UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRADO EN INGENIERIA INFORMÁTICA

Trabajo de final de grado

Chatbot para la plataforma de participación ciudadana Decidim Barcelona

Autor: Víctor Llinares Muñoz

Directoras:	**Dra. Inmaculada Rodríguez Santiago**
	Dra. Maite López Sánchez
Realitzado en:	**Departament de Matemàtiques i Informàtica**

Barcelona, 30 de junio de 2023

Abstract

Artificial Intelligence is in a moment of rapid growth in the implementation of different technologies. Among these technologies, the recent boom in the use of conversational agents, such as ChatGPT, stands out. We can observe that nowadays new chatbots with new and better functionalities are appearing every day, and every day, these functionalities are becoming more similar to the capabilities of a human being. This final degree project is an initiation to the world of conversational agents that allows in the first instance to identify their potential to be able to exploit them with the help of a conversational framework that allows to create chatbots. Specifically, in this project we have used the conversational framework `RASA`.

On the other hand, we are living in an era where everyone is connected thanks to the Internet. As a consequence, virtual communities have come to light. These virtual communities are defined as a group of people with common goals who interact with each other in an online environment. Such virtual communities, like Artificial Intelligence, will be something we will see more commonly in the near future.

In this context, using as a basis a previous TFG that formalized virtual communities, as well as the figure of a chatbot in them, this project proposes the inclusion of a conversational agent in a virtual community specialized in participatory democracy, so that it helps citizens to know it and to participate democratically in the decisions that affect their community. Specifically, this work focuses on the virtual community of `Decidim.barcelona`. This is a community of citizen participation that aims to improve the city of Barcelona through participatory processes such as participatory budgets, citizen assemblies, debates or collection of signatures.

Resumen

La Inteligencia Artificial se encuentra en un momento de rápido crecimiento de la implantación de diferentes tecnologías. De entre estas tecnologías destaca el reciente auge de la utilización de los agentes conversacionales, como puede ser ChatGPT. Podemos observar que en la actualidad cada día aparecen nuevos chatbots con nuevas y mejores funcionalidades y cada día, dichas funcionalidades se asemejan más a las capacidades de un ser humano. Este trabajo de fin de grado constituye una iniciación al mundo de los agentes conversacionales que permita en primera instancia identificar sus potencialidades para poder a continuación explotarlas mediante la ayuda de un framework conversacional que permite crear chatbots. Concretamente en este proyecto se ha utilizado el framework conversacional `RASA`.

Por otro lado, estamos viviendo en una época donde todas las personas estamos conectadas gracias a internet. Como consecuencia, han salido a la luz las comunidades virtuales. Dichas comunidades virtuales se definen como un grupo de personas con objetivos comunes que interactúan entre ellas en un entorno en línea. Dichas comunidades virtuales, igual que la Inteligencia Artificial, será algo que veremos más comúnmente en un futuro próximo.

En este contexto, usando de base un TFG anterior que formalizaba las comunidades virtuales, así como la figura de un chatbot en las mismas, este proyecto propone la inclusión de un agente conversacional en una comunidad virtual especializada en la democracia participativa, de forma que ayude a la ciudadanía a conocerla y a participar democráticamente de las decisiones que afectan a su comunidad. Concretamente, este trabajo se centra en la comunidad virtual de `Decidim.barcelona`. Se trata de una comunidad de participación ciudadana que pretende mejorar la ciudad de Barcelona mediante procesos participativos tales como presupuestos participativos, asambleas ciudadanas, debates o recolección de firmas.

Resumen

La Intel·ligència Artificial es troba en un moment de ràpid creixement de la implantació de diferents tecnologies. D'entre aquestes tecnologies destaca el recent auge de la utilització dels agents conversacionals, com pot ser ChatGPT. Podem observar que en l'actualitat cada dia apareixen nous chatbots amb noves i millors funcionalitats i cada dia, aquestes funcionalitats s'assemblen més a les capacitats d'un ésser humà. Aquest treball de fi de grau constitueix una iniciació al món dels agents conversacionals que permeti en primera instància identificar les seves potencialitats per a poder a continuació explotar-les mitjançant l'ajuda d'un framework conversacional que permet crear chatbots. Concretament en aquest projecte s'ha utilitzat el framework conversacional `RASA`.

D'altra banda, estem vivint en una època on totes les persones estem connectades gràcies a internet. Com a conseqüència, han sortit a la llum les comunitats virtuals. Aquestes comunitats virtuals es defineixen com un grup de persones amb objectius comuns que interactuen entre elles en un entorn en línia. Aquestes comunitats virtuals, igual que la Intel·ligència Artificial, serà una cosa que veurem més comunament en un futur pròxim.

En aquest context, usant de base un TFG anterior que formalitzava les comunitats virtuals, així com la figura d'un chatbot en aquestes, aquest projecte proposa la inclusió d'un agent conversacional en una comunitat virtual especialitzada en la democràcia participativa, de manera que ajudi la ciutadania a conèixer-la i a participar democràticament de les decisions que afecten la seva comunitat. Concretament, aquest treball se centra en la comunitat virtual de `Decidim.barcelona`. Es tracta d'una comunitat de participació ciutadana que pretén millorar la ciutat de Barcelona mitjançant processos participatius com ara pressupostos participatius, assemblees ciutadanes, debats o recol·lecció de signatures.

2023 Informatics Engineering UB.

Agradecimientos

Quiero agradecer la paciencia y la ayuda que me han brindado durante todo el semestre, las doctoras Inmaculada Rodríguez Santiago y Maite López Sánchez, tutoras de este trabajo de final de grado. Haciendo mención especial a la doctora Inmaculada, la cual me ha acompañado durante todo el desarrollo del proyecto.

Por otro lado, quiero agradecer a mis familiares y amigos los cuales me han apoyado durante el trabajo de final de grado, sobretodo en los momentos finales del mismo.

Índice general

1. Introducción **2**
- 1.1. Comunidades Virtuales . . . 3
- 1.2. Chatbots en Comunidades Virtuales . . . 3
- 1.3. Objetivos . . . 4
- 1.4. Estructura de la memoria . . . 5

2. Conceptos Previos **6**
- 2.1. IA y IA Conversacional . . . 6
- 2.2. NLP vs NLU vs NLG . . . 7
- 2.3. Framework conversacional . . . 8
- 2.4. Decidim Barcelona . . . 9

3. Análisis **11**

4. Diseño de la Aplicación **16**
- 4.1. Arquitectura de la aplicación . . . 16
- 4.2. Diseño del chatbot . . . 17
  - 4.2.1. Descripción de sus componentes . . . 17
  - 4.2.2. Intents del chatbot de Decidim.bcn . . . 21
  - 4.2.3. Entities del chatbot de Decidim.bcn . . . 24
  - 4.2.4. Slots del chatbot de Decidim.bcn . . . 26
  - 4.2.5. Actions del chatbot de Decidim.bcn . . . 28

5. Implementación y Resultados **32**
- 5.1. Tecnologias Utilizadas . . . 32
  - 5.1.1. Frameworks Conversacionales . . . 32
  - 5.1.2. Servidores Web . . . 33
- 5.2. Arquitectura de Rasa . . . 34

5.3. Arquitectura del Sistema 34
5.3.1. Decidim Barcelona 36
5.3.2. Flask 36
5.3.3. Rasa Webchat 37
5.3.4. Rasa Action Server 40
5.3.5. Base de Datos 42
5.4. Resultados 47
5.4.1. Usuaria No Registrada 47
5.4.2. Usuaria Registrada 54
5.4.3. Usuaria Registrada y No Registrada 59
5.5. Chatbot Decidim vs ChatGPT 62

6. Conclusiones y Trabajo futuro 66

A. Manual Técnico 67

B. Apéndice con detalles de implementación 69

Índice de figuras

2.1. Relación entre NLP - NLU - NLG 7

4.1. Estructura general de un chatbot con integración en una página web 16

4.2. Extracto del fichero nlu.yml donde vemos la estructura del mismo . 18

4.3. Extracto del fichero archivo `stories.yml` donde vemos la sintaxis de una story 19

4.4. Extracto del fichero config.yml donde se puede ver una parte de mi configuración del pipeline 20

4.5. Diagrama de funcionamiento de las policies 20

4.6. Extracto del archivo domain.yml, sección de intents 21

4.7. Extracto del archivo nlu.yml, intent saludar 22

4.8. Extracto del archivo nlu.yml, intents de definición 23

4.9. Extracto del archivo nlu.yml, intent dar_barrio 23

4.10. Extracto del archivo nlu.yml, intent devolver_proceso_participativo . 24

4.11. Extracto del fichero nlu.yml donde se pueden apreciar la definición de la entity user_name. 25

4.12. Extracto del fichero nlu.yml donde podemos ver la estructura de una lookup table 26

4.13. Ejemplo de estructura de slots vinculados a entities del chatbot de Decidim.barcelona (archivo domain.yml) 27

4.14. Ejemplo de estructura de slots de tipo *categorical* del chatbot de Decidim.barcelona (archivo domain.yml) 27

4.15. Ejemplo de estructura de una rule del chatbot de Decidim.barcelona (archivo domain.yml) 28

4.16. Ejemplo de response del chatbot de Decidim.barcelona (archivo domain.yml) 29

5.1. Comparativa de las características entre frameworks conversacionales 33

5.2. Arquitectura de un chatbot de Rasa 35

5.3. Diagrama del sistema . . . 35
5.4. Estructura de archivos en Flask . . . 37
5.5. Extracto de código del widget (fichero completo en la Figura B.3 del Anexo) . . . 38
5.6. Extracto de código donde se observa "onSocketEvent" . . . 39
5.7. Extracto de código donde se muestra la implementación de una custom action . . . 41
5.8. Ejemplo de query en GraphQL . . . 42
5.9. Ejemplo de resultado de la query 5.8 en GraphQL . . . 43
5.10. Ejemplo de query en python a la API de Decidim . . . 44
5.11. Ejemplo para obtener datos de una colección en MongoDB . . . 46
5.12. La chatbot inicia la conversación, se presenta y le pregunta su nombre 48
5.13. La chatbot le da opciones a la usuaria para que clique . . . 48
5.14. La chatbot le explica quien puede participar . . . 49
5.15. La chatbot le propone buscarle un proceso participativo . . . 49
5.16. La chatbot le da a escoger algún barrio donde buscar . . . 50
5.17. La chatbot le ofrece un proceso . . . 50
5.18. La chatbot le da otras opciones a la usuaria para que clique . . . 51
5.19. La chatbot le ofrece algunos conceptos que puede que la usuaria no entienda . . . 51
5.20. La chatbot le explica cómo registrarse . . . 52
5.21. La chatbot le muestra una descripción de la página de procesos . . . 52
5.22. La chatbot le muestra una descripción de la página de un proceso . . . 53
5.23. La chatbot le muestra una descripción de la página de encuentros de un proceso . . . 53
5.24. La chatbot le muestra una descripción de la página de propuestas de un proceso . . . 54
5.25. Inicio de conversación con usuaria registrada . . . 55
5.26. La chatbot le propone ofrecerle un proceso según sus intereses . . . 55
5.27. La chatbot le ofrece un proceso según sus intereses . . . 56
5.28. La chatbot le propone si le interesa un resumen de algún encuentro 56
5.29. La usuaria pide un resumen de encuentros y la chatbot le ofrece los disponibles . . . 57
5.30. La chatbot le ofrece un resumen del encuentro seleccionado . . . 57
5.31. Resumen de los comentarios del encuentro . . . 58
5.32. La chatbot ofrece 2 funcionalidades en la página de propuestas . . . 58

5.33. La chatbot le ofrece propuestas a partir de sus amistades 59
5.34. La chatbot le ofrece las últimas 3 propuestas 59
5.35. La chatbot le proporciona una explicación según la página donde esté (en este caso la página principal de Decidim) 60
5.36. La chatbot se presenta 60
5.37. La chatbot le proporciona el título y *link* de un proceso 61
5.38. La chatbot le proporciona la definición del concepto 61
5.39. La chatbot muestra si el proceso tiene una componente o no 62
5.40. Pregunta al chatbot: “Qué es Decidim Barcelona” 63
5.41. Pregunta al chatbot: “Soy de Horta” 64
5.42. Pregunta al chatbot: “Me puedes ofrecer un proceso participativo de mi barrio?” 65

B.1. Extracto de código de app.py 69
B.2. Código base del widget de Rasa 70
B.3. Screenshot del widget de la página principal 71
B.4. Función que envía un intent de forma externa a Rasa 71
B.5. Ejemplo para obtener la instancia de una bdd en MongoDB 72
B.6. Ejemplo para insertar datos en una colección en MongoDB 72

Capítulo 1

Introducción

En los últimos años se puede observar un crecimiento significativo del mundo digital. En particular, las plataformas digitales han evolucionado de manera impresionante y han cambiado la forma en que interactuamos, brindando una multitud de servicios. Podríamos destacar las redes sociales como Instagram o Tik Tok, así como las plataformas digitales de participación ciudadana que están empezando a surgir ahora.

Dichas plataformas son espacios digitales de reunión y consenso para conectar gobernantes y ciudadanía y generar una cultura participativa que permita encarar de forma colaborativa las necesidades de cada comunidad (ya sea el municipio, la región, el país, o cualquier otro tipo de comunidad o asociación). En la actualidad estas plataformas participativas están creciendo debido al hecho de que más y más gente ve necesario una mayor implicación ciudadana para solucionar problemas de las ciudades (o colectivos) y mejorar la convivencia, de una forma más participativa y democrática [1].

Pero estas plataformas aún no son suficientemente conocidas, y aún menos utilizadas. Para mejorar la participación de las personas usuarias en estas plataformas, este proyecto propone la incorporación de un chatbot a dichas plataformas. Un chatbot, también denominado agente conversacional, es una inteligencia artificial capaz de mantener una conversación con una persona [2]. La incorporación de esta IA a una plataforma digital de participación ciudadana podría mejorar la experiencia de las usuarias y las usuarias y mejorar su participación en ella. Esta tecnología está enfocada tanto para personas usuarias nuevas (a las cuales se les podría orientar para conocer mejor la plataforma e incentivarles a participar), como para usuarias ya conocedoras de la plataforma (ayudándoles mediante una orientación personalizada a sus necesidades).

Por tanto, este proyecto pretende enlazar dos servicios digitales que están en auge, por un lado las plataformas digitales de participación ciudadana y por otro lado los chatbots. Para ello, nos hemos decantado por `Decidim.barcelona`, una plataforma de participación ciudadana impulsada por el Ayuntamiento de Barcelona. Debido al tiempo de desarrollo de un Trabajo de Final de Grado solamente nos hemos centrado en el apartado de procesos participativos. Por otro lado, he-

mos escogido Rasa como framework conversacional mediante el cual se diseña e implementa el chatbot comunitario.

1.1. Comunidades Virtuales

Definiéndola de forma simple, una comunidad virtual es un grupo de personas con objetivos comunes que interactúan entre ellas en un entorno en línea. En las plataformas de participación ciudadana existen multitud de comunidades virtuales de personas que se unen para apoyar un proceso, debatir, etc..

Existen definiciones más conceptualizadas, como la que se da en [3]: "*una comunidad virtual es la combinación de un sistema técnico, el objetivo del cual es maximizar la realización de actividades, y de un sistema social, en que el objetivo es maximizar la calidad de vida de las personas usuarias del sistema*".

En cualquier caso, las comunidades virtuales se estructuran alrededor de la interacción entre los miembros. Esta interacción es causada por la participación de las personas usuarias en las diversas actividades que se ofrecen: normalmente pueden ser *actividades informativas*, que se basan en la recopilación, distribución y utilización de información en cualquier formato; *actividades discursivas*, como puede ser compartir la opinión o entablar conversaciones con otras personas participantes; o *actividades participativas*, que pueden ser tanto *individuales*, por ejemplo formar parte de una votación o responder cuestionarios, como *colaborativas*, como puede ser una videoconferencia o involucrarse en proyectos de equipo. La diversidad de actividades favorecerá la participación y la interacción de las personas usuarias y, así también, la socialización y el aprendizaje interpersonal [4].

Teniendo en cuenta lo que se ha comentado en esta sección, destacando que las comunidades virtuales se estructuran alrededor de la interacción entre los miembros, resultaría de gran utilidad, para cualquier comunidad virtual, la implementación de mecanismos que facilitaran y animaran la participación de los miembros y en particular para la plataforma Decidim.barcelona, la cual se encuentra aún en proceso de crecimiento.

1.2. Chatbots en Comunidades Virtuales

El estudio sobre los agentes conversacionales o chatbots [5] muestra su tendencia creciente como tema de investigación dada su amplia aplicación. Aún así cabe destacar la diversidad de casos de uso entre diferentes chatbots, es decir, hay que analizar cuáles son las necesidades de la comunidad virtual para enfocar una especialización del asistente virtual.

Se deberían entender también los diferentes roles que un chatbot puede tener. Principalmente nos centraremos en aquellos chatbots que imparten una función de informador, y aquellos chatbots que imparten una función de dinamizador (el cual fomenta la participación de las personas usuarias en la comunidad virtual). Ambos

son importantes, y uno se complementa con el otro. Podríamos entender al chatbot informador como una versión más simple que permite a la usuaria entender cómo funciona la plataforma digital, y cómo participar de forma efectiva (*actividades informativas*), además de guardar los gustos o características de la usuaria. Por otra parte, el chatbot dinamizador a partir de la información que obtiene el primero o, a partir de una base de datos relacionada con la plataforma, puede ofrecer de forma personalizada diferentes actividades a cada participante. De esta forma, el chatbot puede tanto mantener activa la participación de los miembros en los procesos o actividades de la comunidad (*actividades participativas*) como animar también la participación entre ellos mediante debates (*actividades discursivas*).

Existen retos técnicos para que los agentes conversacionales se comporten como se espera: entender las intenciones, las expectativas y los contextos de uso de las personas usuarias probablemente impulsará que sea una herramienta más eficaz [6]. Por tanto, a la hora de implementar un chatbot se tienen que tener en cuenta el acompañamiento orientado, sobretodo para nuevos participantes que no entienden la web. Además de la utilización de conversaciones anteriores con éstos y de datos específicos que marcan la diferencia entre lo que le interesa a una usuaria o a otro.

1.3. Objetivos

El objetivo general de este TFG consiste en el diseño e implementación de un prototipo de chatbot dentro de la comunidad virtual de `Decidim.barcelona`. Dicha inteligencia artificial pretende mejorar la experiencia de las personas usuarias de la web de Decidim. El agente conversacional se dirige tanto a nueva s usuarias, que no sepan utilizar las funciones de la web, o que su intención sea empezar a participar en diferentes procesos participativos, como a usuarias recurrentes a las cuales se les animará a seguir participando en procesos de la plataforma según sus gustos y preferencias. De todo el dominio de Decidim.barcelona (procesos participativos, iniciativas ciudadanas y órganos de participación), nos centraremos, de forma específica en los procesos participativos.

Este objetivo general se desglosa en los siguientes objetivos específicos:

1. Estudiar diferentes plataformas conversacionales para seleccionar una de ellas.
2. Aprender sobre la plataforma conversacional elegida.
3. Aprender sobre la plataforma de participación ciudadana `Decidim.barcelona`.
4. Implementar un servidor que emule la web de Decidim.
5. Desarrollar y entrenar un agente conversacional que ayude a las personas usuarias durante la navegación por Decidim.barcelona.
6. Integrar el agente en dicha plataforma emulada.

1.4. Estructura de la memoria

La memoria se estructura de forma que en el `Capítulo` 2 se expondrán una serie de conceptos previos necesarios para poder entender mejor el resto de capítulos de este proyecto. A continuación, en el `Capítulo` 3, se listarán las funcionalidades y conversaciones posibles del chatbot comunitario desarrollado. Posteriormente en el `Capítulo` 4, se hablará sobre el diseño global de la aplicación y de los diferentes componentes de un agente conversacional en la plataforma Decidim.barcelona. Seguidamente, en el `Capítulo` 5 se entrará en detalle en la implementación de dicha inteligencia artificial, explicando además, las tecnologías usadas y donde también mostraremos los resultados. Finalmente, cerraremos este proyecto con la conclusión y los futuros avances.

Capítulo 2

Conceptos Previos

2.1. IA y IA Conversacional

La Inteligencia Artificial (IA) es la columna vertebral de la innovación en la computación moderna [7]. Ésta representa un conjunto de tecnologías que permiten a los ordenadores realizar una variedad de funciones avanzadas, con el propósito de emular la inteligencia humana. La IA incluye áreas tales como el aprendizaje automático, la visión artificial, reconocimiento y generación de voz, los sistemas multiagente, etc.

Situándonos en el contexto particular de la *IA conversacional* (también denominada como chatbot o agente conversacional), ésta se considera una rama de la Inteligencia Artificial porque trata de imitar habilidades humanas como son captar, comprender, y generar lenguaje, tanto escrito como hablado. Éste tipo de IA pertenece a la Narrow AI, que se refiere a las inteligencias artificiales que eligen e intentan reproducir sólo un conjunto limitado de habilidades humanas, con el fin de manejar un conjunto limitado de tareas [8].

En general, consideramos conversacional todo sistema que entiende lo que las usuarias escriben o dicen en su propia lengua y que responde utilizando la misma lengua.

Estos sistemas utilizan grandes volúmenes de datos, aprendizaje automático y procesamiento del lenguaje natural para imitar las interacciones humanas, reconocer la voz y el texto.

2.2. NLP vs NLU vs NLG

Esta sección introduce el área del NLP (Natural Languaje Processing) y presenta los diferentes componentes que un agente conversacional implementa para entender el lenguaje humano correctamente y expresarse de igual manera con la persona usuaria. La Figura 2.1 muestra la relación entre los diferentes conceptos explicados en este apartado, NLP vs NLU vs NLG.

El NLP (Natural Language Processing) se encarga de dotar a la máquina de la capacidad de captar lo que dice una usuaria, descomponerlo y comprender cuál es su significado [9]. El objetivo es determinar una acción adecuada para cada *input* (mensaje) que da la usuaria. Un perfecto ejemplo son los asistentes de voz de los smartphones: una persona solicita a la máquina un contenido (señal de entrada), la máquina interpreta dicho contenido y ejecuta una acción. La intención del NLP es diseñar mecanismos de comunicación usuario-máquina que sean eficaces computacionalmente.

Dentro del NLP se pueden distinguir dos grandes componentes, el NLU, el componente encargado de entender los mensajes de la usuaria y el NLG, el componente que se encarga de generar texto que la usuaria pueda entender.

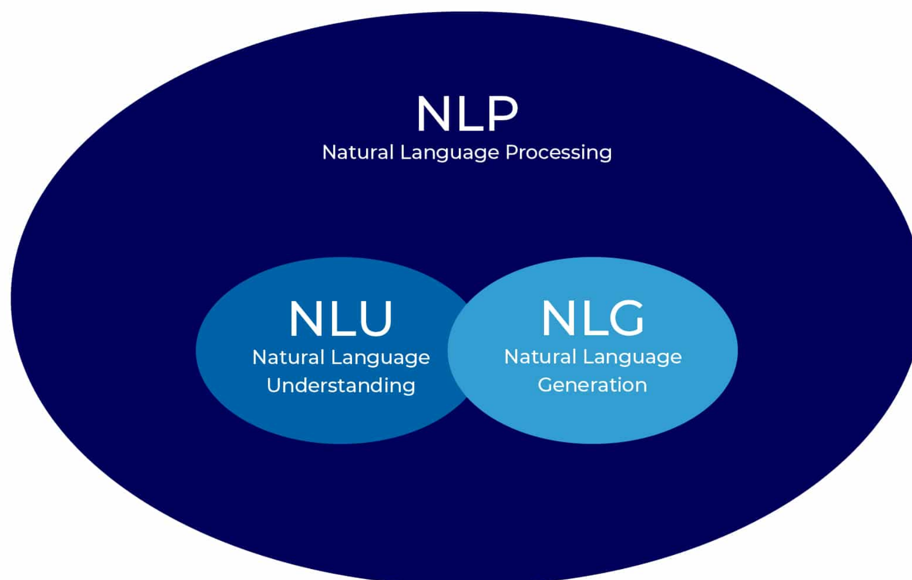

Figura 2.1: Relación entre NLP - NLU - NLG

NLU (Natural Language Understanding) es el componente que permite a un chatbot entender *inputs* de la usuaria expresados en lenguaje natural, es decir en la forma natural de expresarse de una persona. Es un subconjunto de NLP que trata con un área mucho más específica. Se enfoca en cómo manejar mejor las entradas no estructuradas para convertir éstas en un formato estructurado. Así la máquina puede entender, comprender y producir una acción [9].

Por el momento, no todos los chatbots utilizan NLU: algunos sólo se centran

en flujos conversacionales en los que las usuarias sólo pueden seleccionar una única opción, pero no pueden realmente conversar libremente. Sin embargo, los agentes conversacionales que utilizan NLU pueden proporcionar sin duda alguna una experiencia de usuario mucho más natural. Sin embargo, gestionar el componente NLU no es tarea fácil. Ya que es necesario seleccionar un buen conjunto de frases de entrenamiento para el algoritmo de aprendizaje automático y seguir ajustando ese conjunto, con el fin de mejorar constantemente el rendimiento del agente conversacional.

Por último, NLG (Natural Language Generation) se refiere a los procesos de inteligencia artificial que producen una respuesta a partir de datos estructurados en lenguaje natural, como texto o voz [9]. Un ejemplo son los chatbot o los asistentes de voz de Google y Apple. Estos son capaces de generar lenguaje ante una petición como preguntarles por el tiempo que va a hacer hoy. La forma más básica de realizar la generación de respuestas en lenguaje natural es mediante una serie de respuestas predefinidas que el algoritmo escoge en función de como avanza la conversación. La forma más sofisticada es la que usa algoritmos de aprendizaje que generan respuestas de forma dinámica, como por ejemplo ChatGPT.

2.3. Framework conversacional

Rasa es un framework conversacional, un framework[1] destinado al desarrollo de chatbots impulsados por inteligencia artificial. Es increíblemente potente y lo utilizan desarrolladores de todo el mundo para crear chatbots y asistentes contextuales (p. ej. Adobe, Dell, Accenture, Orange, etc..).

Aunque para este proyecto nos centraremos en `Rasa Open Source`, la rama gratuita de RASA y menos potente, ya que el resto de productos de Rasa, como Rasa X o Rasa Pro, están pensados para empresas. Pese a ser, una versión menos potente, es la herramienta perfecta para una primera toma de contacto y crear Inteligencias Artificiales conversacionales.

Rasa Open Source es un framework escrito en `Python` y de código abierto. Basa su funcionamiento en dos componentes principales, NLU y Core:

1. `NLU` es la parte encargada de tomar el texto de la usuaria, analizarlo y descomponerlo para crear una estructura organizada de tal manera que el chatbot pueda comprender el contenido del mensaje.

2. `Core` es la parte encargada de decidir la acción que tiene que tomar el agente una vez ha entendido el *input* de la usuaria, ya sea mostrar por pantalla una respuesta predefinida o responder basándose en una acción más compleja como consultar una base de datos o realizar una llamada a una API.

[1]Un framework es un esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto con objetivos específicos, una especie de plantilla que sirve como punto de partida para la organización y desarrollo de software.

Para que un chatbot pueda entender los mensajes de la usuaria debemos tener una definición de los distintos tipos de mensajes que puede recibir el chatbot. A esto se le denominan `intents` o "intenciones", ya que muestran la intención de dicho mensaje.

Por otra parte, dentro de cada mensaje podemos tener ciertos datos o palabras clave que nos interesa extraer, como podría ser el nombre de la usuaria, un email, etc... A esto se le denominan `entities` o "entidades" y mediante un buen conjunto de ejemplos, o dicho de otra forma, un buen conjunto de datos de entrenamiento, el chatbot será capaz de extraerlas de forma correcta, así como clasificar los intents de igual forma.

Además, es importante mencionar que a veces nos interesa no solo extraer un extracto del mensaje de la usuaria, sino que nos interesa guardarlo para usarlo posteriormente en la conversación. Para ello tenemos los `slots`. Nos servirán para guardar información, extraída de la conversación (o no).

Por último, después de cada mensaje de la usuaria, el servidor de Rasa decidirá la siguiente acción a ejecutar. Hay diferentes tipos de acciones, desde la acción más básica (default action) como puede ser *action_listen*, con la cual el chatbot se mantendrá a la espera de un mensaje de la usuaria, hasta las acciones más complejas (custom actions) las cuales son implementadas por el programador.

Un chatbot de Rasa se puede usar de manera independiente mediante un comando de consola (*rasa shell*), de esta manera se cargará el último modelo de entrenamiento guardado y se podrá interactuar con el chatbot libremente. No obstante, este comando no saca a relucir todo el potencial de Rasa, por ello se recomienda la utilización de cualquier tipo de frontend para poder explotar al máximo las funcionalidades que ofrece de cara a la usuaria. Se verá más adelante que para este proyecto se ha decidido conectar el chatbot a un entorno web y se ha escogido un widget, el cual proporciona una interfaz gráfica al chatbot y que se puede acoplar de manera sencilla a cualquier archivo HTML.

2.4. Decidim Barcelona

Decidim Barcelona es una plataforma de participación ciudadana impulsada por el Ayuntamiento de Barcelona. En la propia página de Decidim.barcelona se define como "la plataforma digital de participación del Ayuntamiento de Barcelona para construir una ciudad más democrática. Un espacio de referencia para construir una ciudad abierta, transparente, colaborativa y con el protagonismo de quien la habita." [10].

Decidim está formado principalmente por 3 componentes, los procesos participativos, los órganos de participación y las iniciativas ciudadanas.

En primer lugar están los procesos participativos, los cuales destacan en Decidim.barcelona sobre los demás componentes debido a que hasta el momento es el servicio más utilizado por las personas usuarias. Motivo por el cual hemos orientado este proyecto en esa dirección, además que no se podría abarcar todo lo que

ofrece Decidim barcelona en el periodo de tiempo de un TFG. En segundo lugar, los órganos de participación son grupos de personas que se reúnen con el fin de debatir y recoger propuestas en torno a los procesos participativos de un barrio concreto. Por último, las iniciativas ciudadanas son el medio que tiene la ciudadanía para promover a través de la recogida de firmas, que el Ayuntamiento lleve a cabo una determinada acción.

Más concretamente, un `proceso participativo` se define como "una serie de encuentros delimitados en un tiempo concreto para promover el debate y el contraste de argumentos entre la ciudadanía, o entre ésta y las personas responsables municipales"[11], desde el cual, principalmente, surgen debates, encuentros y propuestas. Un proceso participativo puede estar formado por los anteriores componentes, pero también puede tener otros componentes como encuestas, votaciones, presupuestos, etc. De entre todos ellos destacamos los introducidos a continuación:

Un `debate` es un espacio digital para que las usuarias se informen y decidan sobre las propuestas de cada proceso participativo. Los debates se hacen de forma online mediante comentarios en el apartado de cada debate.

Los `encuentros` son reuniones presenciales de debate, relacionadas con los procesos participativos y de donde surgen propuestas.

Las `propuestas` son ideas recogidas desde los encuentros o creadas individualmente por las personas usuarias, quienes pueden interactuar con las propuestas siguiéndolas, para recibir notificaciones de cada avance, o apoyándolas votando positivamente.

Capítulo 3

Análisis

Esta sección define las funcionalidades que proporcionará la chatbot de Decidim.barcelona a los participantes en la plataforma. Dichas funcionalidades (agrupadas según perfil de persona usuaria) se especifican en un formato `usuaria pregunta - chatbot responde`. Cabe destacar que distinguimos dos tipos de usuarias, las usuarias registradas y las usuarias no registradas. Las usuarias registradas son aquellas usuarias habituales que ya conocen Decidim, han entrado con anterioridad, han visto las diferentes funcionalidades que ofrece la plataforma y no necesitan una visita guiada de ella sino que, de forma personalizada, se les ofrezca nuevos procesos o un seguimiento de los que pueden parecerles interesantes.

Por otro lado, las usuarias no registradas son usuarias que asumimos que no han entrado nunca a Decidim, han oído hablar de ella y se han interesado, o no se han terminado de familiarizar con la plataforma. Estas usuarias son muy importantes para la plataforma de Decidim ya que está en crecimiento y se necesita familiarizar a la ciudadanía del uso de la misma. A estas usuarias se les proporcionará una ayuda más guiada para que puedan familiarizarse con todo lo que la plataforma ofrece, y así fomentar su participación.

A continuación se listan agrupadas en categorías las diferentes interacciones (o conversaciones) que la usuaria y la chatbot Sara pueden tener. Cabe destacar, que las siguientes interacciones contienen ejemplos de posibles mensajes de la usuaria, no obstante, la usuaria puede expresarse de otra forma similar.

1. Interacciones Sociales

    1) Interacción 1-Saludo

        - La usuaria saluda a la chatbot.
        - La chatbot Sara saluda de vuelta.

    2) Interacción 2-Agradecer

        - La usuaria le da las gracias a la chatbot.
        - La chatbot Sara responde "de nada" o similares.

    3) Interacción 3-Despedirse

- La usuaria se despide de la chatbot.
- La chatbot Sara se despide también.

4) Interacción 4-Contexto
   - La usuaria pregunta a la chatbot “¿Dónde estoy?”.
   - La chatbot Sara le muestra que puede hacer según la página donde esté la usuaria.
5) Interacción 5-Presentación
   - La usuaria pregunta a la chatbot “¿Qué eres?”.
   - La chatbot Sara se presenta.
6) Interacción 5-Funcionalidades
   - La usuaria pregunta a la chatbot “¿Qué puedes hacer?”.
   - La chatbot Sara le muestra sus funcionalidades.

2. Conversación 1-SaludoDefDec: Saludo y definición de Decidim (Usuaria no registrada)
   - La usuaria hace clic en el widget de la chatbot Sara y lo abre.
   - La chatbot Sara saluda a la usuaria y le pregunta su nombre.
   - La usuaria le indica su nombre.
   - La chatbot Sara muestra 4 botones para que la usuaria vea lo que puede hacer.
   - La usuaria selecciona el botón “Decidim”.
   - La chatbot Sara le responde con la definición.
3. Conversación 2-SaludoProcP: Saludo y proceso participativo (Usuaria no registrada)
   - La usuaria hace clic en el widget de la chatbot Sara y lo abre.
   - La chatbot Sara saluda a la usuaria y le pregunta su nombre.
   - La usuaria le indica su nombre.
   - La chatbot Sara muestra 4 botones para que la usuaria vea lo que puede hacer.
   - La usuaria selecciona el botón “¿Quién puede participar en los procesos participativos?” o “¿Cómo se participa?”.
   - La chatbot Sara le responde con una explicación acorde y le pregunta si quiere un proceso de ejemplo.
   - La usuaria dice que sí.
   - La chatbot Sara le da 5 botones con nombres de barrios y le dice que elija uno para filtrar la búsqueda.
   - La usuaria hace clic en uno de ellos.

  - La chatbot Sara le proporciona el título y la url de un proceso participativo.

4. Conversación 3-SaludoConcepto: Saludo y conceptos de Decidim (Usuaria no registrada)

   - La usuaria hace clic en el widget de la chatbot Sara y lo abre.
   - La chatbot Sara saluda a la usuaria y le pregunta su nombre.
   - La usuaria le indica su nombre.
   - La chatbot Sara muestra 4 botones para que la usuaria vea lo que puede hacer.
   - La usuaria selecciona "Otros".
   - La chatbot Sara le muestra 2 botones más con otras funcionalidades.
   - La usuaria selecciona el botón "Conceptos de Decidim".
   - La chatbot Sara le muestra conceptos que la usuaria puede no saber sobre Decidim.bcn.
   - La usuaria selecciona alguno de ellos.
   - La chatbot Sara le responde con la definición y le muestra botones de conceptos relacionados.

5. Conversación 4-SaludoRegistro: Saludo y cómo registrarse (Usuaria no registrada)

   - La usuaria hace clic en el widget de la chatbot Sara y lo abre.
   - La chatbot Sara saluda a la usuaria y le pregunta su nombre.
   - La usuaria le indica su nombre.
   - La chatbot Sara muestra 4 botones para que la usuaria vea lo que puede hacer.
   - La usuaria selecciona el botón "Otros".
   - La chatbot Sara le muestra 2 botones más con otras funcionalidades.
   - La usuaria selecciona el botón "¿Cómo me registro?".
   - La chatbot Sara le muestra una explicación de cómo hacerlo.

6. Conversación 5-VisitarPágina: Explicación de la página (Usuaria no registrada)

   - La usuaria visita una página por primera vez.
   - La chatbot Sara le muestra una explicación sobre la página en la que se encuentra.

7. Conversación 6-PedirProceso: Pedir un proceso participativo (Usuaria no registrada y usuaria registrada)

- La usuaria pide información sobre un proceso participativo.
- La chatbot Sara le devuelve el título y la url de el proceso más reciente.

8. Conversación 7-PedirProcBarrio: Pedir un proceso participativo de un barrio (Usuaria no registrada y usuaria registrada)
    - La usuaria pide información sobre un proceso participativo de un barrio en concreto.
    - La chatbot Sara le devuelve el título y la url de un proceso intentando buscar en esa zona (si no hay en esa zona le devuelve el último proceso participativo añadido a Decidim).
9. Conversación 8-Def: Definición (Usuaria no registrada y usuaria registrada)
    - La usuaria pregunta una definición (p. ej. "¿Qué es un proceso participativo?").
    - La chatbot Sara le devuelve la definición del concepto y muestra botones con conceptos relacionados a dicha definición.
    - Si a la usuaria le interesa alguno de los conceptos hace clic él.
    - La chatbot Sara le devuelve la definición del concepto.
10. Conversación 9-ComponenteProcP: Preguntar sobre el proceso participativo anterior (Usuaria no registrada y usuaria registrada)
    - La usuaria pregunta si el último proceso participativo tenía debates, propuestas, encuentros o presupuestos.
    - La chatbot Sara le confirma si el proceso dispone de ese componente.
        - Si el proceso tiene dicho componente, la chatbot le proporciona el *link* para llegar a él.
        - Si el proceso no tiene dicho componente, la chatbot le informa de ello.
11. Conversación 10-SaludoNombre: Saludo (Usuaria registrada)
    - La usuaria hace clic en el widget de la chatbot Sara y lo abre.
    - La chatbot Sara le saluda con su nombre.
12. Conversación 11-ProcInterés: Ofrecer proceso participatorio según los gustos de la usuaria (Usuaria registrada)
    - La usuaria navega, por primera vez en la conversación, hasta la página donde están todos los procesos participativos.
    - La chatbot Sara le ofrece buscar un proceso según sus gustos (obtenidos desde la base de datos).
    - La usuaria dice que sí.

- La chatbot Sara le devuelve el título y *link* de un proceso acorde a sus gustos.
  - Si no existe ninguno acorde a sus gustos se le notifica.

13. Conversación 12-ResumenEnc: Página de encuentros (Usuaria registrada)

    - La usuaria navega, por primera vez en la conversación, hasta la página de encuentros de un proceso participativo concreto.
    - La chatbot Sara le ofrece los encuentros con comentarios, que obtiene de la base de datos.
    - La usuaria selecciona uno.
    - La chatbot Sara le devuelve el resumen de dicho encuentro.

14. Conversación 13-OrdenarProp: Ordenar propuestas (Usuaria registrada)

    - La usuaria navega, por primera vez en la conversación, hasta la página de propuestas de un proceso participativo concreto.
    - La chatbot Sara le ofrece elegir entre las últimas propuestas de dicho proceso participativo y las propuestas que siguen sus amistades.
    - La usuaria selecciona el botón de "Últimas propuestas".
    - La chatbot Sara le devuelve el título y *link* de las últimas 3 propuestas del proceso donde se encuentre.

15. Conversación 14-PropAmistades: Ordenar propuestas (Usuaria registrada)

    - La usuaria navega, por primera vez en la conversación, hasta la página de propuestas de un proceso participativo concreto.
    - La chatbot Sara le ofrece elegir entre las últimas propuestas de dicho proceso participativo y las propuestas que siguen sus amistades.
    - La usuaria selecciona el botón de "Propuestas de amistades".
    - La chatbot Sara le devuelve propuestas que sus amistades siguen.

Capítulo 4

Diseño de la Aplicación

4.1. Arquitectura de la aplicación

La arquitectura de este proyecto se basa en un modelo `cliente-servidor`. Es muy usada sobre todo en el diseño de webs y servicios online, y se basa en la existencia de un servidor, que proporciona el servicio, y una serie de clientes que realizan peticiones al servidor y reciben respuestas del mismo.

En la Figura 4.1 se muestra de forma general la estructura de un chatbot (independientemente del framework conversacinal escogido) el cual está integrado en una web. Por un lado tendremos la parte visual del proyecto (frontend), con la que la usuaria interactuará, para ello debemos tener una pagina web, la cual tendrá incluida una interfaz que permita escribir mensajes, visualizaros y conectarse con el servidor del framework conversacional. Dicha pagina web será cliente del servidor web. Por otro lado existe la parte del servidor del framework conversacional (backend), donde, la interfaz descrita anteriormente será cliente de éste. Dicho servidor debe poder recibir mensajes, comprenderlos y devolver una respuesta al chatbot, la cual mostrará.

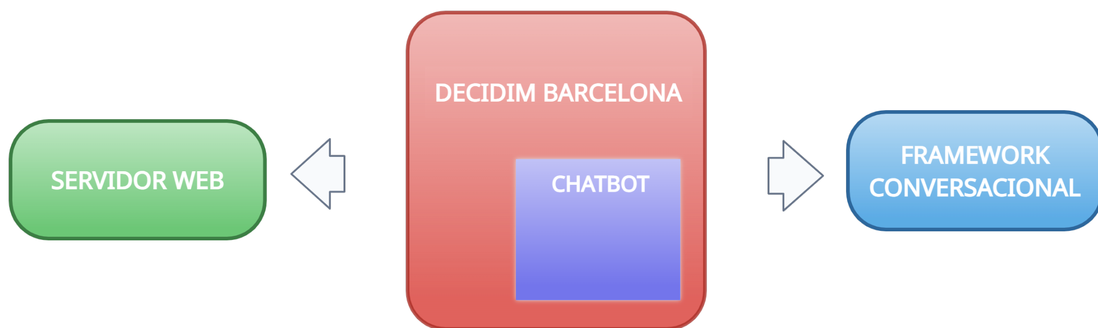

Figura 4.1: Estructura general de un chatbot con integración en una página web

4.2. Diseño del chatbot

4.2.1. Descripción de sus componentes

De forma general, Rasa se divide en dos componentes, el `training data`, que son los datos con los que el chatbot entrena y el `dominio`, que es un resumen de todo lo que conoce el agente. El dominio define el universo en el que opera el agente conversacional. Especifica los intents, entidades, slots, respuestas, formularios y acciones que el chatbot debe conocer y que puede usar durante la conversación. También define una configuración para las sesiones de conversación [12]. Una sesión de conversación representa el diálogo entre el agente y la usuaria. Una sesión es creada cada vez que una usuaria empieza una conversación con el chatbot o cuando ha pasado cierto tiempo de inactividad. Por otro lado, el training data, son los datos de entrenamiento del agente. Estos datos se dividen en dos subgrupos, por una parte tenemos los datos de entrenamiento de NLU. Estos datos de entrenamiento se generan mediante una gran cantidad de ejemplos de posibles mensajes de la usuaria. Mediante todos estos datos se puede entrenar al modelo para que el chatbot sea capaz de diferenciar entre un intent u otro a la hora de categorizar el mensaje de la usuaria, esto es, clasificar el mensaje de la usuaria en uno de los intents definidos. Cuantos más ejemplos y más variados, mejor clasificará el intent. En segundo lugar, están los datos de entrenamiento de conversación. Estos datos se especifican mediante las *stories* y *rules* (posibles flujos de conversación usados para entrenar al agente). Las stories servirán para entrenar al chatbot y que en conversaciones que se salen del flujo definido en las rules y las stories pueda predecir que acción tomar. Para predecir la siguiente acción a tomar se especifican las policies en el archivo `config.yml`.

Podemos destacar 4 archivos que se generan al crear un chatbot Rasa. Estos archivos son: `nlu.yml`, `stories.yml`, `domain.yml`, `config.yml`

El archivo `nlu.yml` (ver Figura 4.2) es fundamental para la capacidad del chatbot de entender a la usuaria (Natural Language Understanding). Aquí se listarán los intents, que son las intenciones o propósitos que buscaremos captar de las usuarias, por ejemplo, un mensaje de “buenos días” tiene como intención “saludar”, mientras que un mensaje de “¿cómo estará el clima mañana?” tiene la intención “consultar clima”. En este archivo se define el `training data`, explicado anteriormente. Por tanto, es importante proveer al `nlu.yml` de una amplia variación de ejemplos con todas las formas en las que la usuaria podria expresarse, pero sin sobrepasarse con ejemplos muy similares ya que solo dificultaría el entrenamiento. Además en cada intent podemos indicar ciertas entities que el chatbot tiene que extraer del mensaje de la usuaria, como por ejemplo, si la usuaria escribe “Me puedes dar un proceso de Horta?”, en este caso el chatbot no solo tiene que ser capaz de entender el mensaje, sino de extraer Horta como entity. Dependiendo de la versión de RASA la sintaxis variará mínimamente.

```
- intent: despedirse
  examples: |
    - Adeu
    - Adios
    - Hasta otra!

- intent: agradecer
  examples: |
    - Gracias
    - Muchas gracias
    - Te lo agradezco

- intent: donde_estoy
  examples: |
    - Que es esto?
    - Donde estoy?
    - Que hago ahora?
    - Que puedo hacer aqui?
    - No se que hacer.
```

Figura 4.2: Extracto del fichero nlu.yml donde vemos la estructura del mismo

En el archivo `stories.yml` (ver Figura 4.3) se escriben las diferentes interacciones entre el chatbot y la usuaria con las que se entrenará al asistente. Este archivo es usado para interacciones complejas con la usuaria, donde hay más de un mensaje de la usuaria. Se deben incluir múltiples ejemplos de historias con diferentes caminos o finales, con el objetivo de que el chatbot pueda predecir cuál es la mejor acción a tomar después de una interacción con la usuaria.

Muy relacionado con este fichero existe el archivo `rules.yml`, en este fichero se plasma de igual manera interacciones con la usuaria. La diferencia entre ambos archivos es que en el archivo de `rules.yml` las interacciones son más simples. Son interacciones a partir de un único mensaje de la usuaria. Por ejemplo puedes definir una regla que siempre que se reciba un intent "saludar", el chatbot muestra por pantalla "Hola, ¿Cuál es tu nombre?".

El archivo `domain.yml` es el lugar donde quedan registrados todos los componentes de entrenamiento del chatbot. Por un lado tenemos una lista con el nombre de todos los intents de la usuaria que hemos escrito en el archivo `nlu.yml`, y por otro lado una lista de todas las entities (recordemos que son partes de los intents con información relevante, como son fechas, nombres de ciudades, etc). También tenemos un registro de los diferentes `slots`, o memoria del chatbot, y otra lista con el nombre de todas las custom actions, o acciones personalizadas, implementadas en `actions.py`. Finalmente, ocupando gran parte del archivo tenemos las respones, o respuestas, que son mensajes predeterminados que se vinculan a ciertos intents, de forma que cuando el chatbot recibe un intent concreto lanza una response concreta.

Por último, tenemos el archivo `config.yml`, donde destacamos el apartado de pipeline y policies. En el pipeline se definen los procesos que se le aplicarán al texto de entrada. Este apartado de la configuración está estrechamente relacionada con el NLP, ya que recordemos que partiendo de un texto desestructurado se generaban

```
# saludar
  - story: saludar and discoverability
    steps:
    - intent: saludar
    - action: utter_saludar
    - intent: discoverability
    - action: utter_discoverability

  - story: saludar and discoverability2
    steps:
    - intent: saludar
    - action: utter_saludar
    - intent: quien_eres
    - action: utter_quien_eres
    - intent: discoverability
    - action: utter_discoverability
```

Figura 4.3: Extracto del fichero archivo `stories.yml` donde vemos la sintaxis de una story

datos estructurados que el chatbot puede procesar. En la Figura 4.4 se muestra un extracto del pipeline utilizado en este proyecto, cabe destacar que estas componentes del pipeline se ejecutan una tras otra. Cada parte del pipeline es un componente diferente que ayuda en las tareas de Lenguaje Natural, esto es:

1. `Tokenizers`: La tokenización consiste en separar el texto en entidades llamadas tokens, para facilitar el procesamiento de éste.

2. `Featurizers`: Estos componentes convierten texto en features, que son representaciones numéricas del texto, para su uso en otros algoritmos.

3. `Classifiers`: Los clasificadores se encargarán de decidir a qué intent corresponden los mensajes de la usuaria.

De igual manera, en el `config.yml` se listan las políticas (policies) bajo las cuales se regirá la conversación. El chatbot usa estas políticas para decidir cual será la siguiente acción a ejecutar. Hay 3 políticas predefinidas para todos los chatbots de Rasa. En la Figura 4.5 se muestran las tres políticas que por defecto están implementadas en un chatbot de Rasa:

1. `RulePolicy`: Decide la acción siguiente si el intent recibido concuerda con alguna de las reglas (rules) implementadas.

2. `MemoizationPolicy`: Decide la acción siguiente si las interacciones previas ocurrieron en las historias (stories) de entrenamiento.

3. `TEDPolicy`: Usa una red neuronal recurrente para predecir la mejor acción a tomar, basándose en los datos de entrenamiento.

```yaml
pipeline:
  - name: "WhitespaceTokenizer"
    intent_tokenization_flag": False
    token_pattern": None
  - name: RegexEntityExtractor
    use_lookup_tables: True
    use_regexes: True
    use_word_boundaries: True
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
    constrain_similarities: true
```

Figura 4.4: Extracto del fichero config.yml donde se puede ver una parte de mi configuración del pipeline

Cada política tiene un principio de funcionamiento diferente, y la combinación de ellas hace que el chatbot sea más robusto y pueda tomar mejores decisiones. En la Figura 4.5 podemos observar cómo las políticas funcionan con una jerarquía de prioridad, donde las que están definidas antes, se ejecutan antes. De forma que una política solo se aplicará si la anterior no ha sido capaz de predecir una acción.

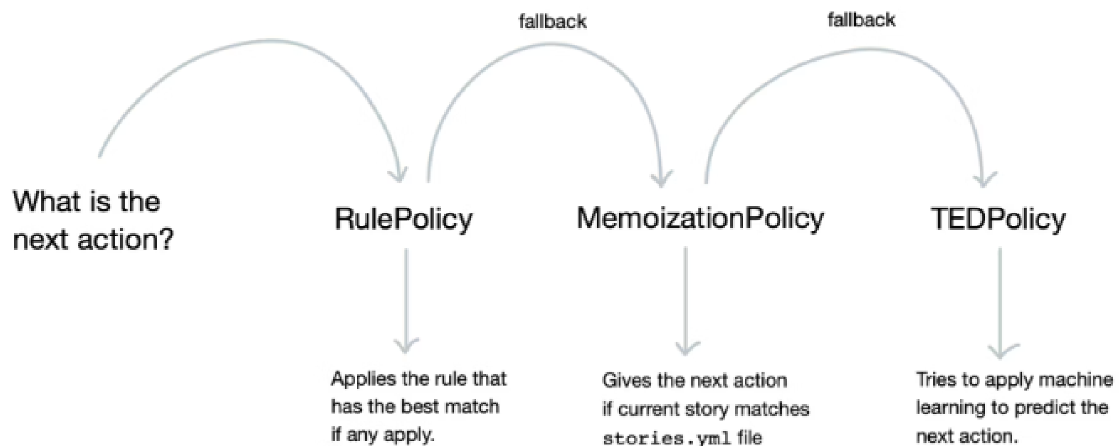

Figura 4.5: Diagrama de funcionamiento de las policies

4.2.2. Intents del chatbot de Decidim.bcn

Como ya se introdujo en la `Sección` 2.3, los `intents` representan los diferentes mensajes que la usuaria puede enviar al chatbot. Una usuaria podría mandar un mensaje, por ejemplo, de “buenos días”. Dicho mensaje se podría clasificar como intent de tipo “saludar”. De esta forma, como hemos visto anteriormente, deberíamos dotar al chatbot de una gran cantidad de ejemplos de mensajes de saludo. Estos ejemplos deberán estar adaptados a la manera de hablar de las personas usuarias, en este caso usando un lenguaje coloquial. Algunos ejemplos útiles podrían ser: “Hola”, “Heey!”, “Hola, buenas”. Todos estos diferentes mensajes estarían clasificados con el mismo intent, ya que la intención de ese mensaje es saludar.

A continuación se muestra una lista con los intents utilizados en este proyecto, los cuales se encuentran definidos, como se puede observar en la Figura 4.6, en el domain.yml (como una lista de intents) y en el fichero nlu.yml (como datos de entrenamiento de cada intent). Cabe destacar que estos intents tienen una correspondencia directa con las funcionalidades del chatbot presentadas en el `Capítulo` 3:

```
intents:
- afirmar
- agradecer
- negar
- despedirse
- saludar
- dar_nombre
- consejo_de_barrio
- debate
- encuentros
```

Figura 4.6: Extracto del archivo domain.yml, sección de intents

1. Intents sociales (Interacciones sociales definidas en el `Capítulo` 3)
   1) `afirmar`: La usuaria afirma.
   2) `agradecer`: La usuaria agradece.
   3) `negar`: La usuaria niega.
   4) `despedirse`: La usuaria se despide.
   5) `saludar`: La usuaria saluda.
   6) `dar_nombre`: La usuaria indica al chatbot su nombre.
2. Intents de definiciones
   1) `consejo_de_barrio`: La usuaria pregunta al chatbot qué es un consejo de barrio.

```yaml
- intent: saludar
  examples: |
    - Hola
    - Hey
    - Buenas
    - Buenos dias
```

Figura 4.7: Extracto del archivo nlu.yml, intent saludar

2) `debate`: La usuaria pregunta al chatbot qué es un debate.
3) `encuentros`: La usuaria pregunta al chatbot qué es un encuentro.
4) `comentarios`: La usuaria pregunta al chatbot qué es un comentario.
5) `iniciativas_ciudadanas`: La usuaria pregunta al chatbot qué es una iniciativa ciudadana.
6) `organos_de_participacion`: La usuaria pregunta al chatbot qué es un órgano de participación.
7) `presupuesto`: La usuaria pregunta al chatbot qué es un presupuesto.
8) `procesos_participativos`: La usuaria pregunta al chatbot qué es un proceso participativo.
9) `propuestas`: La usuaria pregunta al chatbot qué es una propuesta.
10) `quien_puede_participar`: La usuaria pregunta al chatbot quien puede participar en los procesos participativos.
11) `como_puedo_participar`: La usuaria pregunta como puede participar en los procesos participativos.
12) `decidim`: La usuaria pregunta al chatbot qué es Decidim.
13) `signIn`: La usuaria pregunta al chatbot como registrarse o iniciar sesión.

3. Intents de localización

1) `dar_barrio`: La usuaria le indica su barrio al chatbot.

4. Intents de proceso participativo

1) `devolver_proceso_participativo`: La usuaria solicita información sobre un proceso participativo (indicando la entidad neighborhood_location o no).
2) `debate_en_proceso_participativo`: La usuaria pregunta si el último proceso participativo que el chatbot le ha proporcionado dispone de algún debate.

```
- intent: decidim
  examples: |
    - Define decidim
    - Qué es decidim

- intent: comentarios
  examples: |
    - Define comentarios
    - Que son comentarios?
```

Figura 4.8: Extracto del archivo nlu.yml, intents de definición

```
- intent: dar_barrio
  examples: |
    - Soy de [Montjuic](barrio)
    - Soy del barrio de [Nou Barris](barrio)
    - Resido en [Sants](barrio)
    - Resido en el barrio [Sant Andreu](barrio)
    - Me interesa el barrio [Les Corts](barrio)
    - Estoy interesado en [Guinardó](barrio)
    - De [Horta](barrio)
```

Figura 4.9: Extracto del archivo nlu.yml, intent dar_barrio

3) `encuentro_en_proceso_participativo`: La usuaria pregunta si el último proceso participativo que el chatbot le ha proporcionado dispone de algún encuentro.
4) `propuesta_en_proceso_participativo`: La usuaria pregunta si el último proceso participativo que el chatbot le ha proporcionado dispone de alguna propuesta.
5) `presupuesto_en_proceso_participativo`: La usuaria pregunta si el último proceso participativo que el chatbot le ha proporcionado dispone de algún presupuesto.
6) `resumen_encuentros`: La usuaria solicita un resumen de los encuentros del proceso participativo de la página en la que se encuentra.
7) `ordenar_propuestas`: La usuaria solicita las últimas propuestas del proceso participativo de la página en la que se encuentra.
8) `propuestas_amigos`: La usuaria solicita propuestas que sus amistades siguen del proceso participativo de la página en la que se encuentra.

5. Otros Intents

   1) `discoverability`: La usuaria solicita que puede hacer el chatbot por él.

```
- intent: devolver_proceso_participativo
  examples: |
    - Me puedes ofrecer algun proceso participativo, en [Horta](barrio)?
    - Me puedes ofrecer algun proceso participativo, en [mi](mencion) barrio?
    - Me puedes dar el [ultimo](mencion) proceso participativo del que hablamos?
    - Que hay nuevo en los procesos participativos, en la zona de [Sants](barrio)?
    - Ha salido algun proceso participativo nuevo, cerca de [Montjuic](barrio)?
    - Cual es el ultimo proceso participativo que ha salido, cerca de [Guinardó](barrio)?
    - Que proceso participativo me recomiendas, en [Gracia](barrio)?
    - No sé que procesos participativos hay, en la zona de [Nou Barris](barrio).
    - Me puedes ayudar a escoger un proceso participativo, en [Sant Andreu](barrio)?
```

Figura 4.10: Extracto del archivo nlu.yml, intent devolver_proceso_participativo

2) `quien_eres`: La usuaria le pregunta al chatbot quién o qué es.
3) `donde_estoy`: La usuaria le pregunta al chatbot donde está.
4) `iniciar_conv`: Intent que sirve para iniciar la conversación (este intent no está definido en el nlu ya que no es un intent que la usuaria pueda escribir, únicamente se usa en posts externos al servidor Rasa para dar la sensación a la usuaria de que el chatbot le habla primero).
5) `cambio_pagina`: Intent que sirve para indicar que la usuaria ha cambiado de página (este intent no está definido en el nlu ya que no es un intent que la usuaria pueda escribir, únicamente se usa en posts externos al servidor Rasa para controlar cuando la usuaria cambia de página).
6) `discoverability_conceptos`: Intent que sirve para mostrar una lista de conceptos.
7) `discoverability_extendida`: Intent que sirve para mostrar la 2a parte de las funcionalidades del chatbot.
8) `resolver_mencion`: Intent que sirve para obtener el item de una lista que el chatbot ha ofrecido a la usuaria.

4.2.3. Entities del chatbot de Decidim.bcn

Como se introdujo en la `Sección` 2.3, dentro de los intents (que definen los mensajes de la usuaria), veremos definidas en algunos casos, `entities`. Las entities son datos relevantes que queremos obtener de los mensajes de la usuaria. En este proyecto se han utilizado las siguientes:

1. `barrio`: Esta entity se usa tanto en el intent "dar_barrio" como en el intent "devolver_proceso_participativo" para indicar un barrio.

2. `user_name`: Esta entity se utiliza en el intent "dar_nombre" para indicar el nombre de la usuaria. Además se utiliza en el intent "iniciar_conv" enviado desde el widget, indicando el nombre de la usuaria, cuando éste está registrado.

3. `contexto`: Esta entity se usa en el intent "cambio_pagina" enviado desde el widget para indicar en que página se encuentra la usuaria.

4. `usuario_registrado`: Esta entity se utiliza en el intent "iniciar_conv" enviado desde el widget, indicando si la usuaria está registrado o no.

5. `mencion`: Esta entity se usa en el intent "resolver_mencion" para indicar una mención a un elemento de una lista. Además se usa en el intent "devolver_proceso_participativo" si la usuaria quiere preguntar por su barrio (ver Figura 4.10).

6. `slug_actual`: Esta entity se usa en el intent "cambio_pagina" para indicar el slug del proceso en el que se encuentra la usuaria.

En la Figura 4.11 podemos ver un ejemplo de la definición de un intent en el que se extrae la entity "user_name". Hay varias formas de extraer dichas entities, pero siempre se necesita de un extractor de entities. La forma predeterminada de extraerlas en Rasa es mediante DIET Classifier (Dual Intent Entity Transformer Classifier) . Como hemos visto anteriormente en la `Sección` 4.2.1, un clasificador es un componente definido en el archivo `config.yml` el cual se encarga de decidir a qué intent corresponden los mensajes de la usuaria. Este componente no solo clasifica intents sino que además es capaz de clasificar las entidades. Es el componente más usado en Rasa para esta tarea.

```
- intent: dar_nombre
  examples: |
    - Mi nombre es [Jose](user_name)
    - Me llamo [Victor](user_name)
    - Me llamo [Isaac](user_name)
    - Soy [Marta](user_name)
    - [Inmaculada](user_name)
    - [Maite](user_name) es mi nombre
```

Figura 4.11: Extracto del fichero nlu.yml donde se pueden apreciar la definición de la entity user_name.

La forma más básica de ayudar al extractor de entities es usando sintaxis que se detalla en la Figura 4.11, donde en el mismo intent se especifican mediante varios ejemplos los valores que puede tener la entity. Pero para ayudar al extractor de entidades y/o para solo aceptar los valores que te interesan existen los Sinónimos, las Expresiones Regulares y las Lookup Table. No obstante, en este proyecto solamente se han utilizado las lookup table.

Las lookup tables son un componente muy interesante y nos sirve para tener un conjunto de valores concretos que puede tener una entidad. En la Figura 4.12 se muestra un extracto del fichero `nlu.yml` donde se definen dichas tablas. En este caso dicha tabla se usa para controlar los barrios aceptados y obviar el resto de localizaciones que no aparezcan en la tabla. Por ejemplo cuando un ciudadano le envía el mensaje al chatbot "Soy de Horta". Además en este proyecto se ha utilizado otra lookup table, ésta ha sido usada para controlar los nombres de las personas usuarias. De la misma forma que el ejemplo que vemos en la Figura 4.12, se han

incluido una gran cantidad de ejemplos para que el chatbot únicamente entienda dichos nombres.

```
- lookup: barrio
  examples: |
    - Ciutat Vella
    - Eixample
    - Sants
    - Montjuic
    - Les Corts
    - Sarria
    - Sant Gervasi
    - Gracia
    - Horta
    - Guinardó
    - Nou Barris
    - Sant Andreu
    - Sant Marti
```

Figura 4.12: Extracto del fichero nlu.yml donde podemos ver la estructura de una lookup table

4.2.4. Slots del chatbot de Decidim.bcn

Los `slots` los podríamos definir de forma muy simple como la memoria del chatbot. Son aquellas entities que quieres guardar durante todo el transcurso de la conversación. Los slots pueden guardar cualquier tipo de dato, desde texto, números, booleans, hasta listas. Pero en los slots destacamos principalmente 2 categorías dependiendo de como consigan el valor:

1. `from_entity`: Son aquellos slots cuyo valor proviene directamente de otras entities. Siguiendo el ejemplo anterior de la Figura 4.11, la entity "user_name", podría estar relacionada con un slot "user_name" definido en el domain si indicamos que tiene que recibir el valor de la entidad mediante la sintaxis *from_entity: entity*. Podemos ver un ejemplo usado en el proyecto en la Figura 4.13. El valor de estos slots se puede guardar de forma automática (usando *from_entity: entity*) o manual (asignando un valor concreto mediante una custom action).

2. `custom`: Son aquellos slots cuyo valor no tiene por que estar vinculado a ninguna entidad. El valor de estos slots se debe guardar de forma manual mediante custom actions. Por ejemplo para guardar el valor de una variable que indique si la usuaria está registrado o no.

Una funcionalidad muy interesante de los slots es que pueden influir en el flujo de una conversación dependiendo de su valor. Existen un tipo de slots llamados

```
user_name:
  type: text
  influence_conversation: false
  mappings:
  - type: from_entity
    entity: user_name
```

Figura 4.13: Ejemplo de estructura de slots vinculados a entities del chatbot de Decidim.barcelona (archivo domain.yml)

categorical, estos slots solo aceptan una serie de valores indicados en la definición del propio slot, en el fichero `domain.yml`. Para entenderlo mejor propongo el ejemplo de mi propio proyecto en la Figura 4.14 donde se puede apreciar que el slot "user_name" de tipo categorical puede tener únicamente dos valores: REGISTRADO o NO_REGISTRADO y además el atributo *influence_conversation* es true. De esta forma podemos utilizar este slot para controlar diferentes flujos en la conversación mediante *stories* o *rules*. En la Figura 4.15 se muestra un ejemplo sencillo donde se controla el flujo de la conversación mediante 2 slots, de forma que solamente se aplique esta regla en el caso de que la usuaria la usuaria esté registrado en la plataforma y que se encuentre en la página "Listado Procesos". En caso de que dichas condiciones se cumplan y el chatbot reciba un intent "cambio_pagina", ejecutará la acción "action_control_flow"

```
usuario_registrado:
  type: categorical
  values:
    - REGISTRADO
    - NO_REGISTRADO
  influence_conversation: true
  mappings:
  - type: from_entity
    entity: usuario_registrado
```

Figura 4.14: Ejemplo de estructura de slots de tipo *categorical* del chatbot de Decidim.barcelona (archivo domain.yml)

Para definir las conversaciones previamente mencionadas en el `Capítulo` 3, necesitamos de una serie de slots para controlar el estado y el flujo de la conversación, en las cuales podamos guardar ciertos datos. Los cuales son:

1. `barrio`: Este slot se usa para acordarse del barrio de la usuaria durante la conversación.

2. `contexto`: Este slot sirve para guardar la página en la que se encuentra la usuaria.

```
- rule: ofrecer procesos
  condition:
  - slot_was_set:
    - usuario_registrado: "REGISTRADO"
  - slot_was_set:
    - contexto: "LISTADO PROCESOS"
  steps:
  - intent: cambio_pagina
  - action: action_control_flow
```

Figura 4.15: Ejemplo de estructura de una rule del chatbot de Decidim.barcelona (archivo domain.yml)

3. `slug_actual`: Este slot sirve para guardar el slug de un proceso (el slug es el id por el cual se busca un proceso en la API de Decidim.barcelona).

4. `lista_ofrecida`: Este slot sirve para guardar una lista ofrecida a la usuaria y posteriormente poder acceder a ella.

5. `user_name`: Este slot sirve para guardar el nombre de la usuaria durante la conversación.

6. `usuario_registrado`: Este slot sirve para controlar el tipo de usuaria que es (habitual o nuevo).

7. `estado_contexto`: Este slot es un diccionario donde las keys son las diferentes paginas que puede visitar la usuaria y los valores son booleans. Este intent sirve para controlar si es la primera vez que una usuaria llega a dicha página o ya ha estado antes.

4.2.5. Actions del chatbot de Decidim.bcn

Las `actions` son las diferentes acciones que puede hacer el chatbot. Despues de cada mensaje que la usuaria escriba, el modelo entrenado hará una predicción de la siguiente o siguientes acciones a ejecutar en base al intent que ha predicho el modulo NLU y las entidades extraidas de dicho intent.

Se destacan los siguientes tipos de actions en RASA: Responses, Custom Actions, Forms y Default Actions.

Las `responses` son mensajes predeterminados que se vinculan a ciertos intents, de forma que cuando el chatbot recibe un intent concreto lanza una respuesta concreta. En la Figura 4.16 podemos ver su sintaxis, de forma habitual las responses tendrán únicamente un atributo texto, aunque al incorporar más texto a una misma response permite tener respuestas más variadas, ya que el agente elegirá una de forma aleatoria. En la Figura 4.16 podemos ver un ejemplo de esto, además podemos observar la estructura para usar el valor de una entity en las respuestas.

```
utter_comentarios:
- text: Los comentarios son contribuciones de los usuarios al hilo de una *propuesta*
  buttons:
  - title: Qué son las propuestas
    payload: /propuestas
  - title: Qué es un debate
    payload: /debate
```

Figura 4.16: Ejemplo de response del chatbot de Decidim.barcelona (archivo domain.yml)

Las `custom actions` son las más interesantes de todas. Como su nombre indica son acciones *customizables*, las cuales se escriben en lenguaje python. Puedes programar la custom action para que desarrolle la acción que mejor te convenga, conectarse a una bdd, hacer un cálculo en base a datos que la usuaria ha escrito, hacer una consulta a una API, mostrar por pantalla un texto concreto, etc...

También existen los `forms`, son un tipo especial de acción personalizada. Si, por ejemplo, tienes algún diseño de conversación en el que esperas que el asistente te pida un conjunto específico de información, se pueden utilizar formularios. Pero debido a que en este proyecto no ha sido necesario, no se mencionarán más adelante.

Las `default actions` son acciones que el gestor de diálogos incorpora por defecto. La mayoría de ellas se predicen automáticamente en función de determinadas situaciones de conversación. Como por ejemplo "action listen", la cual es una acción predeterminada que simplemente espera un *input* de la usuaria. Es posible personalizar las default actions. Si definimos una acción en el archivo `actions.py` con el mismo nombre que la default action que queremos personalizar podemos cambiar o alterar su funcionamiento. Esta funcionalidad no ha sido implementada en este proyecto, no obstante es un aspecto de las default actions que es interesante comentar. Habitualmente se suele cambiar el comportamiento de acciones como *action_default_fallback*, la cual se ejecuta cuando el chatbot no tiene la suficiente certeza de que acción utilizar. En este caso se podría *customizar* dicha acción para que mostrara varios mensajes por pantalla, por ejemplo.

Las custom actions implementadas en este proyecto son las siguientes:

1. `action_procesos_participativos`: Esta acción se llama en el contexto del intent "dar_proceso_participativo", para responder a la usuaria en la petición de información de un proceso. Para ello, la acción accede a la entity "barrio", si dicha entity tiene algún valor, filtra según la zona. También accede al slot "usuario_registrado" y a la entity "mencion", si la usuaria está registrado y ha escrito "mi barrio", se hará una query a la base de datos para obtener su barrio y se devolverá información de un proceso en dicho barrio. De forma alternativa, si la usuaria ha escrito "último barrio" se le devolverá la información del último barrio al que ha navegado o del último barrio ofrecido por el chatbot.

2. `action_mirar_en_proceso_participativo`: Esta acción se llama en el contexto de los intents "debate_en_proceso_participativo" , "encuentro_en_proceso_participativo",

"propuesta_en_proceso_participativo", "presupuesto_en_proceso_participativo", para responder a la usuaria respecto si el proceso tiene alguna de las caracteristicas anteriores. Para ello se accede al slot "slug_actual" y se hace query a la Api de Decidim con dicho slug para obtener los componentes.

3. `action_show_context_and_id`: Esta acción se llama en el contexto del intent "donde_estoy" o en un cambio de página para dar información de la página en la que está la usuaria. Para ello, se accede al valor del último intent, si se ha llamado mediante un cambio de página, se controla que sea la primera vez que llega a dicha página mediante el slot "estado_contexto", en caso de serlo se le muestra una descripción de la página. En caso de que el intent sea "whereami" se le muestra la descripción siempre.

4. `action_control_flow_proposals`: Esta acción se llama cuando una usuaria registrado llega a la página de propuestas de un proceso participativo para mostrar lo que puede ofrecer el chatbot. Para ello se controla que solo se muestre una vez mediante el slot "estado_contexto" y en caso de ser la primera se le muestran la opción de obtener las últimas 3 propuestas o la opción de obtener propuestas de interés de sus amistades.

5. `action_last_3_proposals`: Esta acción se llama cuando la usuaria hace clic en una de las opciones en la página de propuestas. Para ello se obtiene el valor del slot "slug_actual" y se hace una query a la API de Decidim.barcelona para sacar las últimas propuestas del proceso actual.

6. `action_propuestas_amigos`: Esta acción se llama cuando la usuaria hace clic en una de las opciones en la página de propuestas. Para ello se obtiene el valor del slot "user_name" y se obtienen sus amistades de la base datos. Posteriormente se obtienen las propuestas que siguen dichas amistades de otra colección de la base de datos.

7. `action_offer_sumarization_encuentros`: Esta acción se llama cuando una usuaria registrado llega a la página de encuentros de un proceso participativo o cuando una usuaria solicita un resumen con el intent "resumen_encuentros" para ofrecer los encuentros a los cuales puede hacer un resumen. Para ello se controla que solo se muestre una vez mediante el slot "estado_contexto" y en caso de ser la primera vez obtiene el valor del slot "slug_actual" y hace una query a la base de datos para obtener los encuentros de dicho proceso con comentarios y guardar la lista de encuentros ofrecida en el slot "lista_ofrecida".

8. `action_sumarization`: Esta acción se ejecuta en el contexto del intent "mencion", para ofrecerle un resumen del encuentro seleccionado. Para ello obtenemos el valor del slot "lista_ofrecida", obtenemos el slug del encuentro de dicha lista mediante la entity "mencion" la cual hará referencia al encuentro con palabras como "primero", "último", etc... Finalmente se busca en la base de datos los comentarios de dicho encuentro y se llama a un método que permite resumir texto.

9. `action_control_flow`: Esta acción se llama cuando una usuaria registrado llega a la página de todos los procesos participativos. Envía a la usuaria un mensaje para preguntarle si quiere obtener un proceso participativo según sus intereses (los intereses son palabras clave guardadas en la bdd para definir la categoría de procesos participativos que le interesan a dicha usuaria).

10. `action_offer_pp_tematica`: Esta acción se llama cuando una usuaria dice que sí a la acción "action_control_flow". Accede al slot "user_name" y una query a la bdd para obtener los intereses de la usuaria y posteriormente query a la API de Decidim.barcelona para obtener procesos participativos en base a esos intereses.

11. `action_validar_neighbor`: Esta acción se llama en el contexto del intent "dar_barrio", para controlar que el barrio obtenido sea correcto. Para ello obtenemos la entity "barrio" y la comparamos con una lista predefinida de barrios aceptados. En caso de que el barrio esté aceptado se guarda el valor en el slot "barrio".

Capítulo 5

Implementación y Resultados

5.1. Tecnologias Utilizadas

Este apartado resume brevemente las diferentes tecnologías utilizadas en este proyecto. Como se ha comentado en la `Sección` 4.1, para lograr una mayor inmersión de la usuaria a la hora de utilizar el chatbot, es necesario una página web que emule el contenido de la plataforma digital, en este caso Decidim.barcelona. Para ello en este proyecto se ha decido la utilización de Flask debido a ciertas características que se explicarán más adelante en la `Sección` 5.3.2. Además, es necesario dotar al proyecto de un framework para el desarrollo del chatbot, en este caso se ha creido que la mejor opción era Rasa. Finalmente, en la parte de las consultas de datos se ha utilizado la API de Decidim.barcelona para todos aquellos datos provenientes de los procesos participativos y finalmente, una base de datos en MongoDB para todos aquellos datos relacionados con las comunidades virtuales y el resumen de debates. Dicha base de datos simula la bbdd real de Decidim.bcn y nos permite así hacer determinadas consultas desde el chatbot.

5.1.1. Frameworks Conversacionales

A la hora de diseñar e implementar un agente conversacional existen una serie de plataformas conversacionales que te ayudan a llevar a cabo tu objetivo. Como hemos comentado a lo largo de la memoria, el framework escogido para este proyecto ha sido Rasa, no obstante existen otros como DialogoFlow, Luis, Watson que explicaremos brevemente en esta sección.

Dialogflow es una plataforma potenciada por Google con comprensión del lenguaje natural (NLU) que te facilita el diseño de una interfaz de usuario de conversación y su integración a tu aplicación para dispositivos móviles, aplicaciones web, dispositivos, bots, sistemas de respuesta de voz interactiva y más [13].

Luis es una plataforma conversacional de Microsoft. Es un servicio conversacional de inteligencia artificial basado en la nube que aplica inteligencia de aprendizaje automático personalizado a una conversación o un texto de lenguaje natural de

una usuaria para predecir el significado global y extraer información pertinente y detallada. [14]. LUIS se retirará el 1 de octubre de 2025. No obstante, a partir del 1 de abril de 2023, ya no se podrán crear recursos de este servicio.

Watson es una plataforma conversacional potenciada por IBM. Se basa en modelos de deep learning, machine learning y procesamiento de lenguaje natural (NLP) para comprender preguntas, buscar las mejores respuestas y completar la acción prevista de la usuaria. Watson también utiliza la clasificación de intenciones y el reconocimiento de entidades para comprender mejor a los clientes en contexto y transferirlos a un agente humano cuando sea necesario [15].

En la Figura 5.1 podemos ver de forma detallada la comparación entre las características de los diferentes frameworks disponibles, a partir del análisis del estudio de [16]

Chatbot Offerings Rating Matrix	NLU		Graphic Dialog Node Management	Native Code Dialog Node Management	Machine Learning	Hosting	Cost	NLU API Capability	Enterprise Ready Scaling
	Intents	Entities							
IBM Watson Assistant	↗	↗	⇧	↗	↗	Commercial Cloud	➡	↗	⇧
Amazon Lex	↗	↗	⬇	⇧	↗	Commercial Cloud	↗	↗	↗
Microsoft Azure LUIS Bot Framework Composer / Emulator	↗	⇧	↗	⇧	↗	Commercial Cloud	↗	↗	⇧
Microsoft Power Virtual Agent	➡	➡	↗	↗	➡	Commercial Cloud	⬇	⬇	➡
Rasa	⇧	↗	⇧	ML Approach	⇧	Install Anywhere	Open Source	⇧	⇧
Cisco MindMeld	↗	↗	⬇	↗	↗	Install Anywhere	Open Source	↗	➡
Google DialogFlow ES	↗	↗	➡	⇧	↗	Commercial Cloud	↗	↗	↗
Google DialogFlow CX	↗	↗	⇧	⬇	↗	Commercial Cloud	⬇	↗	↗
NVIDIA Jarvis	⇧	⇧	⬇	⬇	⇧	Install Anywhere	Dedicated Hardware	↗	⇧

Figura 5.1: Comparativa de las características entre frameworks conversacionales

Como se puede ver en la Figura 5.1 Rasa destaca respecto al resto en la mayoría de características, sumado a que es un framework gratuito y a la experiencia de las tutoras en otros TFG donde se usaban otros frameworks, se llegó a la conclusión de que Rasa era el candidato perfecto para utilizar.

5.1.2. Servidores Web

Hay diversos frameworks que nos pueden servir para esta tarea, como Fastapi, Django y Flask, cada uno con sus ventajas y sus inconvenientes. `Fastapi` es un framework moderno y rápido creado especialmente para python, es un framework que destaca en que es genial para principiantes, es muy flexible a la hora de cómo estructurar el código y es fácil agregar, modificar o eliminar código. La principal desventaja de Fastapi se debe a que es un framewoek demasiado nuevo y por tanto,

tiene poca documentación y una comunidad pequeña. Por otro lado tenemos `Django`, el cual es el framework más grande y robusto de los mencionados en esta sección. Django es la mejor opción para proyectos grandes. Es el más utilizado para python gracias también a su extensa y clara documentación, a su gran comunidad y que muchos *features* están ya implementados. En este caso las desventajas se deben a que el código tiene muy poca flexibilidad, se estructura siempre de la misma forma y es complicado salir de dicha estructura. Además al tener muchas características ya implementadas la curva de aprendizaje es muy baja. Finalmente, está `Flask`. Es el segundo framework más popular en python. Es útil para principiantes, ya que es fácil para comenzar, es flexible y tiene una gran comunidad ya que no es un framework nuevo. En general es un framework sencillo y fácil de implementar, esa fue la decisión por la que decidí utilizar Flask

5.2. Arquitectura de Rasa

NLU Pipeline es la parte que se encarga de la clasificación de intents, la extracción de entities y la recuperación de respuestas. El componente de gestión del diálogo (Dialogue Policies), por otro lado, decide la siguiente acción en una conversación en función del contexto.

La Figura 5.2 ofrece una visión general de la arquitectura de Rasa. El Bot User (1), es la interfaz interactiva que utiliza la usuaria para comunicarse con el servidor de Rasa y es el encargado de captar los mensajes de la usuaria y mostrar las respuestas que el servidor envía. Este widget se conecta con el servidor de Rasa mediante un canal socket.io (2). Una vez conectados, el agente (3), que lo podemos entender como el controlador de esta arquitectura, recibe el mensaje de la usuaria. Este mensaje se lo transmite al NLU (4), el cual, como se ha mencionado anteriormente, transforma el mensaje en una estructura y extrae el intent y en caso de que también haya, las entities. Posteriormente el agente envía el resultado generado por el NLU al generador de diálogos (5), el cual decidirá la siguiente acción a ejecutar. En caso de que la siguiente acción sea una custom action, el agente se comunicará con el Action Server (6) y este le devolverá el mensaje de respuesta. A su vez, durante toda esta ejecución, el Tracker Store (7) se encarga de guardar en memoria la conversación. Cabe destacar que en las custom actions, utilizamos dicho tracker para poder obtener datos o eventos de la conversación, como por ejemplo, el último intent extraído del mensaje de la usuaria, el valor de un slot, el valor de la entity obtenida del último mensaje, etc..

5.3. Arquitectura del Sistema

Como se puede observar en la Figura 5.3, el sistema consta de un SERVER FLASK, el cual da servicio al componente DECIDIM BARCELONA. Integrado dentro de dicho componente está implementado el WIDGET RASA, que proporciona una interfaz gráfica e interactiva para la usuaria. Este widget es el cliente de

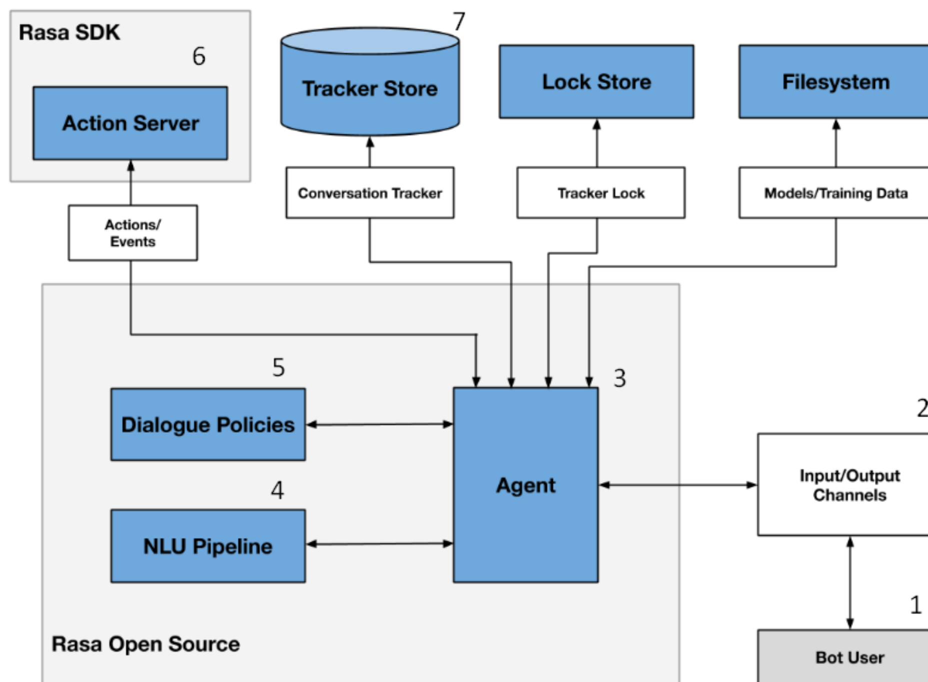

Figura 5.2: Arquitectura de un chatbot de Rasa

SERVER RASA con el que se comunica para enviar y recibir los mensajes entre la usuaria y el chatbot. Dicho servidor de Rasa está a su vez conectado al RASA ACTIONS, con el que se complementa, como hemos visto en la `Sección` 5.2, para ejecutar las custom actions. RASA ACTIONS será el que se conecte a API DECIDIM y a MONGODB para obtener los datos necesarios durante la conversación, utilizarlos en las custom actions y ofrecer los mensajes correspondientes a la usuaria.

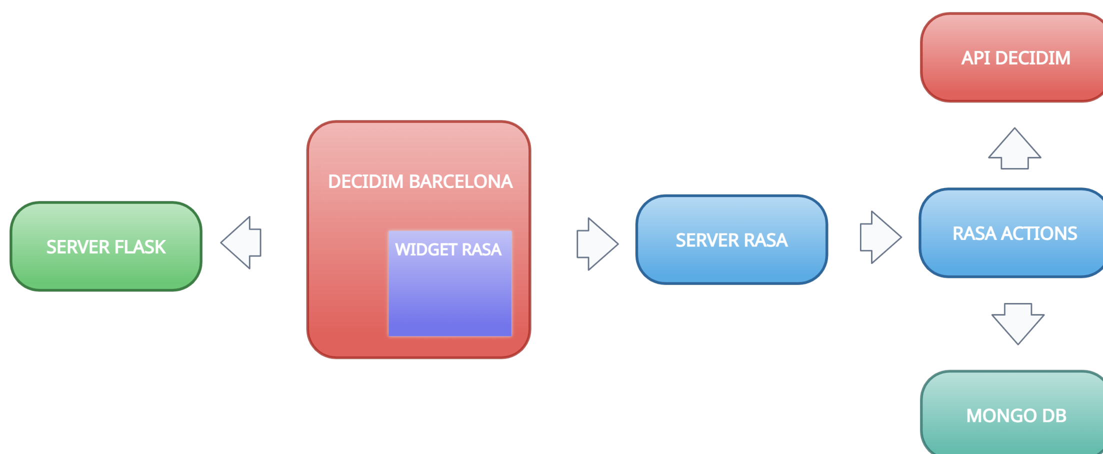

Figura 5.3: Diagrama del sistema

5.3.1. Decidim Barcelona

Empezando por la parte central de la Figura 5.3, tenemos DECIDIM. Dicho componente es, en este caso, una copia del HTML + css + js de la plataforma digital Decidim Barcelona, descargada directamente de la web original. Implementada en este proyecto para recrear el escenario en el cual estaría integrado el chatbot (el cual se explicará en la `Sección` 5.3.3. Para tener dicha copia lo que debemos hacer es posicionarnos en la página que deseamos copiar y clicar en:

botón derecho → guardar como... → página web completa

De esta forma nos descargaremos el HTML completo y los ficheros necesarios para que funcione correctamente, como el CSS, fotografías y archivos Javascript. Por último, lo único que falta para tener una copia operativa es la navegación. Para ello debemos repetir el proceso con las distintas paginas a las que se quiera poder navegar y cambiar aquellas partes de código de forma que ahora la navegación apunte a los ficheros de la copia local.

5.3.2. Flask

Llegados a este punto disponemos de una pequeña página web con su navegación incluida, el problema es que dicha pagina se ejecuta clicando directamente en los propios archivos HTML. Por tanto, el siguiente paso en este proyecto fue agregar un servidor en local que gestione la página web con diferentes endpoints[1], uno para cada archivo HTML y así poder navegar por la web desde dicho servidor local. Para ello usaremos el framework Flask.

Como vemos en la Figura B.1, para usar Flask debemos agregar a nuestra raiz del proyecto un fichero `app.py` el cual crea una aplicación Flask al ejecutarse, e indicar en ese mismo fichero los endpoints de cada archivo HTML. Para que Flask funcione tiene que saber donde están localizados los *templates* (ficheros HTML utilizados en el proyecto) que se mencionan en `app.py`. Se puede indicar la carpeta donde están situados los *templates* mediante el argument `template_folder`, y de igual manera se puede indicar la carpeta donde están situadas las imagenes y demás ficheros usados por el servidor con el argumento `static_path`.

Pese a que Flask es un framework muy flexible, como hemos mencionado al inicio de este apartado y te permite estructurar el código como te vaya mejor, en mi caso seguí la estructura base de Flask, como vemos en la Figura 5.4. Por tanto, estructuré el proyecto de forma que todos los HTML estuvieran en una carpeta `template`, donde Flask busca de forma predeterminada, y las imagenes y demás ficheros en la carpeta `static`, donde de igual forma, Flask busca de forma predeterminada.

Finalmente, simplemente lo ejecutaremos usando el comando *python app.py* desde la carpeta donde se encuentre instalado Flask.

[1] Un endpoint, en Flask, es una dirección del backend que se encarga de dar respuesta a una petición. En este caso ofrecer el servicio de visualizar la página asociada a dicho endpoint

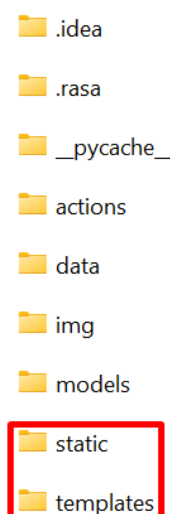

Figura 5.4: Estructura de archivos en Flask

5.3.3. Rasa Webchat

Para poder interactuar con el backend del chatbot de Rasa desde nuestro HTML necesitamos una interfaz (frontend del chatbot de Rasa) capaz de captar mensajes, enviarselos al servidor de Rasa y mostrar la respuesta que Rasa nos da.

El componente más utilizado para estas necesidades y que es apoyado oficialmente por Rasa es `Rasa Webchat` (https://github.com/botfront/Rasa-webchat). Con este widget podremos incorporar Rasa de forma fácil a nuestra web. Únicamente tenemos que añadir en el `body` el código base que se muestra en la Figura B.2 en el Anexo. En mi caso cambié ciertas componentes del widget (en la Figura B.3 en el Anexo, podemos ver el código del widget que se ha implementado en este proyecto). Cambié el lenguaje a español, cambié el *socketUrl* por "http://localhost:5005", ya que es el endpoint donde opera el servidor de Rasa y más detalles como el color, o el título para que el chatbot se vea mejor estéticamente.

Pero de todos los componentes hace falta destacar el `initPayload` y `params`. El `initPayload` es una funcionalidad de Rasa webchat que te permite mandar un intent a Rasa de forma automática la primera vez que se carga el widget, al iniciar una conversación, siguiendo la estructura que vemos en la Figura 5.5. Rasa por su parte recibirá este intent y ejecutará una acción u otra en base a *rules* y *stories*. Esta es la forma de hacer que el chatbot inicie una conversación con la usuaria. Además en mi caso controlo también si se trata de una usuaria logeado o de una usuaria nuevo en Decidim mediante la entity *user_logged* como vemos en la Figura 5.5 en el argumento *initPayload*, donde enviamos el intent "iniciar_conv". De esta forma, aunque está *hardcodeado*, el chatbot puede saber si se trata de una usuaria registrado o no, y ofrecer unas funcionalidades u otras. Para evitar esto, se debería pasar previamente por una página en la que la usuaria iniciara sesión. Debido a que no es el objetivo principal de este proyecto se ha creído que era una buena solución *hardcodearlo* para poder distinguir entre tipos de usuarias. Por otro lado tenemos

el argumento `params`. Este argumento según está explicado en la documentación de `Rasa Webchat` está pensado para indicar el tamaño de las imagenes, pero también para cambiar las opciones de almacenamiento del navegador (storage). Este último punto será clave para adquirir "session persistance".

La "session persistance" la definiriamos como la capacidad del chatbot de mantener la conversación aún cerrando el navegador, recargando la página o navegando por la web. Para conseguir dicha capacidad es necesario pasar como parámetro {storage : "local"} o {storage : "session"}, como vemos en la Figura 5.5, en el argumento `params`. Usando storage local guarda la conversación en el storage local del navegador, eso permite que aún cuando se cierre el navegador o se cambie de página se pueda mantener la conversación. Otra alternativa es el storage session, el cual guarda la conversación en el storage session del navegador y solo mantiene la sesion si se recarga la página.

Debido que en este proyecto no tenemos un chatbot estatico, sino que es esencial que la usuaria pueda navegar por la web mientras usa el chatbot es necesario que la conversación se guarde en el storage local del navegador. Además, para que la conversación tenga inicio y final y no dure "eternamente" cada vez que se carga el HTML de la página principal se llama a "localStorage.clear()" para resetear todo lo que hay guardado.

```
window.WebChat.default(
  {
    customData: { language: "es" },
    socketUrl: "http://localhost:5005",
    initPayload: '/iniciar_conv{"usuario_registrado": "REGISTRADO",
    "user_name": "Victor"}',
    // initPayload: '/iniciar_conv{"usuario_registrado": "NO_REGISTRADO"}
    ',
    title: 'Hola! Soy Sara',
    subtitle: 'Pregúntame sobre Decidim!',
    profileAvatar: '/static/female-call-center-operator.gif',
    inputTextFieldHint: 'Escribe algo..',
    params: {storage: "local"}
  },
```

Figura 5.5: Extracto de código del widget (fichero completo en la Figura B.3 del Anexo)

La estrategia anterior, en la cual usamos *initPayload*, sirve únicamente para el archivo *Decidim.barcelona.html*. Para el resto de archivos HTML que no corresponden a la página principal de Decidim, la implementación del widget varía un poco. Las dos variaciones importantes son: 1) No borramos el localStorage, ya que queremos que el widget sepa en que conversación tiene que seguir. 2) Además, añadimos un argumento nuevo llamado *onSocketEvent*, como se muestra en la Figura 5.6. Dentro de este argumento podemos especificar que cuando el widget reciba un evento de tipo "connect" haga algo. El evento "connect" sucede cada vez que se carga la página HTML y por consiguiente se carga el widget. Una vez suceda, como se puede observar en la Figura 5.6 se llama a la función de Javascript *sendContext()*, la cual, como vemos en la Figura B.4, gracias a que el *localStorage* guarda el id de la conversación, manda un intent de forma externa al servidor de Rasa, indicando mediante la entidad *context* la página en la que se encuentra la persona usuaria (página principal, página de procesos, página de encuentros de un proceso, etc...) para que el chatbot pueda tener esa información. En la Figura B.4 en el Anexo podemos ver la implementación de la *sendContext()* que se ejecuta cuando el widget recibe un evento de tipo "connect".

```
title: 'Hola! Soy Sara',
subtitle: 'Pregúntame sobre Decidim!',
profileAvatar: '/static/female-call-center-operator.gif',
inputTextFieldHint: 'Escribe algo..',
params: {storage: "local"},
onSocketEvent : {
  'bot_uttered': () => console.log('the bot said something'),
  'connect': () => sendContext(),
  'disconnect': () => doSomeCleanup(),
}
```

Figura 5.6: Extracto de código donde se observa "onSocketEvent"

5.3.4. Rasa Action Server

Rasa Action Server es un servidor complementario al servidor de Rasa, como hemos visto anteriormente en la `Sección` 4.2.1, es el servidor que ejecuta las custom actions y se conecta a las bases de datos. El framework Rasa divide su servidor en 2 para así dividir la carga de trabajo. Cuando el chatbot predice una custom action, el servidor de Rasa envía una solicitud POST al Action Server con un JSON que incluye el nombre de la acción que han predicho las políticas de diálogo (Dialogue Policies), el ID de la conversación, el contenido del tracker y el contenido del dominio. En el contenido del tracker destacamos los slots, las entities y el intent. Cuando el servidor de acciones termina de ejecutar una custom action, devuelve un JSON con los mensajes que se deben mostrar a la usuaria y una lista de eventos. A continuación, el servidor de Rasa devuelve las respuestas a la usuaria y añade los eventos al rastreador de conversaciones (tracker).

Los eventos son la forma en que internamente Rasa interpreta las conversaciones (definiendolo de forma muy básica se trata de un resumen de lo que ha pasado en la conversación, como si fueran *logs*), de forma que hay una gran variedad de ellos. No todos los eventos son devueltos por custom actions, ya que algunos los controla automáticamente Rasa. De todos ellos destacamos los siguientes para este proyecto:

1. `SessionStarted`: Este evento se ejecuta automáticamente al inicio de la conversación, reinicia la conversación vaciando el tracker.

2. `FollowupAction`: Este evento se ejecuta de forma manual y sirve para "forzar" la ejecución de una acción sin tener en cuenta las políticas de diálogo.

3. `UserUttered`: Este evento se ejecuta automáticamente cuando la usuaria envía un mensaje al chatbot e indica el texto enviado, así como otros datos (intent, entities, etc...). Por ejemplo este evento se ejecuta cuando la usuaria le dice "Hola" al chatbot

4. `BotUttered`: Este evento se ejecuta automáticamente cuando el chatbot envía un mensaje a la usuaria e indica el texto enviado. Por ejemplo este evento se ejecuta cuando el chatbot le contesta "Hola" a la usuaria.

5. `ActionExecuted`: Este evento se ejecuta cuando automáticamente se ejecuta cualquier acción que ha predicho las políticas de diálogo e indica el nombre de dicha acción.

6. `SlotSet`: Este evento se ejecuta automáticamente en el momento que se asigna el valor de un slot con el valor de una entity con el mismo nombre. De otra forma, se ejecuta manualmente cuando asignamos el valor de un slot desde una custom action.

En el Rasa Action Server se ejecutan principalmente las custom actions, las cuales se definen en el archivo `actions.py`. Este archivo está destinado específicamente a la implementación de custom actions que el programador quiera desarrollar o para

la personalización de default actions que Rasa tiene, como por ejemplo action_listen, si el programador quisiera podría *customizar* esa acción.

Cualquier custom action debe tener dos métodos, por un lado el método “name” que devuelve el nombre de la custom action, como vemos en el ejemplo de la Figura 5.7. Dicho nombre se deberá indicar en el `domain.yml`. Por otro lado es necesario definir el método “run”, el cual define la implementación de la custom action. En el ejemplo de la Figura 5.7 podemos observar la utilidad del tracker presentado en la `Sección` 4.2.1. Dicho tracker se utiliza para obtener el valor de una entity, en este caso *barrio*.Además, en la Figura 5.7 como asignamos el valor de un slot mediante el evento SlotSet. Otro componente importante es el “dispatcher” el cual nos permite, desde la custom action, mandar mensajes a la usuaria. Dichos mensajes pueden enviarse mediante responses ya creadas (y especificadas en el domain.yml) o mediante textos totalmente personalizados, como es el caso del ejemplo.

```
class ActionValidarNeighbor(Action):
  1
    def name(self) -> Text:
        return "action_validar_neighbor"

    def run(self, dispatcher: CollectingDispatcher,
    tracker: Tracker,
    domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
      2
        neighborhood = next(tracker.get_latest_entity_values("barrio"), None)
        if neighborhood is not None:
            if neighborhood.title() in ALLOWED_LOCATIONS:
              3 dispatcher.utter_message(f"Genial. Recordaré ese barrio ({neighborhood})")
                return [SlotSet("barrio", neighborhood)]

        dispatcher.utter_message("Puedes indicarme uno de estos barrios y recordaré que te
        dispatcher.utter_message("Ciutat Vella, Eixample, Sants, Sarria, Sant Gervasi, Les
        return []
```

Figura 5.7: Extracto de código donde se muestra la implementación de una custom action

5.3.5. Base de Datos

En este proyecto se utilizan dos bases de datos, inicialmente se pensó hacer todo utilizando únicamente la base de datos de Decidim, mediante la API mencionada en la `Sección` 2.4 , la cual permitía obtener información de los procesos participativos. Conforme avanzó el proyecto vimos que no era viable utilizar únicamente esa base datos debido a la falta de información que tenia, de modo que se pensó en implementar otra base de datos con datos *fake* para poder llevar un poco más lejos las funcionalidades del chatbot.

API de Decidim

La base de datos de Decidim, de la cual se pueden obtener datos mediante la API de Decidim, ha sido utilizada en este proyecto para obtener datos a tiempo real de los diferentes procesos participativos que existen en la plataforma, como por ejemplo el título del proceso, su fecha de creación o el slug con el cual se identifican los procesos. Dicha API sigue la especificación GraphQL. GraphQL es un lenguaje para hacer queries y manipular datos para APIs, y un entorno de ejecución para realizar consultas con datos existentes. Es un lenguaje sencillo y entendible para la usuaria.

Como vemos en la Figura 5.8, la API de Decidim devuelve una estructura de tipo JSON con los datos que han sido solicitados. GraphQL es una herramienta muy útil debido a que la propia herramienta te facilitará lo que puedes pedir o no mediante el autocompletado, lo único que tienes que hacer es seguir la estructura y él mismo te irá mostrando que puedes y que no puedes pedir.

```
1 {
2   participatoryProcesses(filter: {publishedSince: "2023-03-01"}) {
3     id
4     title {
5       translation(locale: "es")
6     }
7   }
8 }
```

Figura 5.8: Ejemplo de query en GraphQL

Además de todo esto existe una documentación por parte de Decidim en https://www.Decidim.barcelona/API/docs donde te permite entender el funcionamiento de la API, y entrar más en detalle en que consultas puedes hacer y que datos puedes recibir.

En la documentación se explica que la base de datos de Decidim se divide en dos clases principales, Participatory Spaces (espacios participativos) y Components (componentes). Dentro de los espacios participativos hay 5 subclases (Participatory Processes, Assemblies, Consultations, Conferences and Initiatives), de las cuales solo nos centraremos en los Procesos Participativos. Todo espacio participativo puede (y

```
{
  "data": {
    "participatoryProcesses": [
      {
        "id": "287",
        "title": {
          "translation": "Transformem els patis"
        }
      },
      {
        "id": "323",
        "title": {
          "translation": "Futuros usos del Taller Masriera"
        }
      },
      {
        "id": "324",
        "title": {
          "translation": "Reurbanización de los interiores de manzana de la Guineueta "
        }
      }
    ]
  }
}
```


Figura 5.9: Ejemplo de resultado de la query 5.8 en GraphQL

debe) tener algunos componentes. Hay 9 componentes oficiales, estos son: Propuestas, Página, Encuentros, Presupuestos, Encuestas, Contabilidad, Debates, Blogs y Insaculación (procedimiento de elegir alcaldes, regidores y otros oficiales de justicia y de gobierno, mediante sorteo). La idea era poder explotar todas estas componentes durante el transcurso del proyecto, pero como norma general los procesos participativos no suelen tener mucha variedad de componentes y en algunos casos los componentes no tienen muchos datos que explotar. Destacan principalmente los encuentros.

Para hacer consultas a la API de Decidim desde nuestro proyecto, como bien nos comentan en la documentación, se harán mediante el endpoint "https://www.Decidim.barcelona/api" y usando el comando POST. En este proyecto se harán las peticiones mediante un fichero python.

Como muestra la Figura 5.10 la estructura de las consultas se mantiene muy parecida respecto a como se haría desde la interfaz del navegador. Para hacer POSTS desde python importaremos la librería "requests" y haremos un post usando dicha librería. Se llamará al método "post" e indicaremos tanto la url como la query en formato JSON (ver recuadro rojo 1 de la Figura 5.10). Una vez ejecutado el método "post" recibiremos una respuesta. En la respuesta podremos observar tanto el código de estado, como la respuesta en sí. Finalmente, una vez recibimos el output lo cargamos como JSON mediante JSON.loads(String) (ver recuadro rojo 2 de la Figura 5.10).) y obtendremos el JSON asociado a la query.

En este proyecto se han implementado, en el archivo queries.py, las siguientes funciones con consultas a la API de Decidim:

1. `query_latest_ParticipatoryProceses`: query a la API de Decidim la cual

```
def query_Components_ParticipatoryProceses(slug):
    query = ("""query {
                participatoryProcess(slug: """ + '"' + f"{slug}" + '"' + """) {
                    id
                    components{
                    id
                    __typename
                    }
                }
            }""")

    print(query)

    url = 'https://www.decidim.barcelona/api/'
1   r = requests.post(url, json={'query': query}, verify=False)
    print(r.status_code)
    # print(r.text)
2   json_data = json.loads(r.text)["data"]["participatoryProcess"]["components"]
    print(json_data)

    return json_data
```

Figura 5.10: Ejemplo de query en python a la API de Decidim

nos devuelve datos (título, descripción, área, etc...) de los proceso participativo en formato JSON y filtramos para conseguir el publicado más recientemente.

2. `query_ParticipatoryProceses_location`: query a la API de Decidim la cual nos devuelve los procesos participativos y filtramos para conseguir el que corresponde a la localización indicada en los parámetros.

3. `query_Components_ParticipatoryProceses`: query a la API de Decidim la cual nos devuelve un proceso participativo a partir del slug (el slug es el id por el cual se identifican los procesos en la API de Decidim) indicado por parámetros y analizamos si el proceso tiene debates, encuentros, etc...

4. `query_ParticipatoryProces_by_slug`: query a la API de Decidim la cual nos devuelve datos (título, descripción, área, etc...) de un proceso participativo en formato JSON a partir del slug indicado por parámetros.

5. `query_last3_Proposals_by_slug`: query a la API de Decidim la cual nos devuelve las 3 propuestas más recientes de un proceso participativo, a partir de un slug indicado por parámetros.

6. `query_ParticipatoryProceses_interests`: query a la API de Decidim la cual nos devuelve los procesos participativos y filtramos para conseguir uno que cumpla alguno de los intereses pasados por parámetros.

Mongo DB

Además de la API de Decidim, durante el proyecto también se creyó oportuno tener otra base de datos con la cual poder tener datos *fake* y así mostrar funcionalidades adicionales del chatbot que no serían posibles con la API de Decidim (principalmente, porque esos datos no están accesibles por privacidad y otros motivos). Por ejemplo, creemos que un punto importante de la comunidad virtual de Decidim, son las subcomunidades que se forman en grupos o amistades. Para ello, se pensó en hacer una funcionalidad que te animará a participar en aquellos procesos en los cuales los miembros de tu comunidad participarán o siguieran, (una usuaria sigue un proceso debido a que está interesado en dicho proceso), pero por motivos de privacidad esa información de los usuarios no es accesible desde la API. Además de eso, nos pareció interesante tener la capacidad de poder hacer un resumen de los procesos participativos para facilitar las novedades de un proceso a una usuaria la cual hace tiempo que no visita la plataforma. En Decidim, también se puede filtrar según ciertas temáticas, de forma que queríamos tener una funcionalidad que, a partir de ciertos intereses o gustos de una usuaria habitual a la plataforma, se pudiera ofrecer procesos acorde a dichos intereses, no obstante estos datos no están disponibles desde la Api de Decidim. Por tanto, se creó una base de datos dividida en 2 *clusters*. Por un lado, un *cluster* guarda información de los usuarios (nombre, barrio, lista de amistades, lista de procesos que sigue y temáticas de interés), y por otro lado un *cluster* que guarda procesos participativos con sus encuentros y los comentarios de cada encuentro.

Investigando entre todas las posibles bases de datos aptas para python, debido a que no era necesario una base de datos relacional, escogí MongoDB. MongoDB es una base de datos no relacional, una base de datos orientada a documentos. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Dichos documentos son almacenados en BSON, que es una representación binaria de JSON.

Nunca había creado una base de datos en MongoDB, así que tuve que aprender. Para ello seguí el tutorial de la propia página oficial https://www.mongodb.com/languages/python. Donde paso a paso me explicaban como crear una base de datos. El primer paso, desde la página de MongoDB, sería registrarse y crear una organización, posteriormente crear un proyecto dentro de dicha organización y finalmente, dentro del proyecto crear un *cluster*. Una vez creado un *cluster* se debe implementar un método que nos permita acceder a él desde python.

En la Figura B.5 en el Anexo se muestra un método del fichero *pymongo_get_database.py* donde podemos ver la estructura con la cual obtenemos la instancia de la base de datos de MongoDB. En dicho fichero únicamente obtendremos la instancia de la base de datos dentro del *cluster*. Para obtener la URI correcta de cada *cluster* debemos obtenerla desde la página de MongoDB, razón por la cual es necesario tener el *cluster* creado antes de este paso.

Dentro de los *clusters* existen colecciones, en dichas colecciones guardaremos datos parecidos entre sí, por ejemplo, podemos tener una colección de usuarias. Para poder acceder a la colección desde python, como vemos en la Figura B.5,

lo primero será obtener la instancia de la base de datos, para ello crearemos un método que nos la devuelva. Una vez hemos obtenido dicha instancia se llamará a una colección de la base de datos (si no existe la creará). Posteriormente debemos insertar en esa colección los documentos que queramos en formato JSON. En la Figura B.6 en el Anexo, se muestra un extracto del fichero con el cual insertamos las usuarias, donde podemos observar lo comentado en este apartado.

Además de la colección de usuarias, también hay otra colección en la base de datos del proyecto, la colección de procesos participativos. En esta colección, entre otras cosas, he añadido una serie de comentarios hechos por usuarias reales en distintos encuentros, ya que no había forma de recuperarlos mediante la API de Decidim. De modo, que manualmente copié los comentarios y los introduje en otra colección para poder hacer la funcionalidad de resumir encuentros.

Para hacer queries a MongoDB y acceder a los datos de las diferentes colecciones es tan fácil como tener la instancia de la base de datos, obtener la colección deseada y llamar al método “find()”.De esta manera te devolverá todos los documentos que se encuentran en la colección. En la Figura 5.11 se muestra un ejemplo de como hacer una consulta. El método “find()” además, te permite filtrar dentro de la colección. Si le indicas una categoría y su valor te devolverá todos los documentos que satisface dicha filtración, por ejemplo: find({”name” : “Alex”}).

Como se ha comentado anteriormente en la `Sección` 4.2.5, hay dos acciones que requieren de consultas a la base de datos de MongoDB, `action_propuestas_amigos`, `action_offer_sumarization_encuentros` y `action_sumarization`. La acción `action_propuestas_amigos` hace una query a la colección de usuarias de la base de datos para buscar los amistades de la usuaria y otra para obtener los procesos que siguen dichas amistades, posteriormente hace una query a la colección de procesos participativos para obtener las propuestas de dicho proceso y ofrecerla a la usuaria. La acción
`action_offer_sumarization_encuentros` hace una query a la colección de procesos participativos para obtener los encuentros de dicho proceso y ofrecerselos a la usuaria. `action_sumarization` busca el encuentro que la usuaria ha seleccionado en la colección de procesos y obtiene su lista de comentarios con la que posteriormente generará un resumen y lo enviará a la usuaria.

```python
# Create a new collection
collection_name = dbname["users_list"]
item_details = collection_name.find()

for item in item_details:
    print(item)
```

Figura 5.11: Ejemplo para obtener datos de una colección en MongoDB

5.4. Resultados

En esta sección se explicará mediante figuras, las diferentes interacciones que la chatbot de Decidim puede conseguir mediante las conversaciones presentadas en el `Capítulo` 3. Esta sección está divida en 3 subsecciones para mostrar las conversaciones con usuarias registradas, usuarias no registradas y conversaciones tanto para usuarias registradas como no registradas.

5.4.1. Usuaria No Registrada

En este apartado se mostrarán las conversaciones entre una persona usuaria no registrada y la chatbot Sara. De forma que, en la Figura 5.12 podemos observar parte de la Conversación 1-SaludoDefDec, en la cual una vez la usuaria no registrada hace clic en el widget y lo abre, la chatbot de forma proactiva habla a la usuaria.

Posteriormente en la Figura 5.13 podemos ver como la usuaria le indica su nombre a la chatbot y esta le responde con una muestra de sus funcionalidades para animar a la usuaria a clicar en alguna de ellas. En la Figura 5.14 podemos observar la Conversación 2-SaludoProcP, en la cual, cuando la usuaria pregunta o hace clic en el botón "¿Quién puede participar en los procesos participativos?", o pregunta o hace clic en el botón "Cómo puedo participar en Decidim Barcelona" que aparece en la Figura 5.13, la chatbot le muestra una explicación de ello y además, como vemos en la Figura 5.15 también le pregunta si le interesa un proceso participativo de ejemplo. En caso de que la usuaria responda afirmativamente escribiendo o clicando en "Sí", como se muestra en la Figura 5.16 la chatbot le ofrecerá una serie de barrios y cuando clique o escriba alguno, buscará de acuerdo a dicha localización como vemos en la Figura 5.17. No obstante, en caso de no encontrar ningún proceso participativo en el barrio especificado por la usuaria, la chatbot Sara le informará de ello y le mostrará el último proceso añadido a Decidim independientemente del barrio asociado a dicho proceso.

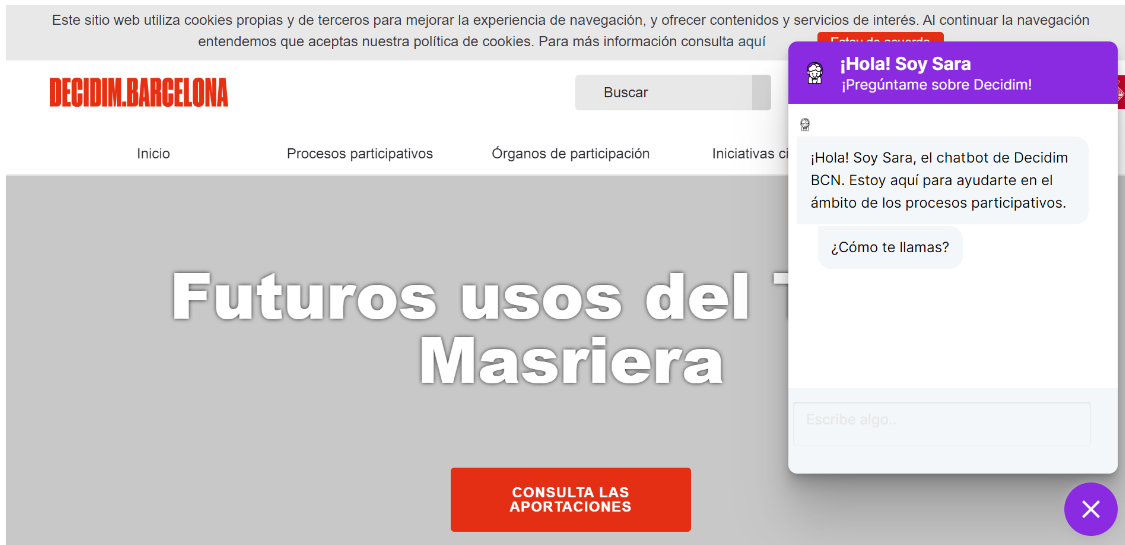

Figura 5.12: La chatbot inicia la conversación, se presenta y le pregunta su nombre

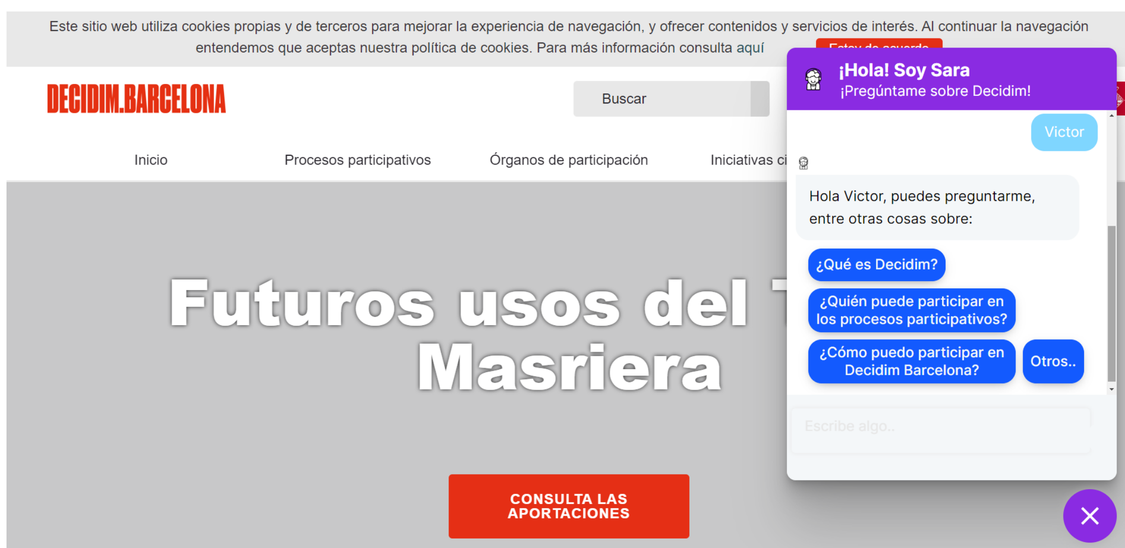

Figura 5.13: La chatbot le da opciones a la usuaria para que clique

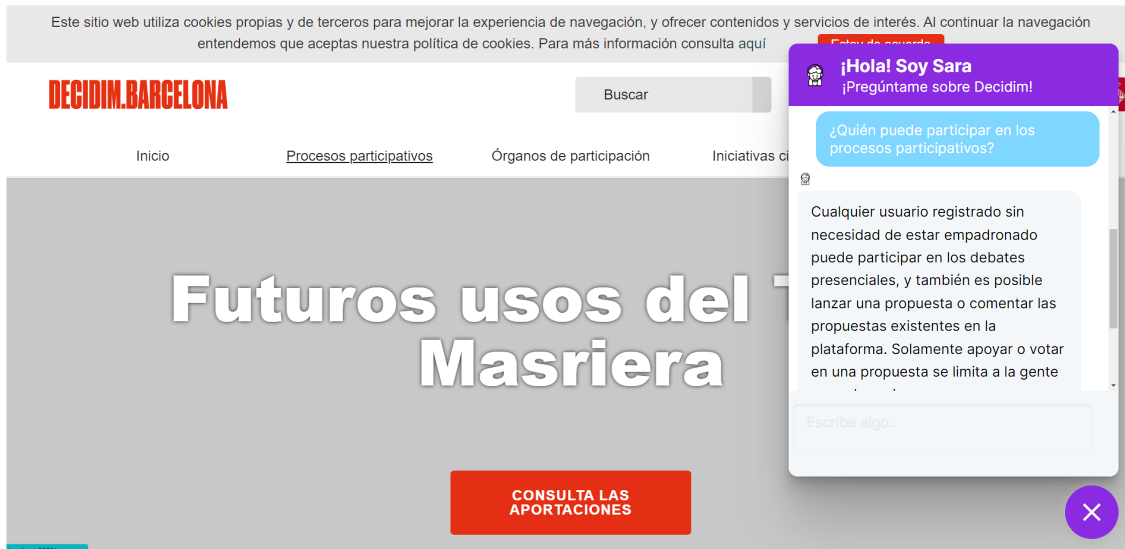

Figura 5.14: La chatbot le explica quien puede participar

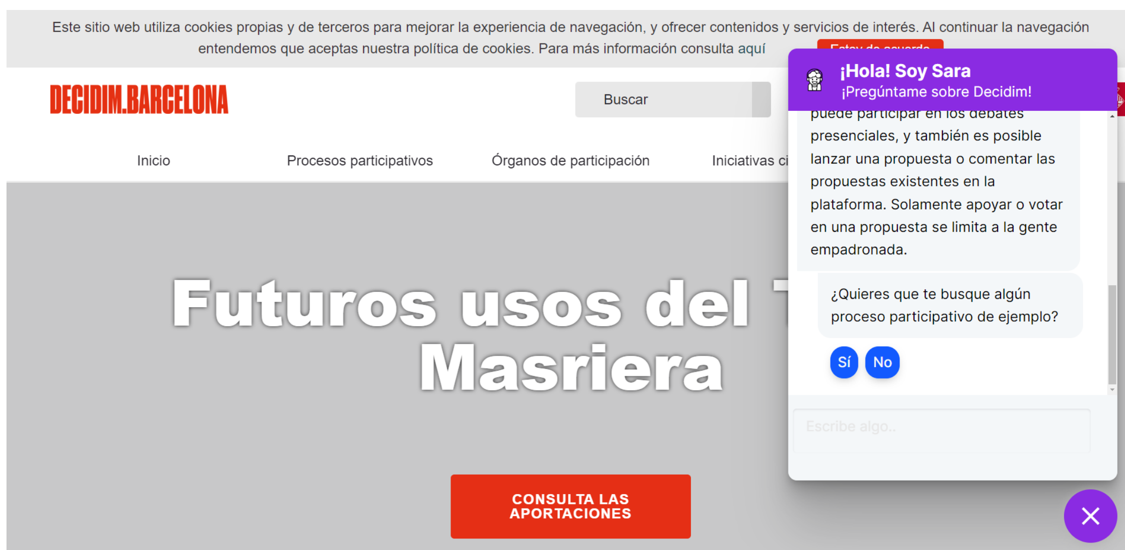

Figura 5.15: La chatbot le propone buscarle un proceso participativo

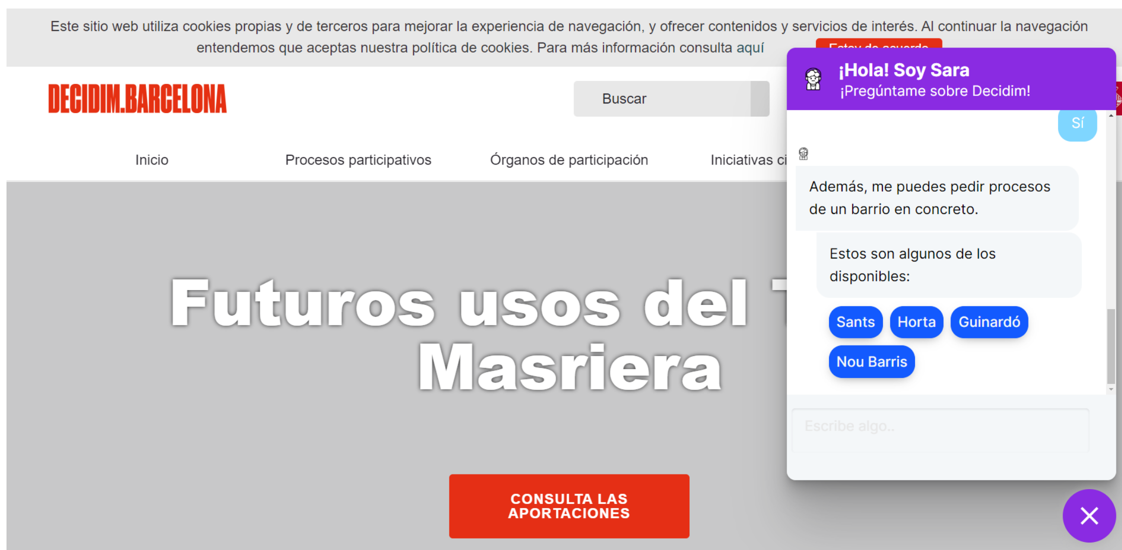

Figura 5.16: La chatbot le da a escoger algún barrio donde buscar

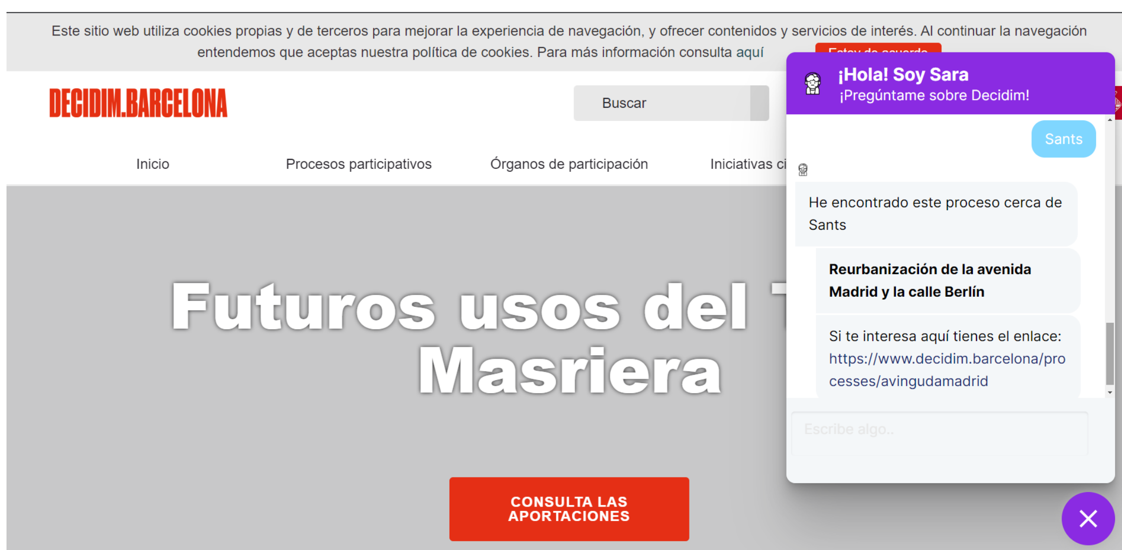

Figura 5.17: La chatbot le ofrece un proceso

De forma alternativa, si en la Figura 5.13 la usuaria hace clic en “Otros” como vemos en la Figura 5.18 la chatbot le mostrará otras funcionalidades. En la Figura 5.19, si la usuaria hace clic en “Conceptos de Decidim” le mostrará algunos conceptos que posiblemente no sepa su significado, como se explicaba en la Conversación 3-SaludoConcepto y clicando en alguno de ellos se mostrará su definición y los conceptos relacionados. Por otro lado, en la Figura 5.20 vemos que si la usuaria hace clic en “¿Cómo me registro?”, le muestra una explicación de como hacerlo, tal y como se explicaba en la Conversación 4-SaludoRegistro.

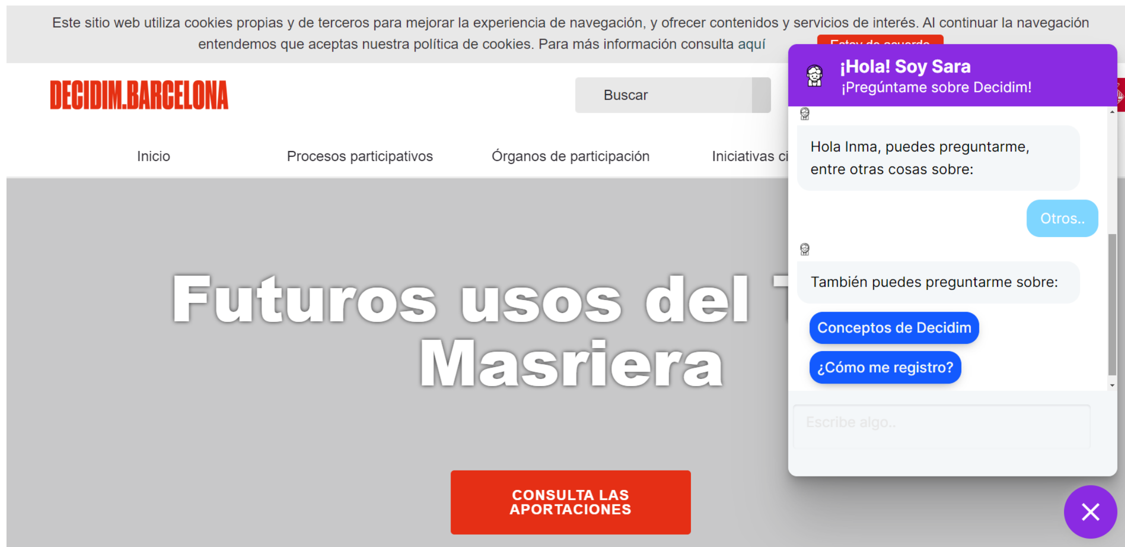

Figura 5.18: La chatbot le da otras opciones a la usuaria para que clique

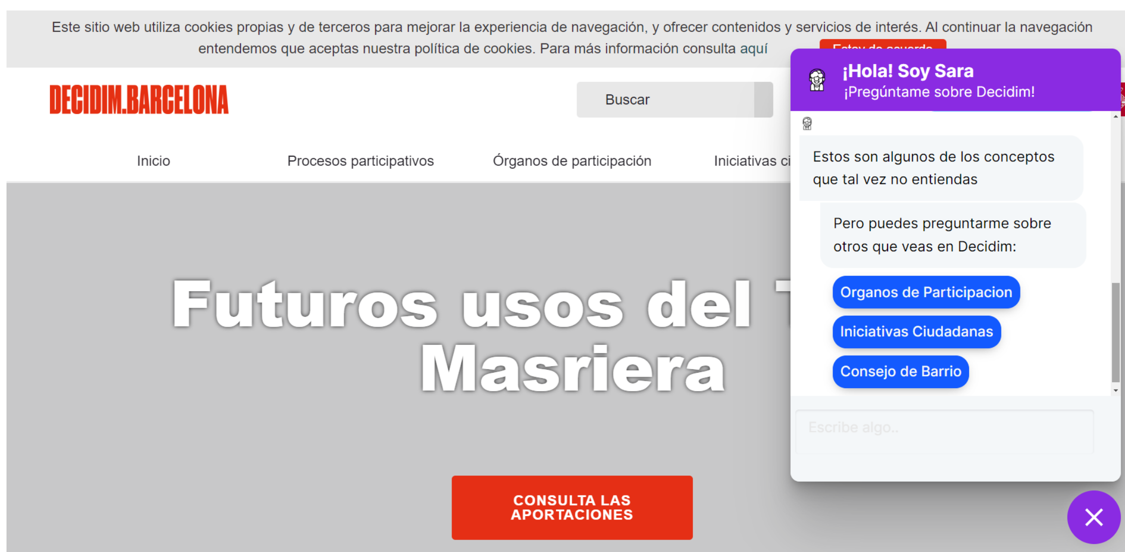

Figura 5.19: La chatbot le ofrece algunos conceptos que puede que la usuaria no entienda

Figura 5.20: La chatbot le explica cómo registrarse

En la Figura 5.21 podemos observar la Conversación 5-VisitarPágina en la cual la chatbot de forma proactiva le explica a la usuaria qué puede hacer dependiendo de la página en la que se encuentre. Esta interacción se da una vez por cada página únicamente, ya que se entiende que una vez que ha visitado la página y se le ha explicado lo que puede hacer en ella, no es necesario volver a repetirlo en caso de que vuelva a visitar dicha página. De igual forma en las Figuras 5.22, 5.23, 5.24 también se muestran los demás escenarios (Página de un proceso concreto, Página de encuentros de un proceso concreto, Página de propuestas de un proceso concreto).

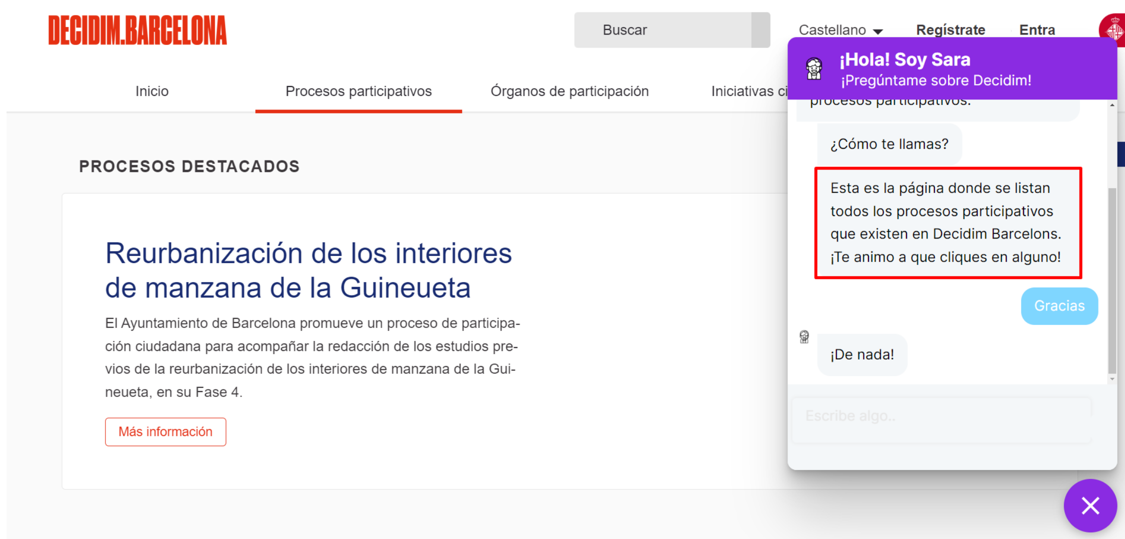

Figura 5.21: La chatbot le muestra una descripción de la página de procesos

Figura 5.22: La chatbot le muestra una descripción de la página de un proceso

Figura 5.23: La chatbot le muestra una descripción de la página de encuentros de un proceso

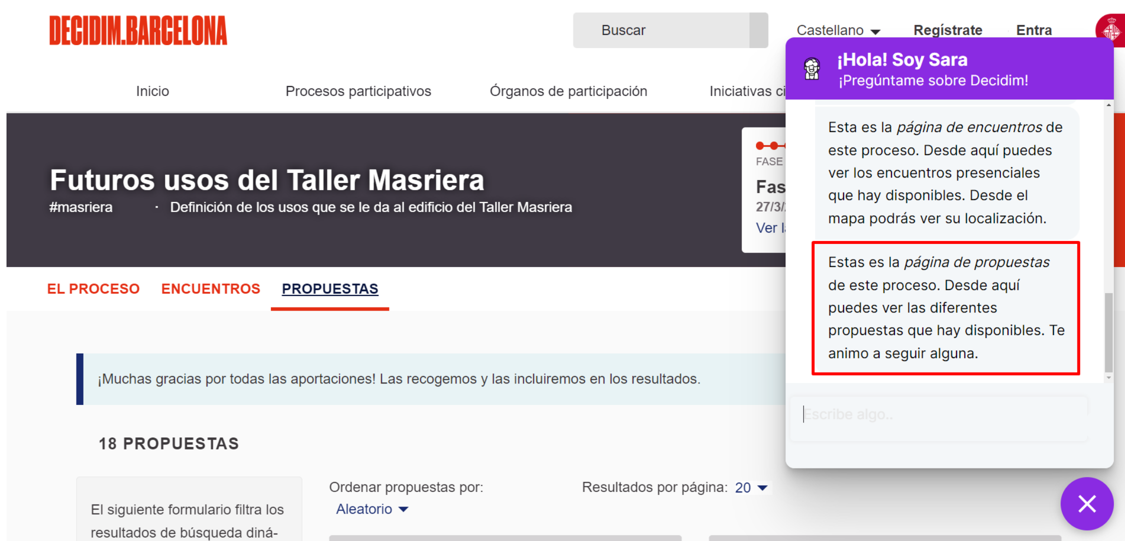

Figura 5.24: La chatbot le muestra una descripción de la página de propuestas de un proceso

5.4.2. Usuaria Registrada

En este apartado se mostrarán las conversaciones entre una persona usuaria registrada y la chatbot Sara. De forma que, en la Figura 5.25 podemos observar la Conversación 10-SaludoNombre en la cual la chatbot de forma proactiva saluda la usuaria con su nombre cuando ella hace clic en el widget.

En la Figura 5.26 podemos observar la Conversación 11-ProcInterés, en la cual si la usuaria visita por primera vez la página de procesos participativos la chatbot le preguntará si quiere que le busque un proceso participativo en base a sus gustos (dichos gustos son obtenidos de la base de datos *fake* creada para simular datos de posibles usuarios reales). En caso de que la usuaria responda afirmativamente escribiendo o clicando en "Sí", como vemos en la Figura 5.27, la chatbot le ofrecerá el título y *link* de algún proceso participativo acorde a sus gustos.

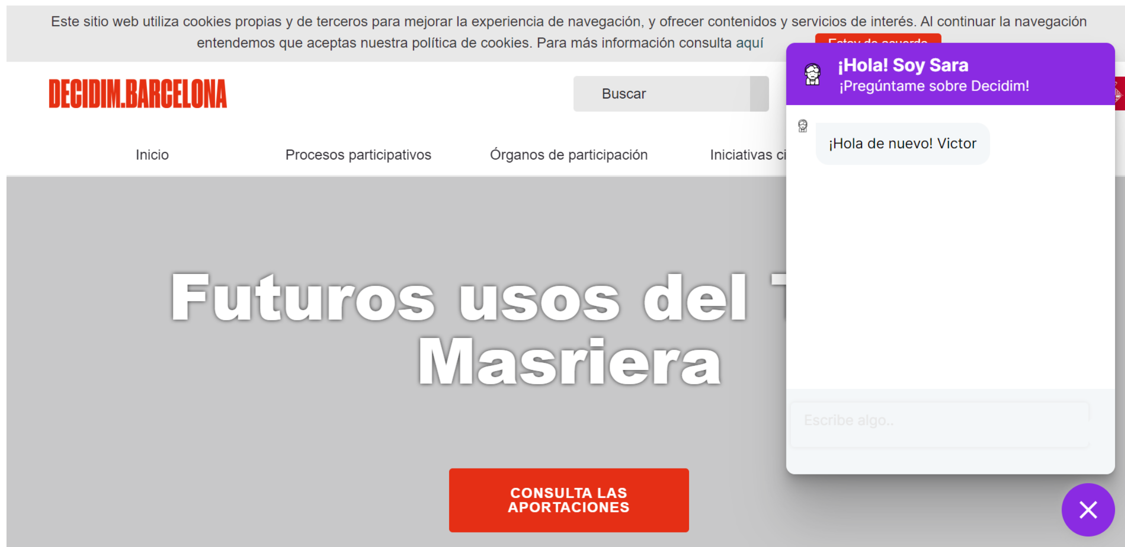

Figura 5.25: Inicio de conversación con usuaria registrada

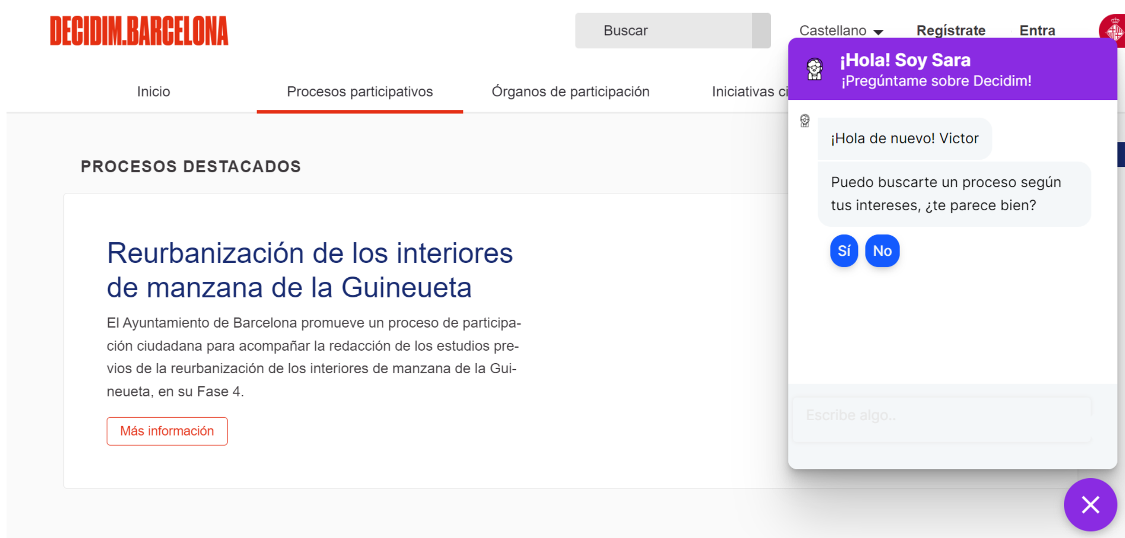

Figura 5.26: La chatbot le propone ofrecerle un proceso según sus intereses

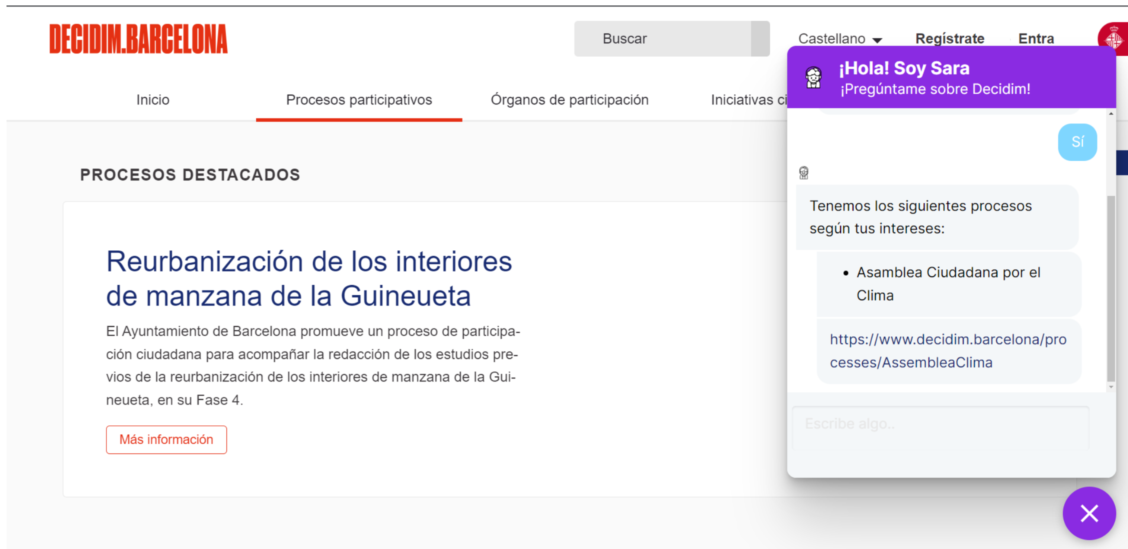

Figura 5.27: La chatbot le ofrece un proceso según sus intereses

En la Figura 5.28 podemos observar la Conversación 12-ResumenEnc, en la cual si la usuaria visita por primera vez la página de encuentros de un proceso, la chatbot le ofrecerá los encuentros de los que puede generarle un resumen. Del mismo modo, en caso de que la usuaria escriba “Me puedes hacer un resumen?”, como vemos en la Figura 5.29, la chatbot le ofrecerá los encuentros disponibles para ello. En caso de que la usuaria escoja alguno de ellos, la chatbot le generará el resumen, como vemos en la Figura 5.30 y 5.31.

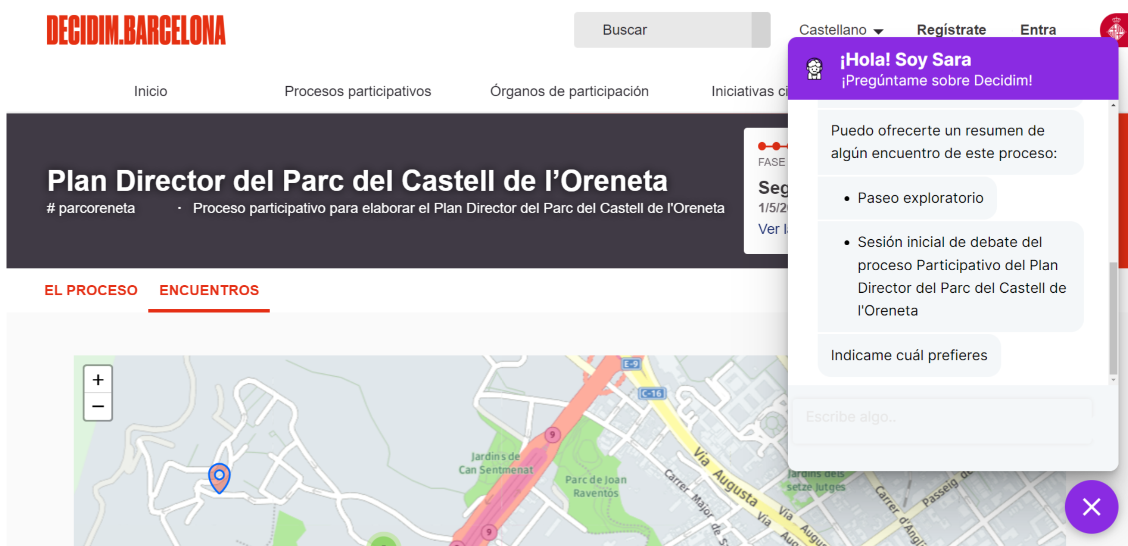

Figura 5.28: La chatbot le propone si le interesa un resumen de algún encuentro

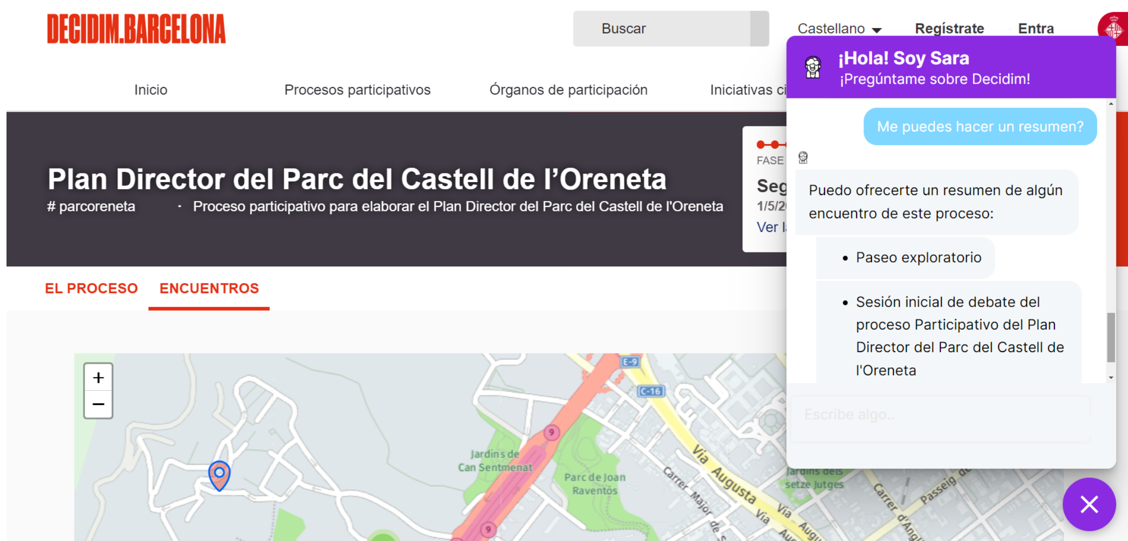

Figura 5.29: La usuaria pide un resumen de encuentros y la chatbot le ofrece los disponibles

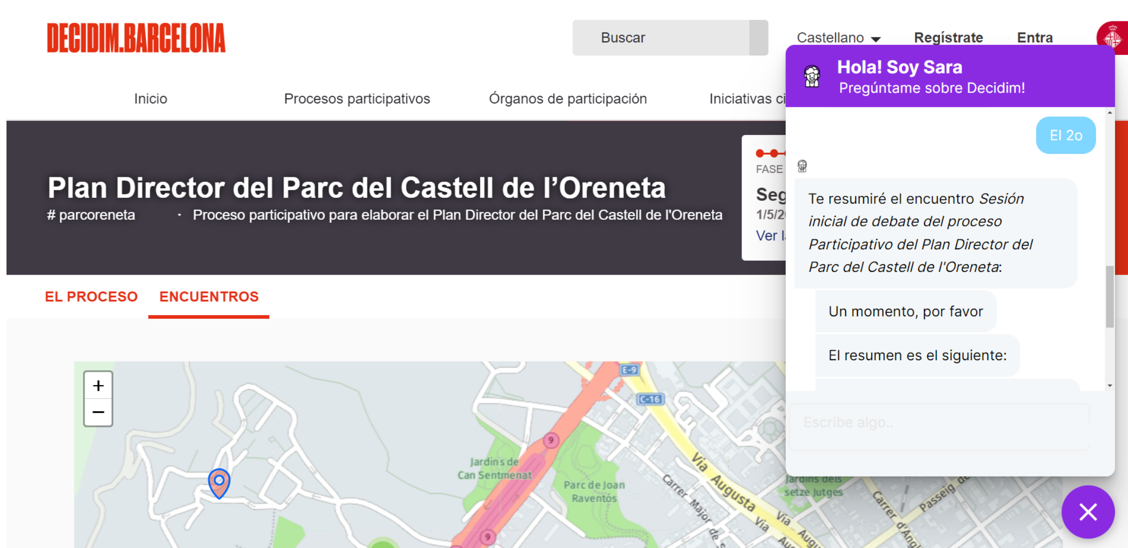

Figura 5.30: La chatbot le ofrece un resumen del encuentro seleccionado

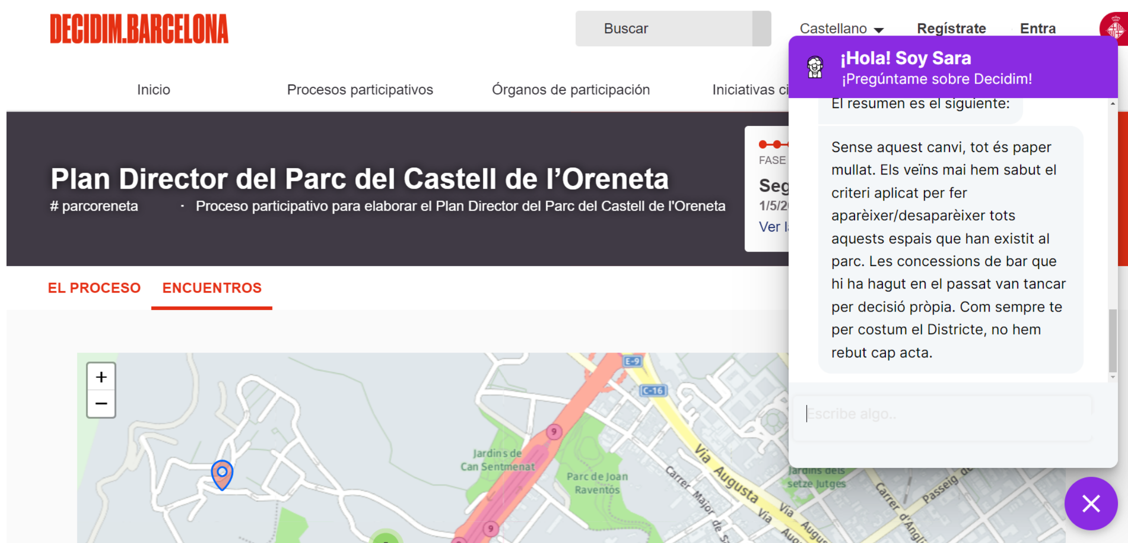

Figura 5.31: Resumen de los comentarios del encuentro

En la Figura 5.32 podemos observar parte de la Conversación 13-OrdenarProp o de la Conversación 14-PropAmistades, en la cual si la usuaria visita por primera vez la página de propuestas la chatbot le mostrará dos funcionalidades disponibles. En caso de que haga clic en "Propuestas de amistades" le proporcionará propuestas que siguen sus amistades, como vemos en la Figura 5.33 (recordemos que en la base de datos de usuarias guardamos entre otras cosas una lista con las amistades de cada usuaria). Por otro lado, si hace clic en "Últimas propuestas" la chatbot le proporcionará las 3 últimas propuestas de dicho proceso, como vemos en la Figura 5.34.

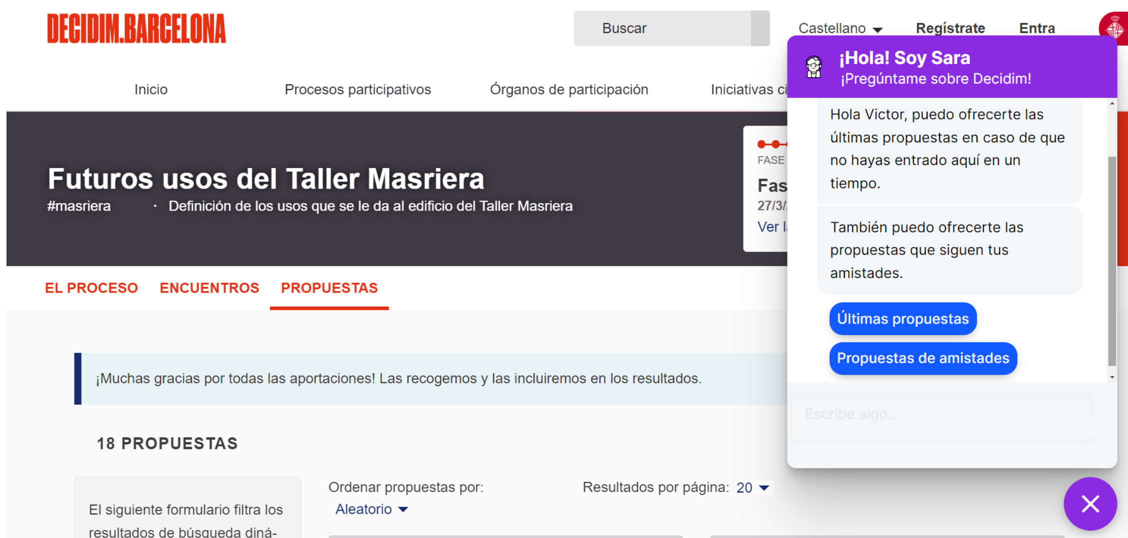

Figura 5.32: La chatbot ofrece 2 funcionalidades en la página de propuestas

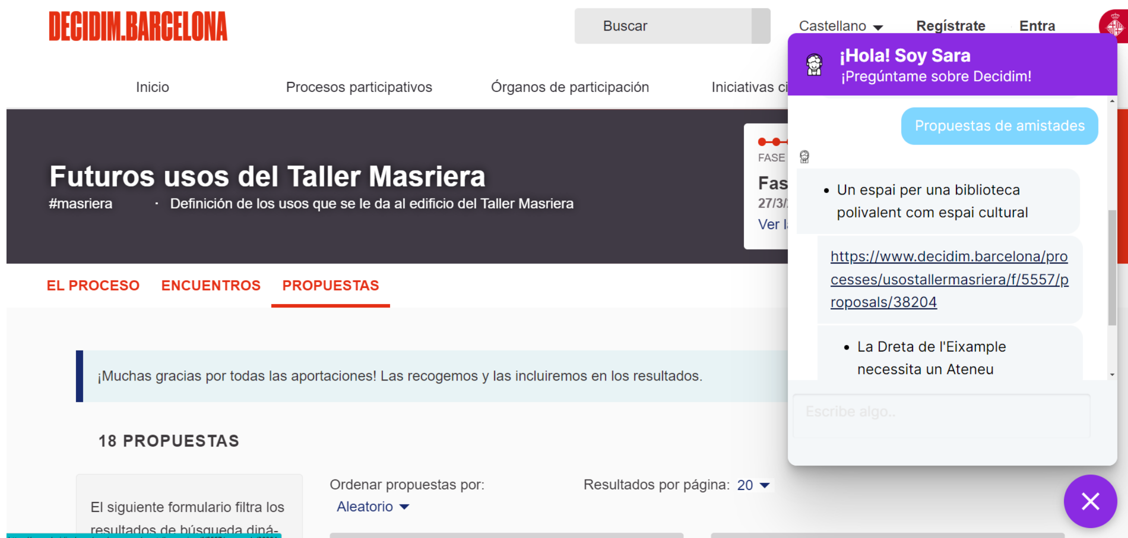

Figura 5.33: La chatbot le ofrece propuestas a partir de sus amistades

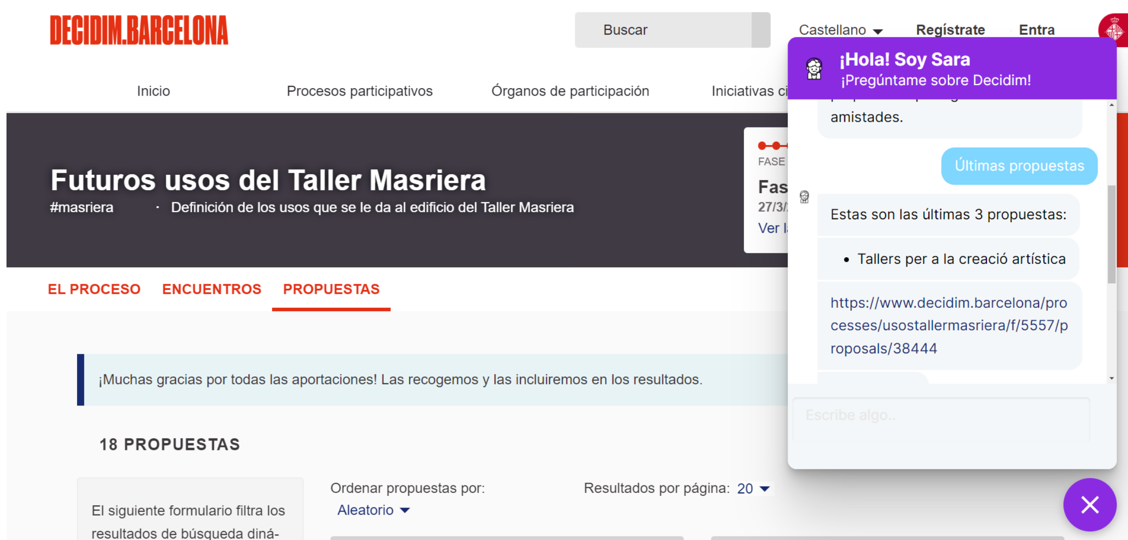

Figura 5.34: La chatbot le ofrece las últimas 3 propuestas

5.4.3. Usuaria Registrada y No Registrada

En este apartado se mostrarán las conversaciones entre una usuaria (independientemente de si está registrada o no) y la chatbot Sara. De forma que en la Figura 5.35 podemos observar la Interacción 4-Contexto en la cual si la usuaria pregunta "Donde estoy" la chatbot le muestra una descripción de la página.

En la Figura 5.36 podemos observar la Interacción 5-Presentación. En la cual si la usuaria pregunta "Que eres?" la chatbot se presenta.

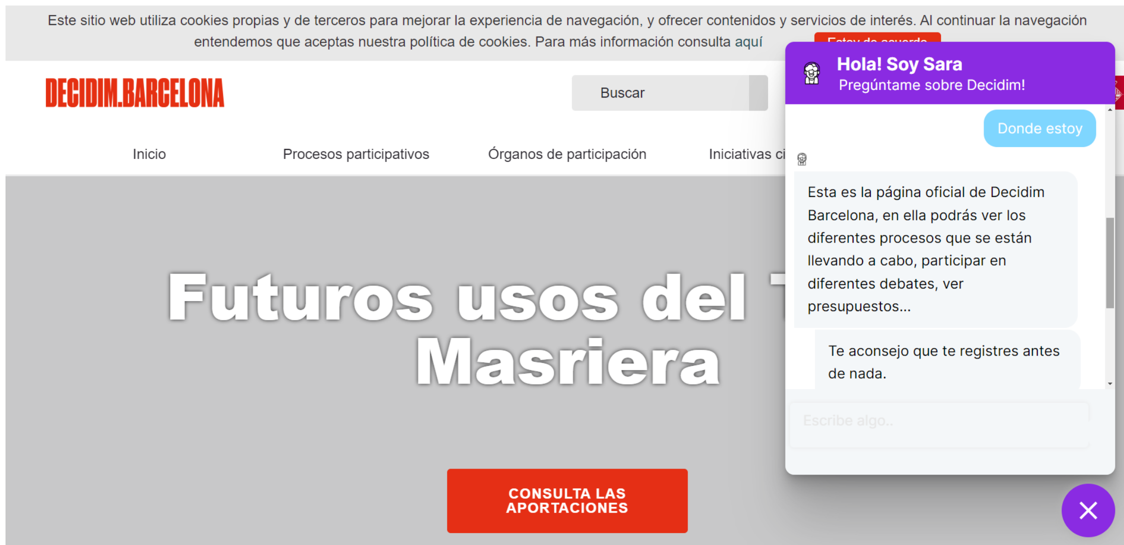

Figura 5.35: La chatbot le proporciona una explicación según la página donde esté (en este caso la página principal de Decidim)

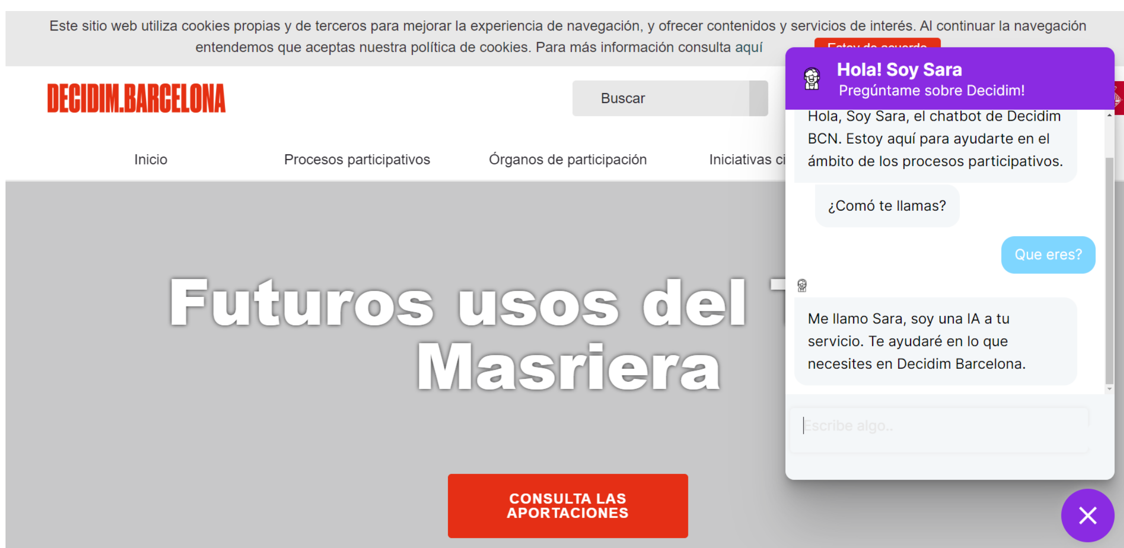

Figura 5.36: La chatbot se presenta

En la Figura 5.37 podemos observar la Conversación 7-PedirProcBarrio en la cual si la usuaria escribe “Me puedes dar un proceso participativo en Nou Barris?” de forma que pide un proceso, la chatbot le proporciona el título y el *link*. En dicho ejemplo la usuaria le indica el barrio en el cual quiere un proceso, en caso de que la usuaria no indicase barrio (Conversación 6-PedirProceso) o no hubiera ningún proceso en dicha localización, la chatbot le proporcionará el último proceso participativo.

En la Figura 5.38 podemos observar la Conversación 8-Def en la cual si la usuaria pregunta por algún concepto, la chatbot le muestra su definición y los conceptos relacionados.

En la Figura 5.39 podemos observar la Conversación 9-ComponenteProcP en la cual si la usuaria le pregunta por componentes de un proceso la chatbot le indicará si tiene o no. En caso de tener le proporcionará un *link* para llegar a dicha componente. Como vemos en la figura en este caso la usuaria inicialmente pregunta si el proceso tiene propuestas y posteriormente le pregunta si el proceso dispone de encuentros, en este último caso debido a que el proceso sí dispone de ellos, le ofrece un *link* con el cual podrá llegar a la página de encuentros del proceso del cual están hablando.

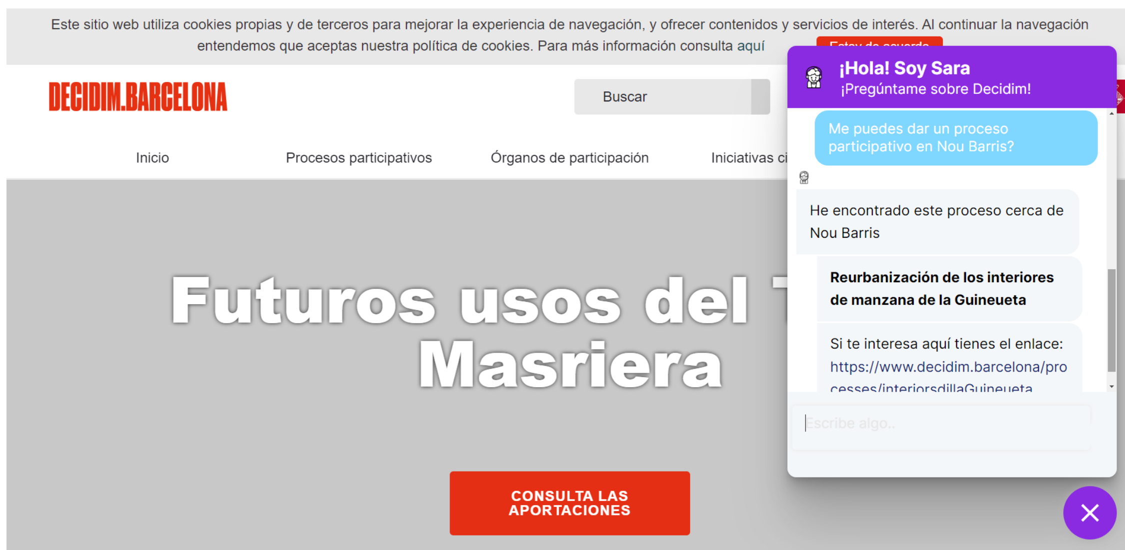

Figura 5.37: La chatbot le proporciona el título y *link* de un proceso

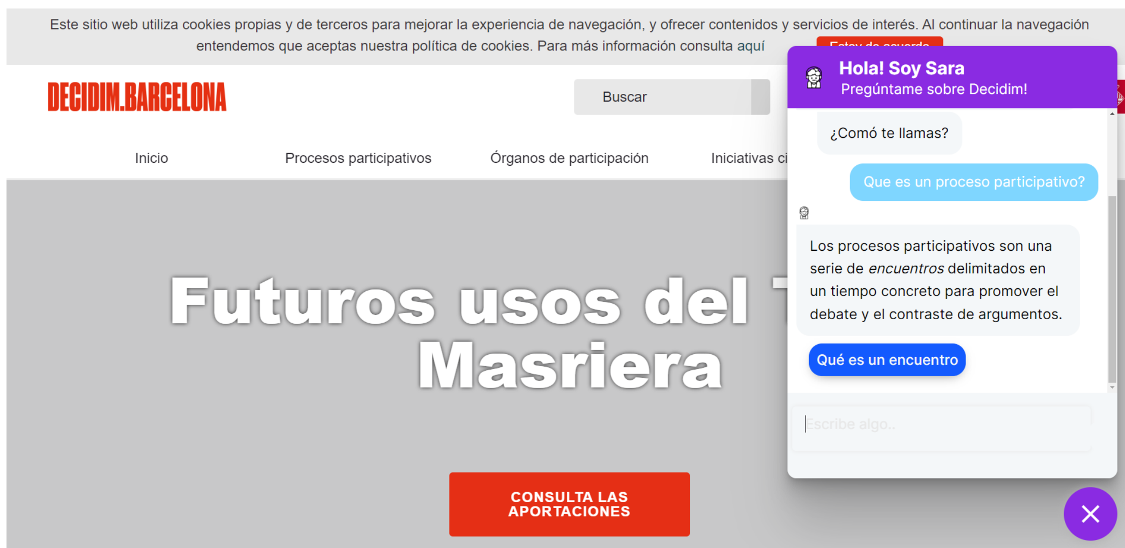

Figura 5.38: La chatbot le proporciona la definición del concepto

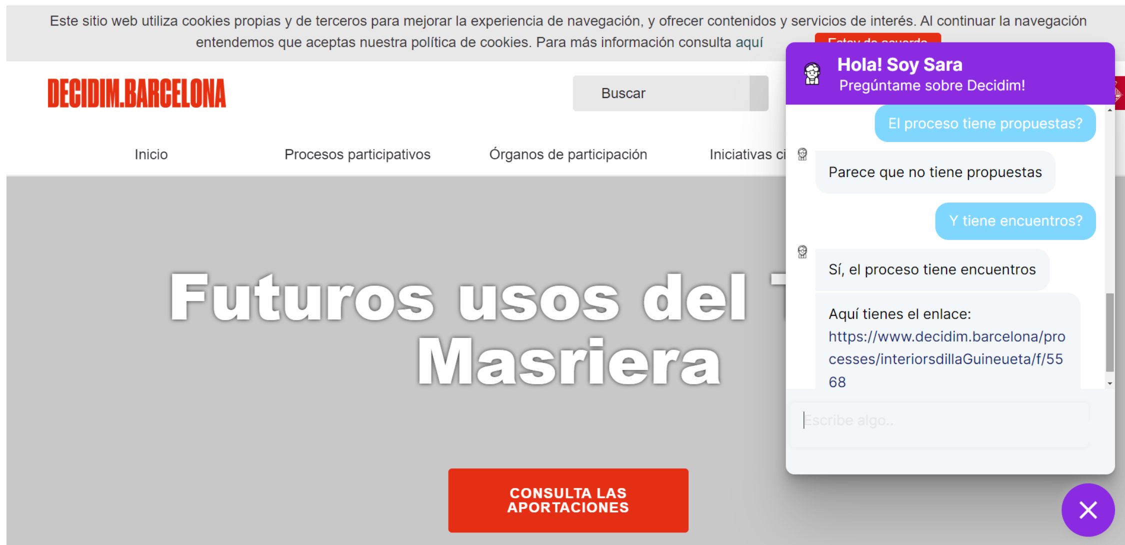

Figura 5.39: La chatbot muestra si el proceso tiene una componente o no

5.5. Chatbot Decidim vs ChatGPT

No se puede finalizar este proyecto sin antes mencionar el actualmente conocido ChatGPT, un prototipo de chatbot de inteligencia artificial desarrollado en 2022 por OpenAI, el cual a fecha de hoy se basa en el modelo GPT-4 de OpenAI [17]. ChatGPT es actualmente muy utilizado y es el precursor de la nueva era de los chatbots. Dicho esto, ChatGPT no puede ofrecer cualquier funcionalidad e incluso los resultados que ofrece ahora no son del todo fiables. Por ello, se pretende comparar dicho chatbot con el creado en este proyecto. No pretendo comparar este chatbot a grandes rasgos con ChatGPT, ya que sería algo imposible. Me centraré en funcionalidades y aspectos más concretamente, de la plataforma digital Decidim.barcelona.

En la Figura 5.40 le pregunto al ChatGPT “Que es Decidim Barcelona?”. El chatbot me responde, como habitualmente hace, con una muy elaborada respuesta sobre qúe es Decidim. Como podemos ver en la figura, la generación de lenguaje de forma dinámica es realmente destacable, ya que en apenas 10 segundos que ha tardado en redactar la respuesta, ha ofrecido una gran cantidad de información de un tema tan concreto como es la plataforma Decidim.

Posteriormente, en la Figura 5.41 le indico al ChatGPT “Soy de Horta”, para comprobar si entenderá el contexto de la conversación y me relacionará el mensaje con Decidim.barcelona. Para mi sorpresa, sí que lo enlaza y de nuevo, responde con una muy elaborada respuesta, donde me indica que tengo la posibilidad de participar en Decidim como residente de Horta e incluso me anima a participar.

Finalmente en la Figura 5.42 se muestra la última parte de la conversación con ChatGPT sobre Decidim Barcelona, donde se preguntará por un proceso participativo. En este punto se ha podido ver cómo dicho chatbot es increíblemente potente a la hora de generar lenguaje coherente. No obstante, también es importante ver hasta donde pueden llegar sus funcionalidades. Para ello le pregunté al chatGTP si

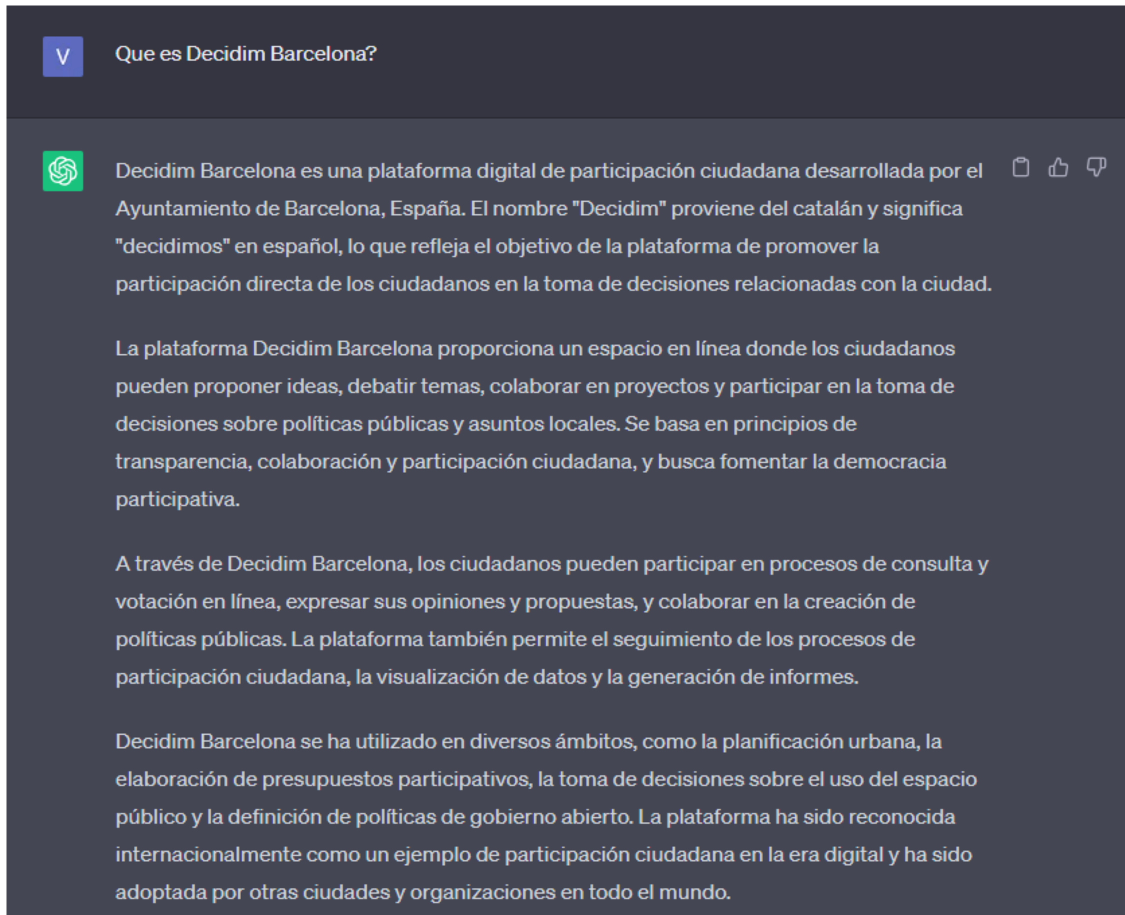

Figura 5.40: Pregunta al chatbot: "Qué es Decidim Barcelona"

me podía ofrecer un proceso participativo de mi barrio. Como vemos en la Figura 5.42 no es una sorpresa que en este punto recuerde que mi barrio es Horta, no obstante como vemos en la figura, no es capaz de conectarse a la API de Decidim y proporcionarnos un proceso participativo. Ya que su mensaje de respuesta es "No tengo acceso a información específica a tiempo real sobre procesos participativos en tu barrio, ya que mi conocimiento se basa en datos recopilados hasta septiembre de 2021". Personalmente, no creo que en un futuro ChatGPT tenga la capacidad de conectarse a la Api de Decidim y devolver un proceso participativo. En un futuro no muy lejano, sin duda, será capaz de conectarse a diferentes bases de datos, pero solamente a unas pocas escogidas selectamente, ya que no sería viable que pudiera conectarse a cualquier Api o base de datos existente, debido a la implementación que requeriría y al poco uso que tendría.

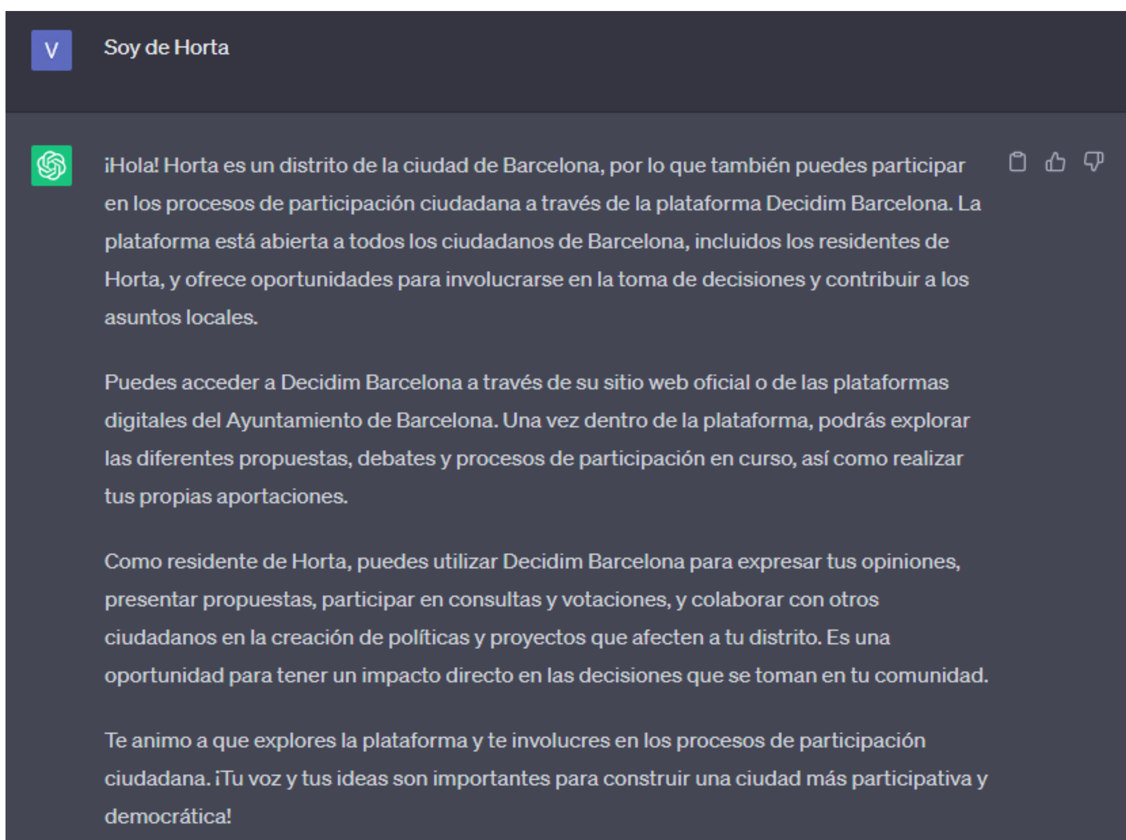

Figura 5.41: Pregunta al chatbot: “Soy de Horta”

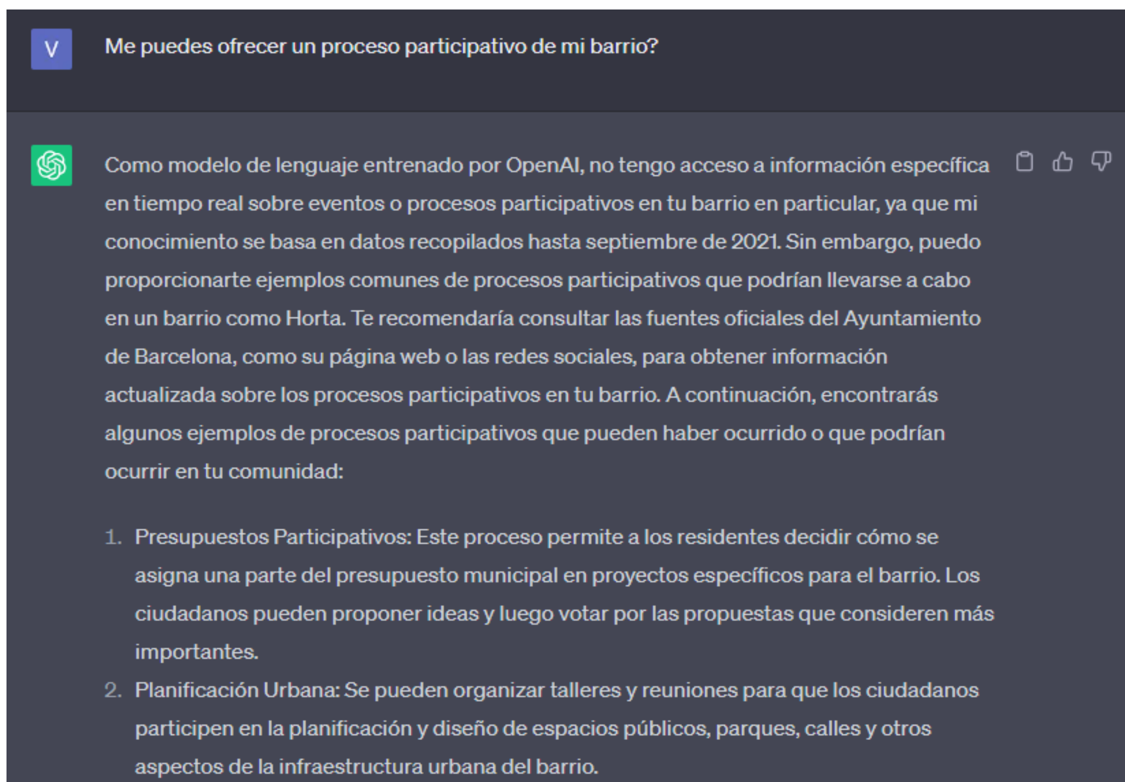

Figura 5.42: Pregunta al chatbot: “Me puedes ofrecer un proceso participativo de mi barrio?”

Capítulo 6

Conclusiones y Trabajo futuro

En este trabajo de fin de grado se ha conseguido aprender sobre el framework conversacional Rasa, se ha conseguido diseñar un agente conversacional adaptado a las necesidades de la plataforma digital Decidim Barcelona para fomentar la participación y el crecimiento de su comunidad y además, se ha conseguido integrar dicho chatbot en una web que emula la plataforma digital. Por tanto, a modo de cierre de este proyecto, podemos confirmar que se han conseguido los diferentes objetivos planteados al inicio de este proyecto. Destacando, tanto el aprendizaje del framework Rasa y de la plataforma de participación ciudadana Decidim Barcelona, como el desarrollo e integración de un chatbot destinado a dicha plataforma digital.

Pese a la limitación de tiempo en el desarrollo y a las diferentes dificultades durante el mismo y debido a la falta de experiencia en Rasa, personalmente estoy satisfecho del resultado obtenido. No obstante soy consciente de las limitaciones de este proyecto, como por ejemplo la falta de intents para poder captar cualquier posible mensaje de la usuaria, o también la falta de más historias con diferentes flujos para que el chatbot pueda reaccionar a cualquier situación. Dichas limitaciones destacan más aún comparándolo con ChatGPT, como se ha presentado en la `Sección` 5.5.

Para finalizar, me gustaría comentar distintos aspectos que se podrían implementar en un futuro, en una posible continuación de este proyecto. En este proyecto el texto que recibe la usuaria son frases predeterminadas escritas por el desarrollador. De forma que un gran avance podría ser la implementación de la capacidad del chatbot para generar lenguaje natural (NLG), mediante llamadas a la Api de ChatGPT para obtener la respuesta que él mismo genera o como avance aún más lejano, implementar la capacidad de generar, dinámicamente, lenguaje natural de forma propia. Otro aspecto importante para el futuro podría ser la incorporación de más funcionalidades para mejorar la participación en la plataforma digital. Además de una mejora en la base de datos, con más usuarias, más procesos participativos, más propuestas, encuentros y con más comentarios en cada uno de ellos.

Apéndice A

Manual Técnico

En este apartado del Apéndice, se pretende, de forma breve, explicar los pasos que tiene que seguir un desarrollador para que a partir del código fuente del proyecto pueda tener el sistema completamente instalado en su máquina, sepa cómo ejecutar dicho código, y así poder extender el proyecto. Los pasos son los siguientes:

1. Requisitos Previos
   - Descarga e instalar Anaconda (se recomienda versión 3.8 de python).
   - Crear un enviroment (conda create -n nombre_del_enviroment python=3.8).
   - Activar dicho enviroment (conda activate nombre_del_enviroment).
   - Instalar Rasa (pip install rasa).
2. Ejecutar el chatbot
   - Activar el enviroment en una terminal y moverse a la ubicación del proyecto (conda activate nombre_del_enviroment).
   - Entrenar el modelo (rasa train).
   - Iniciar el servidor de Rasa permitiéndole recibir mensajes del servidor web (rasa run –enable-api –cors "*").
   - Activar el enviroment en una segunda terminal y moverse a la ubicación del proyecto (conda activate nombre_del_enviroment).
   - Iniciar el servidor de acciones de Rasa (rasa run actions).
   - Activar el enviroment en una tercera terminal y moverse a la ubicación del proyecto (conda activate nombre_del_enviroment).
   - Iniciar el servidor web (python app.py).
   - Abrir el navegador, dirigirse a la url "localhost:5000" y clicar en el widget para iniciar la conversación.

Con todo esto, ya es posible para cualquiera con acceso al código fuente, continuar con este proyecto. Cabe destacar que cada vez que se hagan cambios en el

archivo `nlu.yml`, `domain.yml`, `stories.yml`, `rules.yml` o `config.yml` es necesario reiniciar el servidor de rasa. De forma equivalente, en el caso de hacer cambios en el archivo `actions.py` es necesario reiniciar el servidor de acciones de rasa.

Apéndice B

Apéndice con detalles de implementación

En este apéndice se ha incluido todo el material complementario que no se ha expuesto junto al resto de la memória debido al tamaño de la figura o a la cantidad de código que se observa en la misma.

En la Figura B.1 se observa el código implementado en el archivo `app.py` el cual nos crea la aplicación Flask y define los endpoints de la web.

```
# creates a Flask application with name "app"
app = Flask(__name__)

@app.route("/")
def home_page():
    # return render_template('index.html')
    return render_template('decidim.barcelona.html')

@app.route("/procesos")
def procesos():
    return render_template('Procesos participativos - decidim.barcelona.html')

@app.route("/procesoGuineueta")
def procesoGuineueta():
    return render_template('ProcesoGuineueta-decidim.barcelona.html')
```

Figura B.1: Extracto de código de app.py

En la Figura B.2 se observa el código base con el cual podemos implementar el widget de Rasa.

En la Figura B.3 se observa el código implementado en este proyecto para el widget de Rasa en el archivo `decidim.barcelona.html` y con el cual se inicia la conversación.

En la Figura B.3 se observa la función implementada en este proyecto con la cual se envían intents al servidor Rasa de forma externa. Gracias a ello podemos

```
!(function () {
  let e = document.createElement("script"),
    t = document.head || document.getElementsByTagName("head")[0];
  (e.src =
    "https://cdn.jsdelivr.net/npm/rasa-webchat@1.x.x/lib/index.js"),
    // Replace 1.x.x with the version that you want
    (e.async = !0),
    (e.onload = () => {
      window.WebChat.default(
        {
          customData: { language: "en" },
          socketUrl: "https://bf-botfront.development.agents.botfront.cloud",
          // add other props here
        },
        null
      );
    }),
    t.insertBefore(e, t.firstChild);
})();
```

Figura B.2: Código base del widget de Rasa

controlar la página en la que se encuentra la usuaria.

En la Figura B.5 se observa la función implementada en este proyecto con la cual se obtiene la instancia del *cluster* de MongoDB.

En la Figura B.6 se observa un ejemplo de la estructura para hacer inserts.

```
localStorage.clear();
!(function () {
  let e = document.createElement("script"),
    t = document.head || document.getElementsByTagName("head")[0];
  (e.src =
    "https://cdn.jsdelivr.net/npm/rasa-webchat@1.x.x/lib/index.js"),
    // Replace 1.x.x with the version that you want
    (e.async = !0),
    (e.onload = () => {
      window.WebChat.default(
        {
          customData: { language: "es" },
          socketUrl: "http://localhost:5005",
          initPayload: '/iniciar_conv{"usuario_registrado": "REGISTRADO","user_name": "Victor"}',
          // initPayload: '/iniciar_conv{"usuario_registrado": "NO_REGISTRADO"}',
          title: 'Hola! Soy Sara',
          subtitle: 'Pregúntame sobre Decidim!',
          profileAvatar: '/static/female-call-center-operator.gif',
          inputTextFieldHint: 'Escribe algo..',
          params: {storage: "local"}
        },
        null
      );
    }),
    t.insertBefore(e, t.firstChild);
})();
```

Figura B.3: Screenshot del widget de la página principal

```
function sendContext(){
  let existing = localStorage.getItem("chat_session");
  existing = existing ? JSON.parse(existing) : {};
  let local_session = existing["session_id"]

  var url = 'http://localhost:5005/conversations/' + local_session + '/trigger_intent?output_channel=latest';
    var data = {"name": "cambio_pagina", "entities": {"contexto": "PROCESO ENCUENTROS", "slug_actual": "avingudamadrid"}};
      fetch(url, {
        body: JSON.stringify(data),
        headers: {
          'dataType': 'json',
          'content-type': 'application/json'
        },
        method: 'POST',
        redirect: 'follow'
      })
      .then(response => {
        if (response.status === 200) {
          console.log(response.text());
      } else {
      throw new Error('Something went wrong on api server!');
      }
    })
    .catch(error => {
      console.error(error);
    });
}
```

Figura B.4: Función que envía un intent de forma externa a Rasa

```python
def get_database():

    uri = "mongodb+srv://vicllin7:smU7k7KwMMkdfmJl@clustertfg.kaowwhz.mongodb.net/"
    # Create a new client and connect to the server
    client = MongoClient(uri, tlsCAFile=certifi.where())
    # Send a ping to confirm a successful connection
    try:
        client.admin.command('ping')
        print("Pinged your deployment. You successfully connected to MongoDB!")
    except Exception as e:
        print(e)

    # Create the database for our example (we will use the same database throughout the tutorial
    return client['tfg_DataBase']
```

Figura B.5: Ejemplo para obtener la instancia de una bdd en MongoDB

```python
dbname = get_database()
collection_name = dbname["users_list"]

user_1 = {
    "_id" : "1",
    "user_name" : "Victor",
    "neighbor" : "Nou Barris"
    }
user_2 = {
    "_id" : "2",
    "user_name" : "Inma",
    "neighbor" : "Horta"
    }
user_3 = {
    "_id" : "3",
    "user_name" : "Maite",
    "neighbor" : "Sants"
    }

collection_name.insert_many([user_1,user_2,user_3])
# collection_name.insert_one(item_3)
```

Figura B.6: Ejemplo para insertar datos en una colección en MongoDB

Bibliografía

[1] *Una plataforma digital para fomentar la participación ciudadana.* URL: https://www.cellnex.com/es-es/trends/plataforma-digital-fomentar-participacion-ciudadana. (accedida: 02-06-2023).

[2] *¿Qué es un bot inteligente?* URL: https://www.freshworks.com/es/live-chat-software/chatbots/#:~:text=Los%5C%20chatbots%5C%20son%5C%20aplicaciones%5C%20de,automatizar%5C%20procesos%5C%20en%5C%20su%5C%20empresa.. (accedida: 02-06-2023).

[3] Aldo De Moor y Hans Weigand. “Formalizing the evolution of virtual communities”. En: *Information Systems* 32.2 (2007), págs. 223-247.

[4] Barry Wellman et al. “Computer networks as social networks: Collaborative work, telework, and virtual community”. En: *Annual review of sociology* 22.1 (1996), págs. 213-238.

[5] HN Io y CB Lee. “Chatbots and conversational agents: A bibliometric analysis”. En: *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. IEEE. 2017, págs. 215-219.

[6] Manuel Portela. “Interfacing participation in citizen science projects with conversational agents”. En: *Human Computation* 8.2 (2021), págs. 33-53.

[7] *¿Qué es la inteligencia artificial o IA?* URL: https://cloud.google.com/learn/what-is-artificial-intelligence?hl=es-419. (accedida: 24-05-2023).

[8] Chiara Martino. *What is a Conversational AI?* URL: https://medium.com/women-in-voice/what-is-conversational-ai-an-introduction-to-conversational-interfaces-835c86cee741. (accedida: 24-05-2023).

[9] *NLP VS NLU VS NLG.* URL: https://www.upbe.ai/blog/diferencias-entre-nlp-nlu-y-nlg/. (accedida: 29-05-2023).

[10] *¿Qué es Decidim.Barcelona?* URL: https://www.decidim.barcelona/pages/decidim?locale=es. (accedida: 31-05-2023).

[11] *¿Qué son los procesos participativos?* URL: https://www.decidim.barcelona/pages/processos-participatius?locale=es. (accedida: 24-05-2023).

[12] *Rasa Domain.* URL: https://rasa.com/docs/rasa/domain. (accedida: 06-06-2023).

[13] *Todo lo que necesitas saber sobre Dialogflow.* URL: https://digitalherramienta.com/dialogflow/. (accedida: 02-06-2023).

[14] *¿Qué es Language Understanding (LUIS)?* URL: https://learn.microsoft.com/es-es/azure/cognitive-services/luis/what-is-luis.. (accedida: 02-06-2023).

[15] *IBM Watson Assistant.* URL: https://www.ibm.com/es-es/products/watson-assistant/artificial-intelligence.. (accedida: 02-06-2023).

[16] *Comparativa de nueve frameworks de chatbot.* URL: https://cobusgreyling.medium.com/comparativa-frameworks-chatbot-3e2432683e1e.. (accedida: 02-06-2023).

[17] *ChatGPT.* URL: https://es.wikipedia.org/wiki/ChatGPT. (accedida: 09-06-2023).