe solicitudes POST con el mensaje del usuario y la `session_id`. Luego, llama a la función `get_response` para obtener una respuesta del bot. Si se produce un error en el formato de los datos o si el mensaje está vacío, se responde con un mensaje de error adecuado.

Algunas sugerencias que necesitarás mejorar para que sea realmente funcional

1. **Manejo de sesiones**: Aunque estás usando una memoria en memoria (el diccionario `session_memory`), en un entorno de producción real podrías necesitar usar algo más robusto, como la base de datos de Django o un sistema de almacenamiento como Redis, dependiendo de la cantidad de usuarios y sesiones concurrentes.
2. **Validación adicional de datos**: Aunque verificas si el mensaje está vacío, podrías agregar una validación adicional para asegurar que el `session_id` es válido, especialmente si este se usa para identificar a un usuario en particular.

Posibles errores y cómo solventarlos

En estos momentos todo funciona correctamente, la conversación es dirigida y fluida, con respuestas naturales, y hemos gestionado determinadas afirmaciones en el propio flujo conversacional tales como "sí", "no" etc, pero ahora nos quedan principalmente 2 problemas:

- No tenemos un buen control sobre el formato sobre las fechas, ya que "mañana", "el próximo jueves" o "el martes que viene" no los pasa a fecha numérica
- No contempla un nº mínimo o máximo de personas para la reserva, podríamos decir 0 o -5 y aceptaría la reserva sin problemas, este tipo de cosas e incongruencias debemos poder gestionarlas.

Así que, vamos allá.

Error de fechas

En este caso tenemos cierto control sobre estas pero queda bastante para poder decir que tenemos un control total de las fechas con el lenguaje natural del español, ya que utilizamos expresiones tales como, mañana, el próximo... o el ... que viene, seguro que se quedan algunas fuera pero al menos estas debemos tenerlas en cuenta para que la conversación con el chatbot sea fluida y no devuelva errores constatemente.

Para esto hemos modificado la función `extraer_fecha()` definida en el archivo `nlp_utils.py`, además he creado otra para preformatear la fecha antes de hacer el análisis y otra para asegurar que el resultado es en español después de obtener la fecha, son las siguientes:

Lo primero añadir a las importaciones:

```
from dateparser.search import search_dates
from datetime import datetime
```

Y al final del código, eliminamos el antiguo `extraer_fecha` e introducimos este código:


```
def limpiar_input_fecha(texto):
    texto = texto.lower().strip()
    # Reemplazos que requieren reordenar
    # Detecta "el jueves que viene" y cambia por "próximo jueves"
    match = re.match(r"(el\s+)?(?P<dia>\w+)\s+que\s+viene", texto)
    if match:
        dia = match.group("dia")
        return f"próximo {dia}"
    # "este jueves" → "jueves"
    texto = re.sub(r"\b(?:este|este|este)\b\s+(lunes|martes|miércoles|jueves|viernes|sábado|domingo)\b", r"", texto)
    # Casos simples: "el próximo lunes" → "próximo lunes"
    texto = re.sub(r"\bel\b", "", texto)
    texto = re.sub(r"\s+", " ", texto).strip()
    return texto

def extraer_fecha(user_input):
    limpio = limpiar_input_fecha(user_input)
    print(f"[DEBUG] Fecha interpretada: '{limpio}'")
    resultados = search_dates(
        limpio,
        languages=["es"],
        settings={
            "PREFER_DATES_FROM": "future",
            "RELATIVE_BASE": datetime.now()
        })
    if not resultados:
        return False, None
    # Tomamos la primera fecha que encuentre
    _, fecha = resultados[0] # el elemento "_" sirve para obviar el primer elemento de los resultados
    if fecha.month > 12 or fecha.day > 31:
        return False, None

    hoy = datetime.now()

    if fecha.year < hoy.year - 1 or fecha.year > hoy.year + 2:
        return False, None
    #return True, fecha.strftime("%A %d de %B") # Ejemplo: "viernes 19 de abril"
    return True, fecha # sin formatear
def formatear_fecha_es(fecha):
    dias = {
        "Monday": "lunes", "Tuesday": "martes", "Wednesday": "miércoles",
        "Thursday": "jueves", "Friday": "viernes", "Saturday": "sábado", "Sunday": "domingo"
    }
    meses = {
        "January": "enero", "February": "febrero", "March": "marzo",
        "April": "abril", "May": "mayo", "June": "junio", "July": "julio",
        "August": "agosto", "September": "septiembre", "October": "octubre",
        "November": "noviembre", "December": "diciembre"
    }
    dia_en = fecha.strftime("%A")
    mes_en = fecha.strftime("%B")
    dia = dias[dia_en]
    mes = meses[mes_en]
    return f"{dia} {fecha.day} de {mes}"
```

Modificación de la parte de tramitación de fechas del archivo chatbot_logic.py

Que no se te olvide importar las nuevas funciones, si son necesarias:

```
from .nlp_utils import detectar_intencion, extraer_fecha, es_afirmacion, formatear_fecha_es

# --- Flujos pendientes: fecha, personas, etc. ---
if estado == "esperando_dia":
    valido, fecha_parseada = extraer_fecha(user_input)
    if not valido:
        return "La fecha que escribiste no parece válida. Intenta con algo como 'mañana' o 'el próximo miércoles'."

    memory[session_id]["estado"] = "esperando_personas"
    fecha_formateada = formatear_fecha_es(fecha_parseada)
    memory[session_id]["fecha"] = fecha_formateada

    return f"Perfecto, para el {fecha_formateada}. ¿Cuántas personas serán?"
```

Error de comensales

El principal problema que teníamos aquí es que no acepte comensales fuera de un rango que empezará siempre en 1 y que termine en este caso en 25, siempre se podrá modificar para ampliar o reducir el número máximo de comensales en una reserva. Y por otro lado que acepte algunas expresiones coloquiales tales como, seremos 12, o vendremos 5... o simplemente dos.

Para conseguirlo, modificamos tanto el archivo `nlp_utils` como el archivo `chatbot_logic`:

`nlp_utils.py`

```
NUMEROS_PALABRAS = {
    "uno": 1, "una": 1, "dos": 2, "tres": 3, "cuatro": 4, "cinco": 5,
    "seis": 6, "siete": 7, "ocho": 8, "nueve": 9, "diez": 10,
    "once": 11, "doce": 12, "trece": 13, "catorce": 14, "quince": 15,
    "dieciséis": 16, "diecisiete": 17, "dieciocho": 18, "diecinueve": 19,
    "veinte": 20, "veintiuno": 21, "veintidós": 22, "veintitrés": 23,
    "veinticuatro": 24, "veinticinco": 25,
    "pareja": 2, "una pareja": 2
}

def extraer_personas(texto):
    texto = texto.lower()
    # Buscar número con dígitos
    match_num = re.search(r"\b(\d{1,3})\b", texto)
    if match_num:
        num = int(match_num.group(1))
        if 1 <= num <= 25:
            return True, num
        return False, None # fuera de rango
    # Buscar número en palabras
    for palabra, valor in NUMEROS_PALABRAS.items():
        if palabra in texto:
            if 1 <= valor <= 25:
                return True, valor
            return False, None # fuera de rango
    return False, None # no se encontró nada útil
```

Y el archivo chatbot_logic.py

Repasar las importaciones necesarias

```
from .nlp_utils import detectar_intencion, extraer_fecha, es_afirmacion, formatear_fecha_es, extraer_personas
```

y el código del archivo en su lugar correspondiente:

```
if estado == "esperando_personas":
    valido, personas = extraer_personas(user_input)
    if not valido:
        return "No entendí cuántas personas serán. Por favor, indica un número entre 1 y 25."
    memory[session_id]["estado"] = "esperando_nombre"
    memory[session_id]["personas"] = personas
    return "¿A nombre de quién estará la reserva?"
```

Con todas estas modificaciones hemos conseguido un chatbot de gestión de reservas completamente funcional, puedes añadir elementos y/o mejorar este proyecto o adaptarlo a cualquier circunstancia para poder gestionar otro tipo de reservas.

Resumen



- Debemos integrar spaCy en nuestro proyecto Django configurándolo adecuadamente para permitir el procesamiento avanzado del lenguaje natural.
- Es necesario modificar el archivo logic_chatbot.py para incorporar las funcionalidades de spaCy, mejorando así la interpretación de las interacciones del usuario.
- Actualizar nlp_utils.py permitirá que el chatbot reconozca saludos y otras intenciones de forma dinámica, adaptándose a las distintas maneras en que los usuarios pueden expresarse.
- El archivo chatbot_logic.py debe adaptarse para que pueda manejar estas detecciones dinámicas, proporcionando respuestas más precisas y naturales.
- Crear un flujo conversacional eficiente involucra la correcta configuración de nlp_utils.py y chatbot_logic.py, asegurando la calidad de la interacción.
- Por último, el archivo views.py debe integrarse adecuadamente para representar el interfaz visual que conecta las funcionalidades anteriores con el usuario final.
- Es común encontrar posibles errores durante el proceso, como mal configuraciones o dependencias faltantes, y es esencial saber cómo solventarlos para mantener el flujo del chatbot.
- Un ejemplo simple para ilustrar una mejora dinámica podría ser que antes el chatbot sólo respondía a "Hola", pero con estas mejoras puede reconocer variaciones como "Hey" o "Qué tal" y responder adecuadamente.

Actividades prácticas

Implementación de Gestión Conversacional

En esta actividad práctica, tu tarea es diseñar e implementar el flujo conversacional de un chatbot utilizando Django y Python, con habilidades de Procesamiento de Lenguaje Natural (PLN). Utilizando la estructura y ejemplos proporcionados en el contenido del curso, crea un chatbot para la gestión de reservas en un restaurante que maneje conversaciones fluidas y contextualizadas. Tu implementación debe incluir el manejo de intenciones, respuestas afirmativas, el reconocimiento de fechas y la cantidad de comensales utilizando archivos:

- nlp_utils.py
- chatbot_logic.py
- views.py

- 1. Implementación de reconocimiento de intenciones:** Modifica la función `detectar_intencion` en `nlp_utils.py` para incluir nuevas intenciones relevantes al contexto del restaurante, como consultas sobre horario de apertura o ubicación.

Para lograrlo, puedes ampliar la lista de intenciones, añadiendo nuevas categorías de palabras cargadas que correspondan a consultas sobre aperturas u ubicación. Modifica también la lógica para una gestión adecuada de las respuestas.

- 2. Adaptación del flujo conversacional:** En `chatbot_logic.py`, ajusta el flujo actual para incluir preguntas sobre preferencias de espacio en el restaurante (ej., patio, interior) al confirmar una reserva.

Añade un nuevo estado en el diccionario de memorias de sesiones que permite al flujo preguntar y guardar la respuesta acerca de las preferencias de espacio del cliente.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos