



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA

CreaBot

Desarrollo de un asistente para la implementación y despliegue de sistemas conversacionales mediante la plataforma RASA

Autor

Pablo Valenzuela Álvarez

Director

David Griol Barres



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

—
Granada, septiembre de 2022

CreaBot

Desarrollo de un asistente para la implementación y despliegue de sistemas conversacionales mediante la plataforma RASA

Autor

Pablo Valenzuela Álvarez

Director

David Griol Barres

CreaBot: Desarrollo de un asistente para la implementación y despliegue de sistemas conversacionales mediante la plataforma RASA

Pablo Valenzuela Álvarez

RESUMEN

Este Trabajo de Fin de Grado utiliza las características de los sistemas e interfaces conversacionales, los árboles de decisión y otras técnicas estadísticas, con el objetivo de ayudar a resolver problemas específicos de distinta complejidad mediante el desarrollo de chatbots, permitiendo a los usuarios conseguir sus objetivos usando diferentes tipos de conversaciones.

Se ha implementado un asistente capaz de generar los archivos necesarios para configurar un chatbot de la manera que el “cliente” necesite. Lo único que necesita como entrada es un archivo de texto de configuración con el debido formato para que sea interpretado y sean generados los ficheros correspondientes.

La plataforma seleccionada para crear estos chatbots se llama Rasa Open Source. Se trata de una plataforma de código libre que es capaz de hospedar chatbots con la capacidad de extraer el significado de una frase formulada por un usuario y responder en consecuencia. También guarda el contexto de una conversación para después usar esa información para cumplir el objetivo/deseo del usuario.

Los tipos de chatbot que este programa puede crear son: de respuesta a preguntas frecuentes o FAQ (preguntas que se repiten mucho o son similares y para las que hay una sola respuesta), secuencia de pasos (donde guarda el contexto y muestra el resultado final, ejemplo: para realizar una reserva en un restaurante, hotel, etc) y de decisiones en forma de árbol o árbol de decisión (sigue las ramas del árbol y llega a un estado final). La intención es crear un estándar para este tipo de chatbots.

La plataforma usada también permite la integración con muchas de las tecnologías actuales. Entre otras se ha integrado con: el servicio de Telegram, lo cual hace posible el uso de nuestro chatbot en un dispositivo móvil; y con la herramienta llamada NGROK, la cual genera URLs públicas a la dirección donde se encuentra el chatbot (generalmente nuestro servidor apache local), con la cual podemos realizar test y presentar demos al cliente.

Como último apunte, se han incluido en el proyecto algunos algoritmos pertenecientes al ámbito de recuperación y minería de datos, usados para mejorar la funcionalidad de comprensión del lenguaje incluyendo de forma automática nuevos ejemplos al chatbot que no sean muy parecidos a los que ya contiene.

Palabras clave: Sistemas conversacionales, script, árbol de decisión, chatbot, Rasa Open Source, preguntas, frecuentes, FAQ, secuencia, pasos, chatbot integration, NGROK, url,

localhost, apache, Telegram, dispositivos móviles, recuperación de datos, minería de datos, aprendizaje automático, internet de las cosas.

CreaBot: Assistant development for the implementation and deploy of conversational systems using the RASA platform

Pablo Valenzuela Álvarez

ABSTRACT

This Bachelor's Degree Final Project uses the characteristics of the conversational systems, conversational user interfaces, decision trees and other statistical techniques, with the objective of helping to resolve specific problems of distinct complexity through the development of chatbots, allowing users to achieve their objective using different types of conversations.

It's developed an assistant able to generate all the files necessary to configure a chatbot in the way that a final user requires. The only thing that this code needs is a template file with a concrete format for it to be read and generates all the files appropriately.

The platform selected for creating these chatbots is called Rasa Open Source. It is based on an open source platform capable of hosting chatbots with the ability of extracting the meaning of text formulated from a user and responding in consequence. Although it keeps the context of a conversation to use that information to accomplish the user's objectives or wishes.

The kinds of chatbots available to create with this program are: frequently asked question, also known as FAQ (question that are frequently repeated or are similar and have only one answer), sequence of steps (where it saves the context and shows the final result, example: restaurant book, or hotel, etc) and decision in tree shape or decision tree (it follows branches until it reaches the final leaf). The final intention is to create a standard for creating chatbots.

The used platform allows the integration of a lot of existing technologies. Ones of this integrations are: the Telegram service, which makes possible the use of our chatbot in a smartphone; and NGROK tool, that generates public URLs of our chatbot location (usually is our Apache server) and permits run test and demo presentations to clients.

As a last note, it's been included in the project some algorithms related to data recovery and data mining aspects. They are used to improve the level of language understanding for the automatic inclusion of new examples in the chatbot, not the same examples it contains but similar.

Keywords: conversational systems, script, decision tree, chatbot, Rasa Open source, frequently asked question, FAQ, sequence, steps, chatbot integration, NGROK, url,

localhost, apache, Telegram, mobile devices, data recovery, data mining, machine learning, internet of things.

Yo, **Pablo Valenzuela Álvarez**, alumno de la titulación **TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 76652136J, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Pablo Valenzuela Álvarez

Granada a 7 de Septiembre de 2022 .

D. David Griol Barres, Profesor del Departamento de Lenguaje y Sistemas Informáticos de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Creabot: Desarrollo de un asistente para la implementación y despliegue de sistemas conversacionales mediante la plataforma RASA***, ha sido realizado bajo su supervisión por **Pablo Valenzuela Álvarez**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 7 de Septiembre de 2022.

El director:

David Griol Barres

Agradecimientos

Quiero agradecer a todos los que han tenido paciencia conmigo, sobre todo a mi familia. Ha sido una etapa de mi vida con altibajos, y sin este apoyo constante hubiera sido más difícil de conseguir.

ÍNDICE DE CONTENIDOS

Resumen	6
Abstract	8
Índice de contenidos	16
Índice de figuras	19
Índice de tablas	24
Capítulo 1: Introducción	26
1.1. Motivación	26
1.2. Objetivos	28
1.3 Estructura de la memoria	29
Capítulo 2: Estado del arte	31
2.1. Sistemas conversacionales	31
2.2. Programación de una aplicación	33
2.2.1. Plataforma: Rasa Open Source	33
2.2.2 Editor: Visual Studio Code	42
2.2.3. Despliegue web: XAMPP	44
2.2.4. Despliegue web: NGROK	45
2.2.5. Interfaz gráfica: PySimpleGUI	46
2.2.6. Conectividad: Telegram Bot API	47
2.2.7. Extra: Minidom	48
2.3. Algoritmos usados para el cálculo de la similitud semántica	49
2.3.1. Similitud Coseno	49
2.3.2. Okapi BM25	50
2.3.3. Longest Common Subsequence	52
2.3.4. Jaro-Winkler	53

2.3.5. Evaluación de algoritmos	54
2.4. Árboles de decisión y modelos estadísticos	56
2.4.1. Árboles de decisión	56
2.5. Aplicaciones similares	57
2.5.1. Dialogflow	57
2.5.2. Amazon Lex	60
2.5.3. Tkinter	61
2.6. Conclusiones sobre las aplicaciones	62
Capítulo 3: Gestión del proyecto	63
3.1. Planificación y presupuesto	63
3.1.1. Metodología	63
3.1.2. Presupuesto	67
Capítulo 4: Desarrollo del proyecto	71
4.1. introducción	71
4.2. Análisis y diseño de la aplicación	74
4.2.1. Interfaz web	74
4.2.2. Interfaz del programa	79
4.3. Implementación	84
4.3.1. Entrada XML	85
4.3.2. Entrada TXT	88
4.3.3. Actions.py	104
4.3.4. Estructura del programa	107
4.3.5. Uso en Telegram	108
4.4. Compilación y detección de errores	110
4.4.1. Funcionamiento	111
4.4.2. Chequeo de errores	111

Capítulo 5: Conclusiones y líneas futuras	118
5.1. Conclusiones	118
5.2. Líneas futuras	118
Bibliografía	120

ÍNDICE DE FIGURAS

Figura 1: Previsión del incremento en el uso de interfaces orales [4]	27
Figura 2: Diagrama de funcionamiento de un sistema conversacional [1]	31
Figura 3: Diagrama de los componentes del sistema conversacional	33
Figura 4: Ejemplos de procesado de lenguaje natural en Rasa [6]	34
Figura 5: Ejemplo de multi-intent en Rasa	34
Figura 6: Ejemplo de aprendizaje automático en Rasa [6]	35
Figura 7: Servicios que se pueden integrar en Rasa [6]	36
Figura 8: Creación de un entorno virtual en Python	36
Figura 9: Activar un entorno virtual	36
Figura 10: Instalación de Rasa Open Source	36
Figura 11: Contenido de un proyecto de Rasa	37
Figura 12: (De izquierda a derecha) Ejemplo de nlu.yml, rules.yml y stories.yml	38
Figura 13: Ejemplo de domain.yml	39
Figura 14: Grafo de las conversaciones de la versión de prueba	39
Figura 15: Diagrama de casos de uso de la versión de prueba	40
Figura 16: Ejemplo de conversación errónea del chatbot.	41
Figura 17: Usando el aprendizaje interactivo de Rasa para corregir una mala decisión	41
Figura 18: Ejemplo de conversaciones corregidas usando aprendizaje interactivo	41
Figura 19: Pantalla principal de VSCode	43
Figura 20: Lenguajes soportados por VSCode [10]	43
Figura 21: Instalación de extensiones	44
Figura 22: Interfaz de XAMPP	45
Figura 23: Ejemplo de uso de NGROK con el comando ngrok http 80	46
Figura 24: Código de “Hola mundo!” usando PySimpleGUI [17]	47
Figura 25: Ventana de “Hola Mundo!” creada con PySimpleGUI [17]	47

Figura 26: Ejemplo credentials.yml con los datos proporcionados por BotFather [18]	48
Figura 27: Ejemplo de uso de Minidom [20]	48
Figura 28: Ejemplo extracción de etiquetas con Minidom [20]	49
Figura 29: Cálculo del ángulo del coseno. Similitud coseno.	49
Figura 30: implementación de la similitud coseno en Python	50
Figura 31: Implementación de BM25 en Python	51
Figura 32: Implementación de LCS en Python	53
Figura 33: Implementación de la similitud Jarp-Winkler en Python	54
Figura 34: Ejemplo de árbol de decisión para jugar al tenis	56
Figura 35: interfaz principal de Dialogflow [33]	58
Figura 36: Creación de intents en Dialogflow [33]	59
Figura 37: Editando respuestas en Dialogflow [33]	59
Figura 38: Ejemplo de conversación con Amazon Lex [35]	60
Figura 39: Código de Hello World en tkinter [37]	61
Figura 40: Ventana de Hello World en Tkinter [37]	61
Figura 41: Diagrama de las fases del proyecto	63
Figura 42: Tipos de sistemas conversacionales del proyecto	71
Figura 43: Formato de las stories en el chatbot FAQ	72
Figura 44: Diagrama de diseño de un formulario en Rasa	72
Figura 45: Diagrama de diseño del modelo de chatbot basado en árboles de decisión	73
Figura 46: Ejemplo de conversación en la interfaz de Scalableminds	74
Figura 47: Código html interfaz web Scalableminds	75
Figura 48: Interfaz de ejemplo de Chatbot-Widget	76
Figura 49: Página inicial de la interfaz web del chatbot	77
Figura 50: Webchat desplegado a la izquierda, opciones del chatbot a la derecha	78
Figura 51: Interfaz principal del programa	79
Figura 52: Gama de temas que ofrece PySimpleGUI	80

Figura 53: Selección de tipo de chatbot y plantilla de texto	80
Figura 54: Panel de edición de plantilla	81
Figura 55: Selección de algoritmos y parámetros relacionados	82
Figura 56: Nombre, localización y creación del chatbot	83
Figura 57: Código de la interfaz del programa	84
Figura 58: Ejemplo de plantilla FAQ en formato XML	85
Figura 59: Ejemplo de plantilla de Pasos en Secuencia en XML	86
Figura 60: Ejemplo de plantilla de árbol de decisión en XML	87
Figura 61: Ejemplo de identificación de las stories en XML	87
Figura 62: Ejemplo plantilla FAQ en texto plano (TXT)	88
Figura 63: Variables para alojar datos procesados	89
Figura 64: Fragmento de código donde se escriben los ejemplos en nlu.yml	89
Figura 65: Fragmento de código donde se añaden las historias en el chatbot FAQ	89
Figura 66: Página de preguntas frecuentes sobre el correo de la ugr	90
Figura 67: Plantilla de las preguntas frecuentes del correo de la ugr	90
Figura 68: Ejemplo conversaciones. Chatbot FAQ	92
Figura 69: Plantilla usada para el chatbot de pasos en secuencia en texto plano (TXT)	93
Figura 70: Procesado de array de entidades en el chatbot de pasos en secuencia	94
Figura 71: Fragmento de código donde añade las preguntas automáticas en el chatbot de secuencia de pasos	94
Figura 72: Fragmento de código donde crea el fichero rules.yml con los datos necesarios para activar el formulario	95
Figura 73: Fragmento de código donde se muestran la información recogida por el chatbot de pasos en secuencia	95
Figura 74: Funcionamiento del chatbot de recogida de datos	96
Figura 75: Plantilla del chatbot de recogida de datos	96
Figura 76: Archivo nlu.yml correspondiente a la plantilla del chatbot de recogida de datos	97
Figura 77: Ejemplo de conversación con el chatbot de recogida de datos	98

Figura 78: Plantilla del chatbot basado en árboles de decisión	99
Figura 79: Fragmento del código donde se escriben las historias del chatbot basado en árboles de decisión	99
Figura 80: Diagrama usado en el chatbot basado en árboles de decisión	100
Figura 81: Plantilla del ejemplo usado en el chatbot basado en árboles de decisión ..	101
Figura 82: Muestra de como se hace el story correspondiente a la línea 30 de la Figura 81	101
Figura 83: Ejemplo del caso de uso de la Tabla 17 usando el webchat	102
Figura 84: Ejemplo del caso de uso de la Tabla 18 usando el webchat	103
Figura 85: Ejemplo del caso de uso de la Tabla 19 usando el webchat	104
Figura 86: Selección del algoritmo en el constructor del fichero actions.py	104
Figura 87: Fragmento del fichero actions.py correspondiente a los chatbots de FAQs y árbol de decisión	105
Figura 88: Modificación del ejemplo para los chatbots de pasos en secuencia	106
Figura 89: Generación de cada validación de slots	106
Figura 90: Ejemplo de fichero actions.py correspondiente al ejemplo usado en la sección 4.3.2.b	107
Figura 91: Estructura del programa	107
Figura 92: Composición de la carpeta copiar	108
Figura 93: Fragmento de código necesario para activar el chatbot en Telegram	109
Figura 94: Conversación en Telegram Web usando el ejemplo de chatbot de Correo Ugr	109
Figura 95: Conversación en Telegram Web usando el chatbot de recogida de datos ..	110
Figura 96: Conversación en Telegram Web usando el chatbot del árbol de fruta	110
Figura 97: Contenido archivo server_actions.bat	111
Figura 98: Contenido del archivo server_html.bat	111
Figura 99: Diagrama con los datos que chequea la función	112
Figura 100: Errores al detectar entidades y responses en la plantilla de preguntas frecuentes	112
Figura 101: Chequeo de errores para los intents en el chatbot de preguntas frecuentes	113

Figura 102: Chequeo de errores para los stories en el chatbot de preguntas frecuentes	113
Figura 103: Chequeo de errores para las entidades en secuencia de pasos	114
Figura 104: Chequeo de errores para los intents en secuencia de pasos	114
Figura 105: Chequeo de responses en el chatbot de árbol de decisión	115
Figura 106: Cheque de stories en el chatbot de árbol de decisión	115
Figura 107: Comprobaciones finales del chequeo de datos	116
Figura 108: Aviso de errores pasando la plantilla vista en la figura 67 a un chatbot de Preguntas frecuentes	116
Figura 109: Aviso de errores al crear un chatbot basado en árboles de decisión con la plantilla de la figura 67	117

ÍNDICE DE TABLAS

Tabla 1: Ejemplo de funcionamiento LCS 1	52
Tabla 2: Ejemplo de funcionamiento LCS 2	52
Tabla 3: Comparación de algoritmos 1	54
Tabla 4: Comparación de algoritmos 2	55
Tabla 5: Comparación de algoritmos 3	55
Tabla 6: Comparación de algoritmos 4	55
Tabla 7: Recursos hardware utilizados	67
Tabla 8: Presupuesto de los recursos hardware	67
Tabla 9: Recursos software utilizados	68
Tabla 10: Presupuesto de los recursos software	68
Tabla 11: Presupuesto de los recursos personales	69
Tabla 12: Coste total del proyecto	69
Tabla 13: Plantilla para la definición de los casos de uso	84
Tabla 14: Ejemplo caso de uso 01. Chatbot FAQ	91
Tabla 15: Ejemplo caso de uso 02. Chatbot FAQ	91
Tabla 16: Ejemplo caso de uso 01. Chatbot pasos en secuencia	97
Tabla 17: Ejemplo caso de uso 01. Chatbot basado en árbol de decisión	102
Tabla 16: Ejemplo caso de uso 02. Chatbot basado en árbol de decisión	102
Tabla 16: Ejemplo caso de uso 03. Chatbot basado en árbol de decisión	103

Capítulo 1: INTRODUCCIÓN

El objetivo de este capítulo es describir el contexto de este Trabajo de Fin de Grado, así como los problemas que se pretenden resolver acerca de los sistemas conversacionales y los objetivos a alcanzar a la finalización del desarrollo del proyecto para solucionar los problemas planteados. Por último, se presenta la estructura utilizada para organizar los contenidos de esta memoria.

1.1. MOTIVACIÓN

Desde que podemos recordar, nuestra especie siempre ha tratado de comunicarse con seres creados artificialmente. Hay bastantes ejemplos tanto en el cine como en la literatura, remontándonos a las eras mitológicas griega y romana podemos encontrar historias de héroes en las que podían hablar con dioses o guerreros hechos de bronce. Durante los siglos XVIII y XIX se desarrollaron los primeros intentos de crear autómatas capaces de imitar el comportamiento humano, se hicieron estudios que sintetizaban sonidos parecidos a la voz humana, y se crearon las primeras máquinas eléctricas [1].

En la década de los 40 del siglo pasado, fue cuando empezó el desarrollo de las primeras computadoras. En esta época Alan Turing creó una manera de medir la inteligencia de un sistema denominado “Test de Turing” [2]. Esta prueba determina la habilidad de una máquina de imitar el comportamiento humano manteniendo un diálogo con una persona, la prueba finaliza con éxito cuando la persona es incapaz de determinar si su interlocutor es una máquina u otro humano [1]. Hace 8 años, una máquina llamada “Eugene” consiguió pasar esta prueba engañando al 33% del jurado haciéndose pasar por un niño de trece años, siendo así la primera en superar este test [3].

Hoy día las interfaces orales están experimentando un gran auge, algunas fuentes apuntan a que este mercado alcanzará los 28 mil millones de dólares en 2027 de inversión (ver Figura 1). Especialmente la zona de Asia Pacífico es en la que se espera que se experimente un mayor crecimiento. Este mercado está en auge gracias a los avances tecnológicos, al cambio de conciencia que han tenido las masas con respecto a esta tecnología, y los bajos costes de los dispositivos de reconocimiento de texto y de voz. China es la mayor responsable de este crecimiento ya que posee las dos empresas más grandes dedicadas al mercado del reconocimiento de voz y texto (Baidu y iFlyTek). También se prevé que la India experimente el mayor crecimiento en este periodo [4].



Figura 1: Previsión del incremento en el uso de interfaces orales [4]

Este interés que se ha generado viene dado por los avances en la Inteligencia Artificial y al Deep Learning, donde se han mejorado las tasas de acierto y su uso es más sencillo para nuevos usuarios y contextos interactivos. Los factores principales que han contribuido a su éxito reciente pueden clasificarse de la siguiente manera:

- la gran mejora en las unidades de procesamiento de gráficos (GPUs) con lo que se obtiene la posibilidad de hacer más cálculos masivos en paralelo, y benefician bastante a los sistemas que usan redes neuronales;
- el Big Data, es decir, grandes cantidades de datos, hacen que los sistemas sean capaces de aprender y ser cada vez más “listos”;
- los nuevos algoritmos de Deep Learning usan estas arquitecturas GPU y que trabajan con enormes cantidades de datos.

En otro orden de cosas, hay que destacar la llegada de los teléfonos y dispositivos inteligentes, que ofrecen mucho potencial y funcionalidad sobre todo en estos últimos años. Además de funcionalidades como micrófono y altavoz, incorporan información sobre: la ubicación del usuario, tiempo y fecha, contactos y el calendario. Esta información puede ser integrada en un interfaz conversacional para personalizar un sistema.

Mirando a nuevos escenarios, gracias a las redes inalámbricas que cada vez son más rápidas, sumada a la disponibilidad casi omnipresente que nos brindan las conexiones WiFi, las mejoras en los procesadores y la aparición de la computación en la nube, han facilitado tareas como la de reconocimiento de voz y permitido que se pueda ejecutar en la nube con ordenadores dedicados. En la actualidad, tecnologías como FFSR (*Far Field Speech Recognition*) permite el reconocimiento de voz entre distintos tipos de dispositivos (electrodomésticos, coches, etc), dando resultado a productos para la interacción dentro de entornos domésticos como son Amazon Echo y Google Home [1].

Últimamente, muchas de las empresas más grandes se han lanzado al mercado de los dispositivos conversacionales, tenemos los ejemplos de Apple con Siri, Amazon y Alexa, Microsoft con Cortana, Duer de Baidu, etc. Con estos dispositivos las empresas pueden crear perfiles específicos de sus clientes que les permite personalizar servicios de comercio electrónico.

Por consiguiente son abundantes las herramientas para crear agentes conversacionales o chatbots: dialogflow, Amazon Lex, chatfuel, y opciones open source como Rasa. Estos se han convertido en una herramienta muy importante de cara a la resolución de problemas a clientes de manera rápida y sencilla.

Los chatbots cada vez son más usados en ámbitos como la educación, recuperación de información, negocios y comercio electrónico. Ofrecen servicios como asistentes en línea que complementan o sustituyen directamente a un ser humano [1]. Según un estudio realizado por Findasense: hay mucha gente reticente a la hora de confiar en los chatbots. Ese estudio mostró que más de un 60% de los usuarios se sintieron satisfechos interactuando con el chatbot, sin embargo dicho porcentaje afirmó que hubieran preferido ser atendidos por un humano. Las personas están familiarizadas con el uso de chatbots en gran medida gracias a la pandemia global del Covid19, sin embargo el reto está en ofrecer una experiencia hiper-personalizada sin perder la calidez que una persona puede ofrecer, explicaba Paula Antelo (regional business strategy lead en Findasense). También puntualizó que el 45% de las empresas encuestadas optarán por “humanizar” el servicio de su chatbot para 2022 [5].

El reto que se nos presenta es similar al Test de Turing pero sin la necesidad de engañar a nadie. La gente prefiere la atención humana a la de un chatbot por lo que es necesario desarrollar sistemas fáciles de usar y con interacciones casi humanas tanto si la comunicación se da por escrito como hablada. Un chatbot que responda por voz dará más la sensación de “humano”.

Otra razón para el proyecto es el gran mercado que abarca y abarcará en el futuro próximo según las estimaciones antes mostradas (ver Figura 1). Cada vez hay más empresas interesadas en el desarrollo de agentes de voz y las que ya están presentes son algunas de las más grandes en el mundo. Esto junto a la facilidad que existe para desarrollar un chatbot hoy día dada la gran cantidad de herramientas, hace que sea cada vez más apetecible “meterse en este mundillo”.

Por estas razones nace el proyecto CreaBot, con el objetivo de facilitar la vida tanto a usuarios como programadores.

1.2. OBJETIVOS

En este apartado comentaré los diferentes objetivos que se definieron para el proyecto. Se abordará cada uno, desde su justificación hasta su futura implementación, así como sus limitaciones.

Un objetivo fundamental de este proyecto es conseguir que un usuario pueda mantener una conversación con nuestro sistema conversacional y llegue a un fin satisfactorio, es decir, que el usuario pueda resolver un problema, tanto si es una pregunta sencilla como si quiere hacer una reserva a un hotel, usando los diferentes tipos de chatbots que pueden desarrollarse utilizando el asistente. Otro objetivo del proyecto es simplificar la creación de chatbots para que gente sin experiencia en programación pueda tener su propio chatbot.

Para ello se ha desarrollado un asistente que es capaz de generar varios tipos de chatbots. La aplicación necesita un archivo de texto que contenga los elementos (preguntas, respuestas, entidades e historias) que necesita el tipo de chatbot que desee crear.

Los objetivos específicos pueden clasificarse de la siguiente forma:

- Elección de las aplicaciones y herramientas
- Diseño de la arquitectura
 - Extracción de datos
 - Gestión de datos
 - Creación de un chatbot
 - Gestión de nuevos ejemplos
- Diseño de la interfaz gráfica
 - Chequeo de datos
- Conectividad con otros sistemas

1.3. ESTRUCTURA DE LA MEMORIA

La estructura del proyecto consta de cinco capítulos y de una bibliografía. El contenido de cada capítulo se describe brevemente a continuación.

- **Capítulo 1. Introducción:** se explica el contenido del proyecto. La motivación y los problemas que pretende resolver, los objetivos a seguir y la propia estructura del proyecto.
- **Capítulo 2. Estado del arte:** se exponen los conceptos usados, las principales herramientas y aplicaciones utilizadas durante el desarrollo del proyecto, y alternativas a ellas.
- **Capítulo 3. Gestión del proyecto:** se explica la metodología y procesos empleados en el desarrollo. También se planifica un presupuesto para el proyecto.
- **Capítulo 4. Desarrollo del Proyecto:** la planificación seguida y las decisiones tomadas durante el desarrollo del proyecto y la propia implementación de él.

- **Capítulo 5. Conclusiones y líneas futuras:** las opiniones finales sobre el trabajo. Posibles ampliaciones o mejoras aplicables, y futuras vías de desarrollo que presenta.
- **Bibliografía:** contiene las referencias de las citas utilizadas durante el desarrollo del documento.

Capítulo 2: ESTADO DEL ARTE

En este capítulo se hace una breve explicación de los conceptos que aparecen en el proyecto. También se comentarán las aplicaciones y herramientas usadas en el desarrollo y los algoritmos utilizados para calcular la similitud semántica. Para finalizar expondré otros sistemas similares a los usados en el proyecto.

2.1. SISTEMAS CONVERSACIONALES

Por definición, un sistema conversacional se puede comprender como un sistema automático con la capacidad de mantener una conversación con una persona, admitiendo una entrada de información y proporcionando salidas en lenguaje natural. Un ejemplo del funcionamiento de un sistema conversacional se muestra en la Figura 2.

Este sistema debería estar compuesto de, al menos, algunas de estas características:

- Poder referenciar información dada por el usuario anteriormente.
- Ser capaz de redirigir el diálogo en uno, o unos, de los entornos definidos.
- Pedir la información necesaria para conseguir el objetivo deseado.
- Solicitar una aclaración, si hay confusión en la información aportada por el usuario.

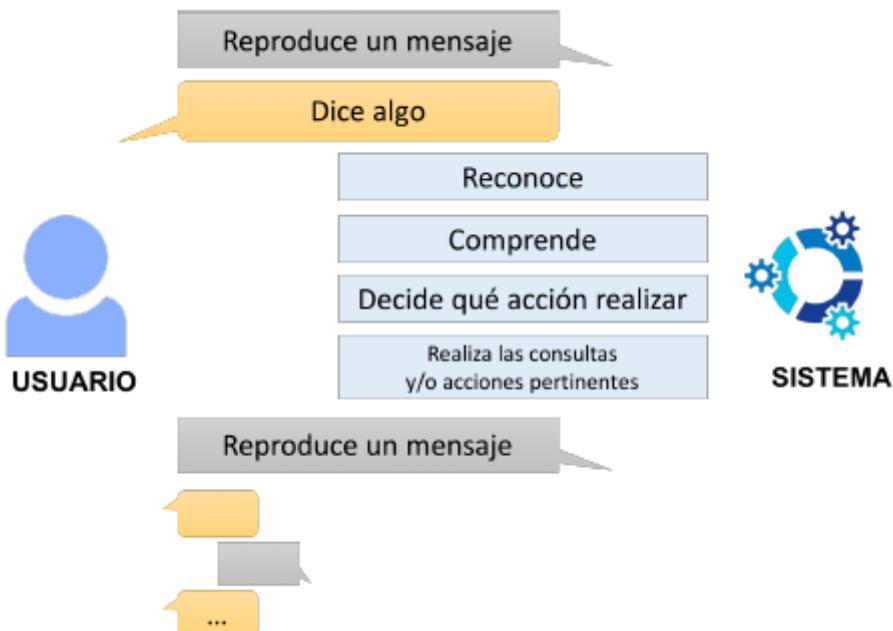


Figura 2: Diagrama de funcionamiento de un sistema conversacional [1]

El desarrollo de una aplicación capaz de mantener conversaciones con otras personas hoy día nos presenta un gran reto: hay que tener en cuenta la gran cantidad de fuentes de conocimiento con distinta estructura que existen y también, las limitaciones de las tecnologías usadas para recibir información del usuario. Sin embargo, hoy día son constantes los avances en el Procesamiento del Lenguaje Natural, las Tecnologías del Habla, la Inteligencia Artificial y los Dispositivos Móviles, permiten que sea fácil y más flexible la comunicación entre humano y máquina.

Algunas aplicaciones de los sistemas conversacionales son:

- Sistemas que muestran información sobre los transportes públicos.
- Sistemas de atención en el ámbito médico.
- Sistemas de banca online.
- De Turismo.
- Aplicaciones accesibles dentro de vehículos.
- Sistemas de accesibilidad para personas con discapacidades.
- Aplicaciones de educación.
- Asistente para dispositivos móviles.
- Sistemas de interacción en el hogar y control domótico.
- Robots y sistemas wearables.
- Etcétera ...

Se puede observar que el número de entornos y tareas en los que se pueden aplicar los sistemas conversacionales son muy extensos. Este proyecto se centrará en la creación de sistemas de conversación mediante texto, lo que se denominaría chatbot.

Este término, chatbot, suele englobar a aplicaciones en las que el usuario interactúa en modo texto con el propio sistema y las conversaciones pueden ser de cualquier tipo y tratar cualquier tópico. Este tipo de asistentes se usan en competiciones relacionadas en el Test de Turing, pero recientemente este término se ha ido relacionando con aplicaciones que respondan preguntas frecuentes (FAQ), utilizando una capacidad limitada de gestión de diálogo, detección de intenciones y reconocimiento de entidades.

En contraposición, los sistemas de diálogo entienden que el usuario tiene un objetivo, y que ese objetivo se tiene que cumplir mediante la interacción con el sistema. Por lo tanto, este debe buscar la mejor manera de cumplir este fin.

Pero esta característica no impide que los sistemas de diálogo lleguen a alcanzar altos grados de complejidad y que incluso tengan que ser capaces de gestionar diálogos sobre diversos temas. Por ejemplo, un sistema de gestión de viajes podría ser capaz de conversar sobre reservas de hotel, medios de transporte o actividades que se pueden realizar en el destino del usuario interesado.

Hoy día, con la proliferación de sistemas basados en tecnologías del lenguaje, el término chatbot se está destinando a cualquier sistema con el que es posible mantener una conversación en lenguaje natural, y también para los sistemas orientados a tareas específicas. Por ejemplo: los asistentes virtuales como Siri, Alexa, Cortana o Google Assistant están considerados chatbots al igual que las aplicaciones que pueden prestar servicios ininterrumpidos en Facebook, Telegram o Slack [1].

2.2. PROGRAMACIÓN DE UNA APLICACIÓN

Esta sección engloba los distintos programas, herramientas y aplicaciones utilizadas durante el desarrollo del proyecto. Se explicará en qué consisten y el porqué de su uso. En la Figura 3 se muestran las distintas secciones en las que se divide este punto.



Figura 3: Diagrama de los componentes del sistema conversacional

2.2.1. Plataforma: Rasa Open Source

Rasa Open Source (o Rasa) es una plataforma de código abierto orientada a crear asistentes virtuales capaces de comunicarse automáticamente con usuarios mediante texto o voz [6].

2.2.1.1. Características principales de Rasa

Con Rasa tenemos un sistema que nos permite:

1. La extracción del significado de los mensajes.
2. La capacidad de mantener conversaciones complejas.
3. El aprendizaje interactivo.
4. La conexión con otros servicios.

Extracción del significado de los mensajes

Rasa provee un sistema basado en el procesamiento de lenguaje natural capaz de convertir los mensajes proporcionados por el usuario en intenciones (intents) y entidades (entities) que los chatbots pueden entender. Este sistema está basado en librerías de aprendizaje automático de bajo nivel como Tensorflow o spaCy, que dotan a Rasa de un software accesible y personalizable a las necesidades del programador [7]. En la Figura 4 puede observarse el proceso de reconocimiento mencionado.



Figura 4: Ejemplos de procesado de lenguaje natural en Rasa [6]

Mantenimiento de conversaciones complejas

En el mundo real, los mensajes de los usuarios pueden ser difíciles de predecir o interpretar, llegando a ser bastante complejos en algunos casos, lo que complica la tarea de mapearlos en un único intent. Por eso Rasa está diseñado para manejar múltiples intents en un solo mensaje, imitando la forma de hablar de los usuarios. Un ejemplo sería el mostrado en la Figura 5, en el que un cliente de un banco quiere mirar su cuenta bancaria y transferir dinero [7].

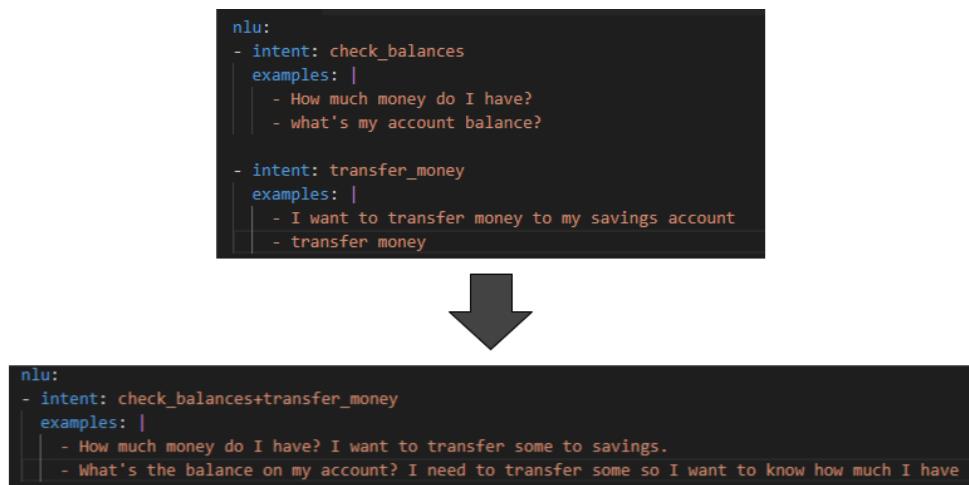


Figura 5: Ejemplo de multi-intent en Rasa

Adicionalmente, Rasa consta de elementos que le permiten guardar el contexto de una conversación, los llamados slots (ver Figura 13). Estos elementos suelen estar vinculados a una entidad y son la memoria del chatbot. Son accesibles en todo momento y se usan para dar mejor respuesta al usuario. Por ejemplo, un usuario escribiría: “quiero ir a un restaurante chino en Granada” y el sistema tendrá que reconocer las entidades “tipo de restaurante/comida” en el caso de restaurante chino y “ciudad” en el de Granada, guardar su valor en los slots correspondientes, y por último, acceder al valor de estos y mostrar una lista de los restaurantes en la localización indicada por el usuario.

Aprendizaje interactivo

Aparte de las conversaciones definidas en un chatbot, Rasa brinda la posibilidad de crear nuevas mediante el uso del aprendizaje automático. Con esta herramienta podemos generar nuevos casos de diálogo o corregir errores (como se puede ver en la Figura 6), simplemente conversando con nuestro chatbot. Un ejemplo de uso de aprendizaje interactivo es mostrado en la sección 2.2.1.4 de la memoria.

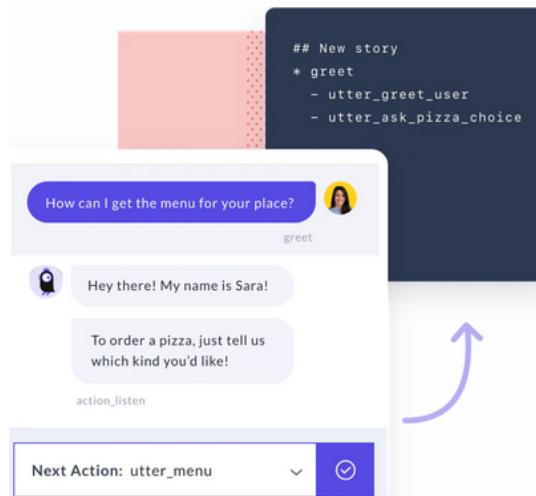


Figura 6: Ejemplo de aprendizaje automático en Rasa [6]

Conexión con otros servicios

Rasa dispone de multitud de conexiones con otros servicios (ver Figura 7) ya sean servicios de mensajería o canales de voz como:

- Facebook Messenger
- Slack
- Telegram (ver sección 2.2.6)
- Twilio
- Cisco Webex Teams
- Google Hangouts Chats

Y de servicios para el despliegue en entornos web como NGROK [9] (ver sección 2.2.4).



Figura 7: Servicios que se pueden integrar en Rasa [6]

2.2.1.2. Instalación de Rasa

En esta breve sección comentaré los requisitos necesarios para instalar Rasa en un equipo y hacer funcionar un chatbot.

Rasa Open Source funciona sobre la versión de Python 3.7 y 3.8, por lo que es necesario tener al menos una de estas versiones activas en tu equipo. Otro aspecto importante es que Rasa necesita de su propio entorno virtual para funcionar.

```
C:\> python3 -m venv ./venv
```

Figura 8: Creación de un entorno virtual en Python

En la Figura 8 se muestra una de las formas de crear un entorno virtual, esta en concreto creará el entorno “venv” en el directorio C. Ahora necesitamos activarlo, para ello se debe ejecutar el comando mostrado en la siguiente figura (ver Figura 9).

```
C:\> .\venv\Scripts\activate
```

Figura 9: Activar un entorno virtual

Activado el entorno solo queda instalar Rasa (ver Figura 10). Esta orden instalará todo lo necesario para ejecutar Rasa en nuestro equipo.

```
pip3 install rasa
```

Figura 10: Instalación de Rasa Open Source

Una vez instalado Rasa y estando dentro del entorno virtual podemos ejecutar una lista de comandos específicos de Rasa. Los más importantes son los siguientes:

- **rasa init** → crea un chatbot en la ubicación donde se encuentra.

- **rasa train** → contando que está en la ubicación de un chatbot, lo entrena y genera un modelo.
- **rasa shell** → te permite conversar con el chatbot usando la consola de comandos y un modelo previamente entrenado.
- **rasa interactive** → inicia la herramienta de aprendizaje interactivo.
- **rasa run** → activa el chatbot usando el modelo entrenado.
- **rasa run actions** → activa el servidor de acciones, es necesario cuando tienes funciones personalizadas.

Por último, cabe resaltar que es necesario tener activo el entorno donde está instalado Rasa cada vez que quieras hacer uso de un chatbot.

2.2.1.3. Contenido de Rasa

En esta subsección explicaré el contenido de un proyecto en Rasa, destacando los elementos más importantes.

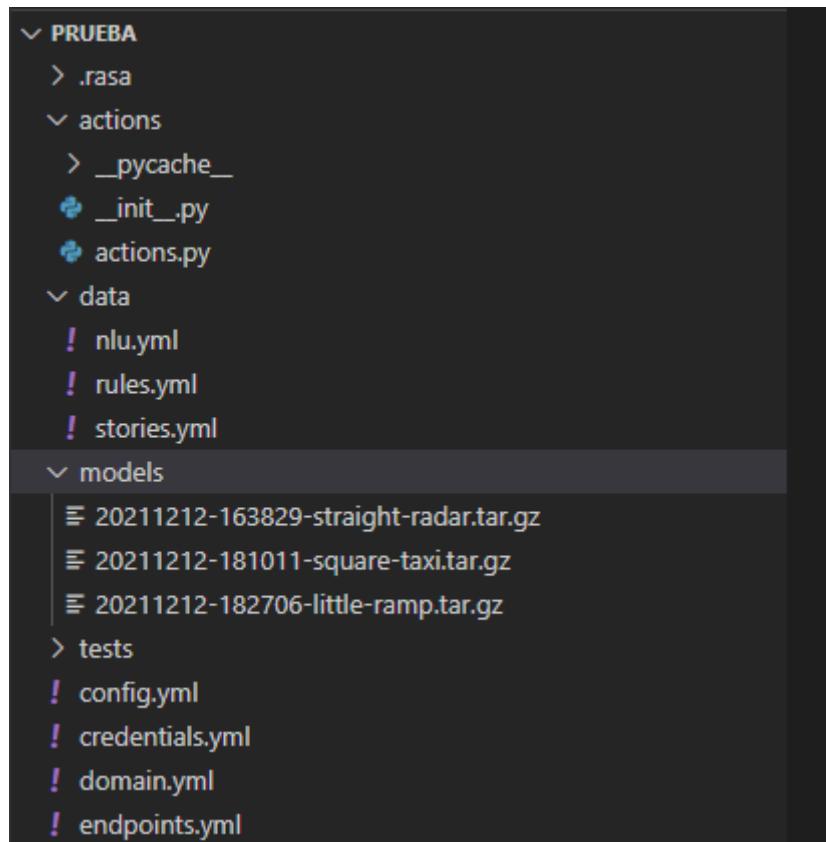


Figura 11: Contenido de un proyecto de Rasa

Como se puede observar en la Figura 11, hay varios ficheros divididos en carpetas. La siguiente lista contiene una breve explicación de los más importantes:

- **Actions:** lugar donde Rasa guarda las funciones en código Python. Suelen estar alojadas en el fichero *actions.py*. Las funciones varían desde la validación de slots hasta el uso de APIs.
- **Data:** donde se encuentran los ejemplos de diálogo y los flujos conversación que sigue el bot. Los ficheros:
 - **nlu.yml:** contiene los ejemplos de diálogo en formato de lista (Ver Figura 12). Cada intent debe contener un ejemplo o más para que el chatbot pueda predecir correctamente.
 - **rules.yml:** son flujos de comunicación cerrados y que se dan siempre (Ver Figura 12). Suele usarse para conversaciones de una única respuesta o para activar formularios.
 - **stories.yml:** ejemplos de conversaciones que el chatbot intentará seguir. (Ver Figura 12).
- **Models:** aquí están los modelos entrenados que usa Rasa para que el chatbot funcione correctamente. Debe haber al menos uno para poder lanzar el chatbot.
- **credentials.yml:** donde se dan acceso a diferentes APIs. (ver Figura 18).
- **domain.yml:** alberga todo el contenido del chatbot, puede contener (ver Figura 13):
 - las entidades (entities).
 - los intents (intents).
 - la memoria (slots).
 - los formularios (forms), donde el chatbot pide datos para llenar los slots.
 - respuestas a intents (responses).
 - acciones (actions), que son las funciones en Python alojadas en *actions.py*.

The figure shows three code editors side-by-side, each displaying a YAML configuration file for Rasa NLU.

- nlu.yml:** Contains examples for various intents like greet, goodbye, and affirm.
- rules.yml:** Contains rules and steps for intents like goodbye and bot_challenge.
- stories.yml:** Contains stories for paths like happy, sad, and interactive_story_1.

```

! nlu.yml
version: "3.0"
nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - hello there
    - good morning
    - good evening
    - moin
    - hey there
    - let's go
    - hey dude
    - goodmorning
    - goodevening
    - good afternoon
- intent: goodbye
  examples: |
    - cu
    - good by
    - cee you later
    - good night
    - bye
    - goodbye
    - have a nice day
    - see you around
    - bye bye
    - see you later
- intent: affirm
  examples: |
    - yes
    - y
    - Indeed
    - of course
    - that sounds good
    - correct

```

```

data > ! rules.yml
1 version: "3.0"
2
3 rules:
4
5   - rule: Say goodbye anytime the user says goodbye
6     steps:
7       - intent: goodbye
8       - action: utter_goodbye
9
10  - rule: Say 'I am a bot' anytime the user challenges
11    steps:
12      - intent: bot_challenge
13      - action: utter_lamabot
14

```

```

data > ! stories.yml
1 version: "3.0"
2
3 stories:
4
5   - story: happy path
6     steps:
7       - intent: greet
8       - action: utter_greet
9       - intent: mood_great
10      - action: utter_happy
11
12   - story: sad path 1
13     steps:
14       - intent: greet
15       - action: utter_greet
16       - intent: mood_unhappy
17       - action: utter_cheer_up
18       - action: utter_did_that_help
19       - intent: affirm
20       - action: utter_happy
21
22   - story: sad path 2
23     steps:
24       - intent: greet
25       - action: utter_greet
26       - intent: mood_unhappy
27       - action: utter_cheer_up
28       - action: utter_did_that_help
29       - intent: deny
30       - action: utter_goodbye
31
32   - story: interactive_story_1
33     steps:
34       - intent: greet
35       - action: utter_greet
36       - intent: mood_great
37       - action: utter_happy

```

Figura 12: (De izquierda a derecha) Ejemplos de nlu.yml, rules.yml y stories.yml

```

1 version: "3.0"
2 intents:
3   - greet
4   - goodbye
5   - affirm
6   - deny
7   - bot_challenge
8   - inform
9 entities:
10  - name
11  - phone
12 slots:
13    name:
14      type: text
15      mappings:
16        - type: from_entity
17          entity: name
18    phone:
19      type: float
20      mappings:
21        - type: from_entity
22          entity: phone
23 forms:
24   data_form:
25     required_slots:
26       - name
27       - phone
28
29 responses:
30   utter_greet:
31     - text: "Hey! Could you give me some of your personal data?"
32   utter_goodbye:
33     - text: "Bye"
34   utter_iamabot:
35     - text: "I am a bot, powered by Rasa."
36   utter_ask_name:
37     - text: "What's your name?"
38   utter_ask_phone:
39     - text: "Your phone number?"
40   utter_submit:
41     - text: "Thanks for the info!"
42   utter_slots_values:
43     - text: "* Name -> {name}\n* Phone -> {phone}"
44 actions:
45   validate_data_form
46 session_config:
47   session_expiration_time: 60
48   carry_over_slots_to_new_session: true
49
50

```

Figura 13: Ejemplo de domain.yml

2.2.1.4. Conversando con la versión de prueba

Al crear un chatbot, Rasa nos proporciona datos de prueba como ejemplo (ver Figuras 11, 12 y 13). Si decidimos entrenar ese modelo podemos crear un chatbot capaz de mantener una conversación bastante simple pero que tiene objetivos y diferentes caminos que el chatbot puede seguir.



Figura 14: Grafo de las conversaciones de la versión de prueba

En la Figura 14 podemos observar que el chatbot consta de tres caminos a seguir según nuestras respuestas. El chatbot empieza preguntando “cómo estás?” ante nuestro saludo, y según las respuestas:

1. Si respondemos que estamos bien, el chatbot responde “genial, sigue así!” y finaliza la conversación.
2. Si respondemos que no estamos bien, el chatbot intenta animarnos mostrando una foto. Seguido el chatbot nos pregunta si nos gustó la foto y si respondemos positivamente, llega al mismo estado final que en el punto 1 y termina la conversación.

- Partiendo del punto 2; si respondemos negativamente ante la foto, el chatbot acaba con el diálogo despidiéndose.

Analizando el código del chatbot, podemos obtener el diagrama de casos mostrado en la Figura 15 , donde la conversación busca llegar a dos estados finales. Y se pueden observar claramente los tres caminos antes comentados.

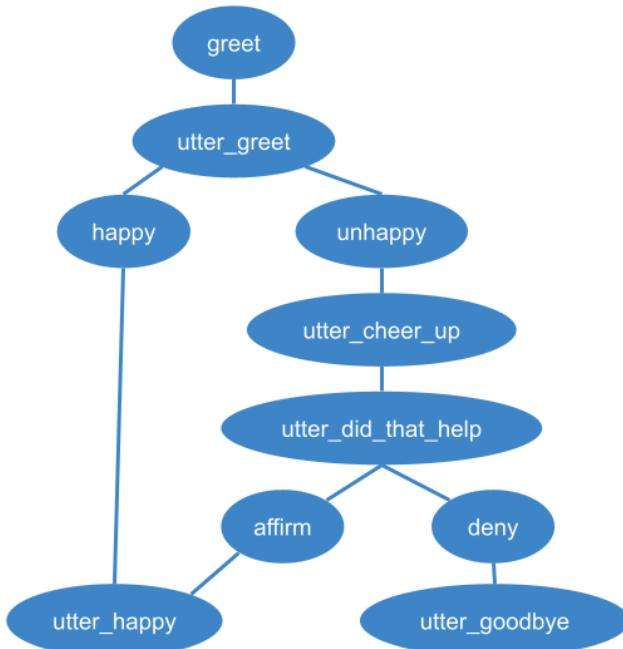


Figura 15: Diagrama de casos de uso de la versión de prueba

2.2.1.5. Usando la herramienta de aprendizaje interactivo

Una de las características comentadas en punto anterior 2.2.1.1, es la de que Rasa es capaz de aprender interactivamente. Al no disponer del completo funcionamiento de Rasa X (explicado en el punto 2.2.1.6), la forma de iniciar el aprendizaje interactivo es desde línea de comandos. Hacer esto es bien sencillo y basta con seguir estos tres pasos:

1. Abrir la consola de windows y activar el entorno de Rasa.
2. Situarnos en el directorio donde está ubicado el chatbot.
3. Ejecutar la orden → *rasa interactive*

En el ejemplo de la Figura 16, se observa como el chatbot comete un error a la hora de calificar la palabra “happy” como “mood_unhappy” (si nos referimos a los estados vistos en la Figura 15). Literalmente significa lo contrario y es una buena ocasión para probar el aprendizaje interactivo

```
Your input -> hi
Hey! How are you?
Your input -> happy
Here is something to cheer you up:
Image: https://i.imgur.com/nGF1K8f.jpg
Did that help you?
Your input -> []
```

Figura 16: Ejemplo de conversación errónea del chatbot.

```
? Your input -> happy
? Your NLU model classified 'happy' with intent 'mood_unhappy' and there are no entities, is this correct? No
? What intent is it? (Use arrow keys)
<create_new_intent>
1.00 mood_unhappy
0.00 greet
0.00 deny
0.00 goodbye
0.00 bot_challenge
0.00 affirm
» 0.00 mood_great
```

Figura 17: Usando el aprendizaje interactivo de Rasa para corregir una mala decisión

En la Figura 17 podemos ver cómo efectivamente considera mal la palabra “happy”. Aquí es donde la herramienta nos pregunta si lo ha hecho bien, y como no es el caso, le decimos que no y buscamos en la lista de intents “mood_great” que es el que corresponde con esa palabra. De esta manera le hemos enseñado al chatbot a responder correctamente cuando reciba la interacción “happy”.

Ahora Rasa se encarga de añadir la palabra a la lista de ejemplos del intent automáticamente, y para que los cambios se hagan efectivos hay que entrenar de nuevo para generar el modelo actualizado. Por último, como se vé en la figura 18, los cambios funcionan correctamente.

```
Your input -> hi
Hey! How are you?
Your input -> happy
Great, carry on!
Your input -> hi
Hey! How are you?
Your input -> unhappy
Here is something to cheer you up:
Image: https://i.imgur.com/nGF1K8f.jpg
Did that help you?
Your input -> []
```

Figura 18: Ejemplo de conversaciones corregidas usando aprendizaje interactivo

2.2.1.6. Conclusiones sobre Rasa Open Source

Para finalizar con la sección de Rasa quiero comentar un par de puntos negativos que tiene esta plataforma. Estos puntos vienen de errores o dificultades que me han ocurrido durante el desarrollo.

Aunque Rasa ofrece una interfaz de desarrollo parecida a la de Dialogflow llamada RasaX, no he conseguido hacerla funcionar. El principal error venía al cargar los modelos ya generados por mí en línea de comandos. De esta manera no podía entablar conversaciones ni hacer test a los chatbots que iba creando, lo que me dejó la opción de encontrar un editor para hacer el código a mano (ver punto 2.2.2).

Otro problema con esta plataforma, es que recientemente Rasa se ha actualizado de la versión 2.0 a la 3.0. No es que sea un problema en sí, ya que la documentación sí que está actualizada a la nueva versión [8], pero la mayoría de tutoriales y videotutoriales (incluso del canal oficial de Rasa) que se encuentran en las plataformas de vídeo usan las versiones antiguas.

Como conclusión del apartado, decir que Rasa Open Source ofrece una plataforma muy sencilla de usar, puedes crear un chatbot funcional con una orden de consola, es bastante sencillo de programar y añadir funcionalidades una vez conoces su formato, y ofrece mucho potencial de cara al desarrollo en conjunto con otros servicios.

2.2.2. Editor: Visual Studio Code

Visual Studio Code o VSCode [10] es un editor de código multiplataforma creado por Microsoft para los sistemas operativos Windows, Linux y macOS. VSCode ofrece a los desarrolladores soporte para múltiples lenguajes de programación (ver Figura 20), soporte de depuración, resaltado de sintaxis, autocompletado de texto y control de Git entre otras características [11].

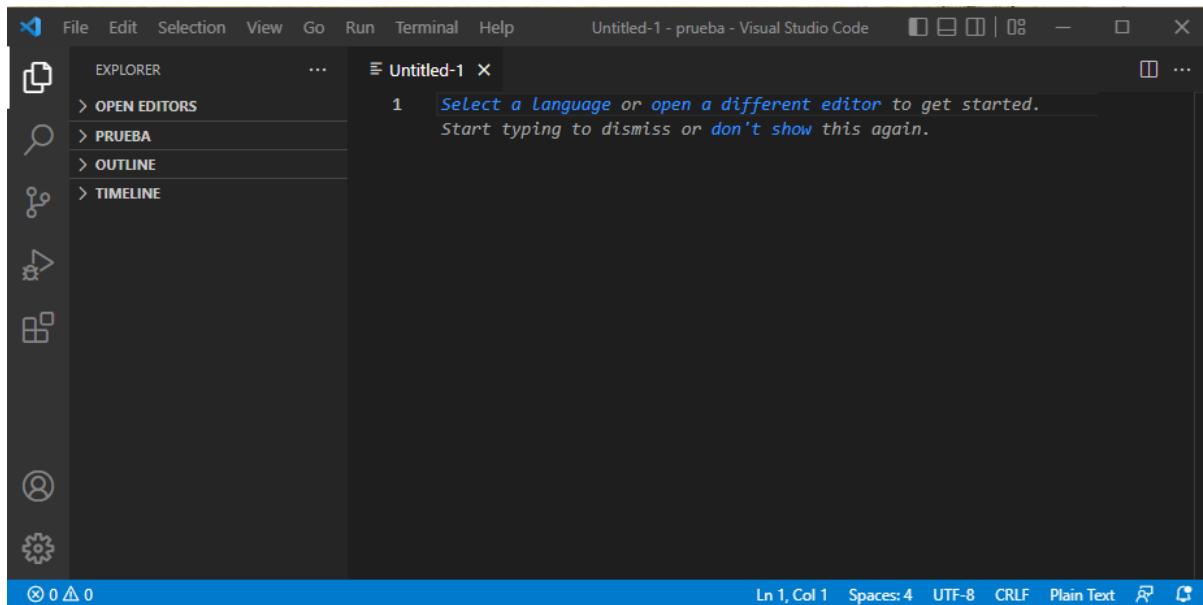


Figura 19: Pantalla principal de VSCode

Creado en 2015, un año después de su lanzamiento llegó al puesto número 13 en la lista de las herramientas de desarrollo más populares. Desde 2018 hasta la fecha de hoy tiene el primer lugar en esta lista [12]. Lo que demuestra lo rápido que se ha hecho con el mercado dado a su facilidad de uso y herramientas que ofrece.

Características	Lenguajes
Resaltado de sintaxis	Archivo batch · C · C# · C++ · CSS · Clojure · CoffeeScript · Diff · Dockerfile · F# · Fortran · Git-commit · Git-rebase · Go · Groovy · HLSL · HTML · Handlebars · archivo INI · JSON · Java · JavaScript · JavaScript React (JSX) · Less · Lua · Makefile · Markdown · Objective-C · Objective-C++ · PHP · Perl · Raku · PowerShell · Properties · Pug template language, ^{8 9} · Python · R · Razor · Ruby · Rust · SQL · Sass · ShaderLab · Shell script (Bash) · Swift · TypeScript · TypeScript React (TSX) · Visual Basic · XML · XQuery · XSL · YAML
Snippets	Groovy · Markdown · Nim ¹⁰ · PHP · Swift
Autocompletado de código	CSS · HTML · JavaScript · JSON · Less · Sass · TypeScript
Refactorización	C# · TypeScript
Depuración	<ul style="list-style-type: none"> • JavaScript and TypeScript for Node.js projects • C# and F# for Mono projects on Linux and macOS • C and C++ on Windows, Linux and macOS • Python with Python plug-in⁸ installed • PHP with XDebug and PHP Debug plug-in⁸ installed

Figura 20: Lenguajes soportados por VSCode [10]

VSCode posee multitud de extensiones, entre ellas la de Python necesaria para el desarrollo del proyecto. Para instalar una extensión (ver Figura 21) hay que dirigirse a la pestaña *Extensions* y usar el buscador, una vez encontrada solo hay que instalarla.



Figura 21: Instalación de extensiones

El uso en el proyecto de esta herramienta creo que está justificado, ya que ofrece un entorno de desarrollo bastante amigable para el lenguaje que se usará. Además de traer un compilador y depurador incluidos, lo cual ahorra tiempo a la hora de testear ciertas funciones.

2.2.3. Despliegue web: XAMPP

XAMPP (Apache + MariaDB + PHP + Perl) es uno de los entornos de desarrollo PHP más populares que existen. Es sencillo de instalar y convierte tu ordenador en un servidor local [13].

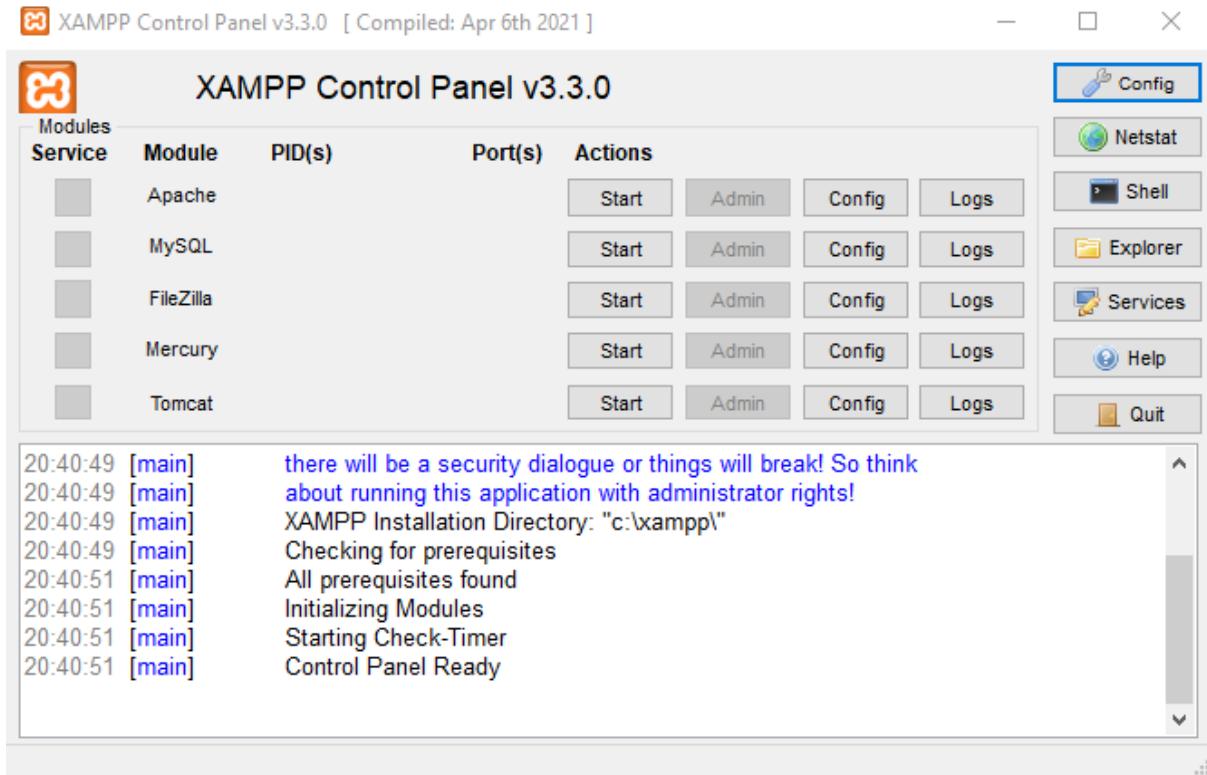


Figura 22: Interfaz de XAMPP

XAMPP es una distribución de Apache gratuita, sencilla de instalar y usar (ver Figura 22). El funcionamiento principal de Apache es dar un servicio web a cualquier usuario interesado en visualizar los ficheros alojados en su propia máquina. Por ejemplo, cuando un usuario accede desde su navegador a algún fichero alojado en nuestro servidor, esta petición le llegará a Apache que mediante HTTP se encargará de darle acceso de forma segura.

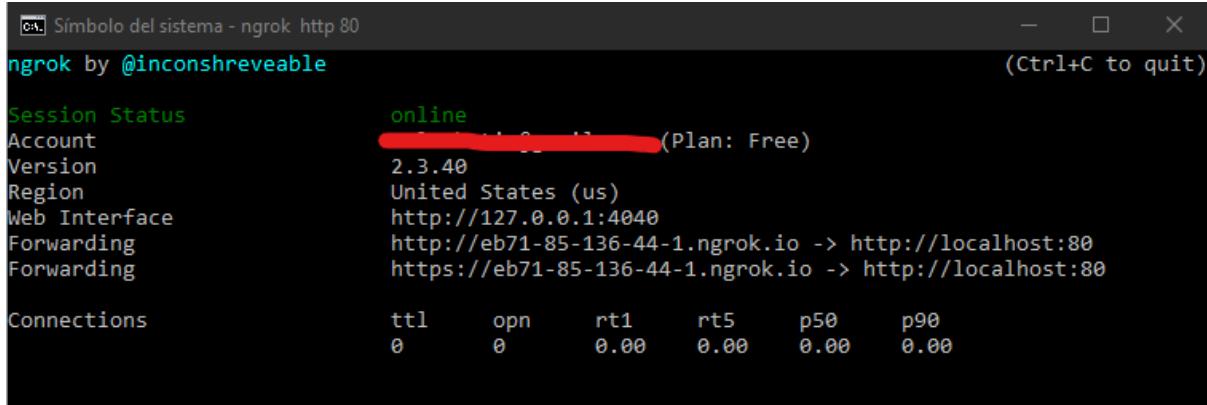
Apache tiene muchas ventajas, la principal es que su código es abierto y gratuito, y es actualizado regularmente. Al tener una estructura basada en módulos, esta nos permite activar y desactivar funcionalidades a nuestro gusto. El uso abusivo de módulos puede generar problemas de seguridad. Otro problema que presenta es la estabilidad cuando hay muchas conexiones concurrentes (sobre unas 100000).

Este proyecto hace uso de un servidor local necesario para alojar chatbots y poder hacer test o demos (ver sección 2.2.4 de esta memoria). Con XAMPP podemos activar el servidor apache con un clic (ver figura 22).

2.2.4. Despliegue web: NGROK

NGROK [14] nos ofrece la posibilidad de integrarnos con aplicaciones que funcionan en la nube. Si estas hacen llamadas a servicios, directamente o vía webhooks, es necesario desplegar nuestro servicio en una URL accesible por una aplicación externa.

Con NGROK podemos generar una URL de forma dinámica. Esta crea un túnel con nuestro servidor web usando el puerto que hayamos proporcionado (en nuestro caso Apache usa el puerto 80). Para generar la dirección solo hay que ejecutar un comando, lo cual hace bastante sencillo el uso de esta herramienta (el resultado es mostrado en la Figura 23).



```
Símbolo del sistema - ngrok http 80
ngrok by @inconshreveable
Session Status      online
Account             [REDACTED] (Plan: Free)
Version             2.3.40
Region              United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          http://eb71-85-136-44-1.ngrok.io -> http://localhost:80
Forwarding          https://eb71-85-136-44-1.ngrok.io -> http://localhost:80

Connections        ttl     opn     rt1     rt5     p50     p90
                    0       0     0.00    0.00    0.00    0.00
```

Figura 23: Ejemplo de uso de NGROK con el comando ngrok http 80

Podemos hacer testeos en dispositivos móviles o tabletas para probar nuestro sitio web de forma rápida, al tener ssh se puede acceder de forma segura. También es posible mostrar el estado de nuestra aplicación al cliente por medio de demos. Estas son las principales ventajas que nos da el uso de NGROK [15].

2.2.5. Interfaz gráfica: PySimpleGUI

PySimpleGUI [16] es una librería de Python que nos da la opción a los programadores de crear interfaces de una forma fácil y sencilla. El propósito principal de esta herramienta es ser sencilla de usar y personalizar.

Las ventanas de esta interfaz son similares a las que usan otras aplicaciones, pero el código usado se reduce en más de un 50%. Sus características se pueden dividir en llamadas de alto nivel que permiten realizar operaciones de entrada/salida en una sola llamada de función, y en la capacidad de crear GUIs personalizada. Permite incluso compactar más el código haciendo uso de atajos para los nombres de los elementos.

En la Figura 24 podemos ver cómo se crea el código para mostrar el característico “Hola Mundo!”, y en la Figura 25 en resultado [17].

```

Python

# hello_world.py

import PySimpleGUI as sg

sg.Window(title="Hello World", layout=[[[]], margins=(100, 50)].read()

```

Figura 24: Código de “Hola mundo!” usando PySimpleGUI [17]



Figura 25: Ventana de “Hola mundo!” creada con PySimpleGUI [17]

Con los avances en el proyecto se hizo necesaria la implementación de una pequeña interfaz para el programa. Los puntos expuestos anteriormente hicieron que me decantara por esta opción y no otras (ver punto 2.5.3 de esta memoria).

2.2.6. Conectividad: Telegram Bot API

Telegram es una plataforma de mensajería instantánea, enfocada también al envío de archivos y a la comunicación en masa. El servicio ofrece desde funcionalidades para realizar charlas entre usuarios, grupos, llamadas y conferencias, envíos multimedia, canales de difusión, bots y más. La parte que nos interesa para el proyecto es esta última.

Telegram Bot API [18], es el nombre de la interfaz de programación orientada a bots. El mecanismo usa MTProto (protocolo de transporte de datos usado por Telegram: gestiona, cifra y transporta mensajería).

Para crear un bot es necesario usar un kit de desarrollo. Para ello, el bot padre (conocido como @botfather) es la herramienta que administra la creación, tókenes, alias y permisos. La implementación es gratuita exceptuando la opción de entablar pedidos que está limitada a empresas de pago [19].

Para integrarlo con Rasa necesitaremos hacer uso de BotFather, para crear nuestro bot en Telegram. Después debemos ir al fichero credentials.yml en los ficheros de nuestro chatbot de Rasa e incluir los datos que Bot Father nos ha proporcionado.

```

telegram:
  access_token: "490161424:AAGlRxinBRtKGb21_rLOEMtDFZMXB16EC0o"
  verify: "your_bot"
  webhook_url: "https://your_url.com/webhooks/telegram/webhook"

```

Figura 26: Ejemplo credentials.yml con los datos proporcionados por BotFather [18]

En la Figura 26 se pueden observar:

- access token: son los permisos de acceso al bot
- verify: se refiere al nombre del bot en sí
- webhook_url: dirección web donde está alojado en chatbot

Hay bastantes componentes soportados por la Telegram API de Rasa, entre ellos destacan: botones, audio, vídeo, fotos y documentos.

El uso de esta herramienta viene impuesta por la propia Rasa, ya que esa es la única forma de conectar estos dos servicios. Es una manera sencilla ya que con proporcionar los datos requeridos y ejecutar nuestro bot, ya lo tenemos activo en Telegram.

2.2.7. Extra: Minidom

Minidom [20] es una librería en Python que implementa la interfaz Document Object Model (Modelo de objetos del documento). Está destinada a ser más simple que una implementación completa de DOM y también más pequeña.

Las aplicaciones DOM suelen comenzar analizando un fichero XML. Mediante el uso de las funciones “parse()” o “parseString()” conecta el analizador sintáctico y lo convierte en un árbol DOM. De este árbol son accesibles todos los elementos de manera muy sencilla con solo una llamada a una función (ver Figura 27 y 28).

```

from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml') # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource) # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')

```

Figura 27: Ejemplo de uso de Minidom [20]

```

expertise = doc.getElementsByTagName("expertise")
print ("%d expertise:" % expertise.length)
for skill in expertise:
    print_(skill.getAttribute("name"))

```

Figura 28: Ejemplo extracción de etiquetas con Minidom [20]

En las primeras etapas del proyecto se usaba una plantilla de entrada en formato XML, por lo tanto era necesario hacer uso de esta librería para extraer los datos.

2.3. ALGORITMOS USADOS PARA EL CÁLCULO DE LA SIMILITUD SEMÁNTICA

Este proyecto tiene una parte fundamental relativa al uso de algoritmos para medir la similitud semántica entre las frases que proporcionen los usuarios al chatbot. En esta sección se hará una breve explicación de los algoritmos que se han integrado.

2.3.1. Similitud coseno

La similitud coseno mide la similitud existente entre dos muestras. Esta medida calcula el ángulo del coseno comprendido entre ellas. Es una función trigonométrica que devuelve 1 si el ángulo es cero, es decir, si ambas muestras son iguales (ver Figura 29). Esta medida se emplea mayoritariamente en la búsqueda y recuperación de texto, y minería de datos [22].

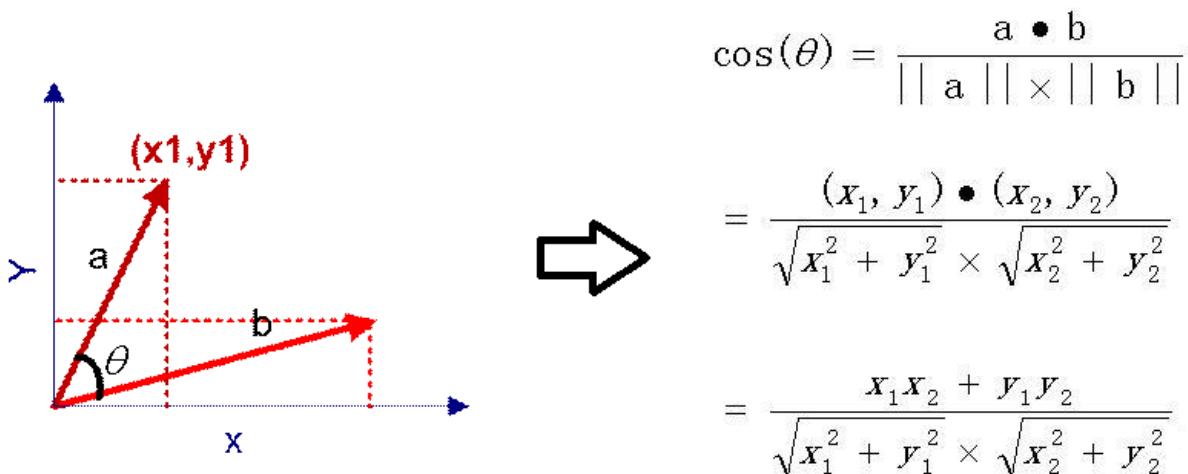


Figura 29: Cálculo del ángulo del coseno. Similitud coseno.

Coseno suave

El coseno suave considera la similitud de características en el modelo de espacio vectorial (MEV). Por ejemplo, palabras que son similares como “game” o “play” son distintas pero están relacionadas. Hay propuestas de varias fórmulas que consideran para el MEV nuevas características consistentes en asignar pesos por similitud. Si no hay similitud los resultados no varían de los calculados por la similitud coseno estándar [23].

$$\text{soft_cosine}_2(a, b) = \frac{\sum \sum_{i,j=1}^N a_{ij} b_{ij}}{\sqrt{\sum \sum_{i,j=1}^N a_{ij}^2} \sqrt{\sum \sum_{i,j=1}^N b_{ij}^2}}.$$

La fórmula anterior muestra cómo se asignan pesos a pares de características considerándolos como una sola. Esto provoca que el tamaño del vector que se usa para comparar aumente al igual que la complejidad de N a N², pero lo compensa obteniendo mejores resultados.

Similitud coseno en Python

En Python podemos usar la primera versión (Figura 29) por medio de la librería sklearn [24], que computa la similitud coseno normalizada entre dos muestras X e Y. Se puede implementar como se muestra en la Figura 30.

```
# coseno #
vectorizer = TfidfVectorizer()

t1 = vectorizer.fit_transform([text1]).toarray()
t2 = vectorizer.transform([text2]).toarray()
cos = cosine_similarity(t1, t2)
score = cos[0][0]
```

Figura 30: implementación de la similitud coseno en Python

2.3.2. Okapi BM25

BM25 es una función de ranking usada en recuperación de información basada en la bolsa de palabras. Esta función devuelve la similitud que hay entre un conjunto de términos y los documentos comparados.

Desarrollada por un grupo de investigación mientras experimentaban con el sistema de recuperación Okapi. Se basa en un sistema de ecuaciones que asignan pesos a términos, este fue derivando hacia un modelo probabilístico basado en una distribución de 2-Poisson. Este modelo provee un marco de trabajo para la comparación entre dos documentos de texto [25].

$$\text{score}(D, Q) = \sum_{i=1}^n \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

La fórmula obtiene la puntuación de una consulta Q en un documento D. Donde:

- **IDF(qi)** es el peso IDF (ver figura 33) del término q_i .
- **f(qi, D)** es la frecuencia de aparición del término q_i en el documento D.
- **|D|** es la longitud del documento D en palabras,
- **avgdl** es la longitud promedio del documento en la colección.
- **k1** y **b** son valores libres, por lo general los valores suelen ser:
 - **k1** → [1.2, 2]
 - **b** → 0.75

IDF es la frecuencia inversa del término en el documento. La fórmula mostrada en la figura 25 da más peso a los términos que ocurren con menos frecuencia.

$$\text{IDF}(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right)$$

Donde **N** es el número de documentos y **n(qi)** el número de documentos que contienen el término q_i

BM25 en Python

Para su implementación en Python (ver Figura 31) he usado la librería *rank_bm25* [26], que contiene varias implementaciones de la función BM25.

```
tokenized_corpus = []
tokenized_corpus[0] = text1.split()
tokenized_query = text2.split()

bm25 = BM25Okapi(tokenized_corpus)
doc_scores = bm25.get_scores(tokenized_query)

score = abs(doc_scores[0])

result += "BM25 --> {}\n".format(score)

bm25 = BM25L(tokenized_corpus)
doc_scores = bm25.get_scores(tokenized_query)

score = abs(doc_scores[0])
```

Figura 31: Implementación de BM25 en Python

2.3.3. Longest Common Subsequence

Longest Common Subsequence o LCS trata de encontrar la subsecuencia común más larga dentro de un conjunto de secuencias dadas. Este problema es clásico en el campo de la ciencia computacional, es la base en programas de comparación de datos y también la utilidad diff (comparación de datos en un conjunto de ficheros), y tiene aplicaciones en lenguaje computacional y bioinformática. Un fórmula puede ser la siguiente:

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \cup x_i & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Un ejemplo de su funcionamiento sería: considerando las secuencias ABCD y ACBAD, consta de cinco secuencias con dos subsecuencias (AB, AC, AD, BD, CD), dos secuencias de 3 (ABD, ACD), y no hay más secuencias comunes. Entonces las secuencias más largas serían ABD y ACD.

	A	C	B	A	D
A	1	1	1	1	1
B	1	1	2	2	2
C	1	2	2	2	2
D	1	2	2	2	3

Tabla 1: Ejemplo de funcionamiento LCS 1

	A	C	B	A	D
A	1	1	1	1	1
B	1	1	2	2	2
C	1	2	2	2	2
D	1	2	2	2	3

Tabla 2: Ejemplo de funcionamiento LCS 2

En las Tablas 1 y 2 podemos ver como se conseguirían las cadenas ABD y ACD del ejemplo anterior siguiendo el método mostrado en la figura 35.

LCS en Python

Su implementación en Python se hace por medio de la función SequenceMatcher (ver Figura 32) de la librería difflib. Esta función busca la subsecuencia más larga contenida en dos secuencias que no contengan elementos no deseados [27].

Con esta función podemos saber el grado de similaridad entre dos secuencias gracias a una subfunción llamada *ratio()*. Su fórmula se define así: $\text{ratio} = 2.0 * M/T$, donde M se refiere a las coincidencias y T es el total de elementos en ambas secuencias [27].

```
# LCS #
score = SequenceMatcher(None, text1, text2).ratio()
```

Figura 32: Implementación de LCS en Python

2.3.4. Jaro-Winkler

La similitud Jaro utiliza la transposición de caracteres, por lo que solamente cuenta el número de caracteres coincidentes entre dos cadenas de texto y el número de transposiciones que son necesarias para llegar de una a otra [28].

$$j_{a,b} = \begin{cases} 0 & \text{si } m = 0, \\ \frac{1}{2} \left(\frac{m}{|a|} + \frac{m}{|b|} + \frac{m-t}{m} \right) & \text{si } m \neq 0 \end{cases}$$

donde:

- m → son los caracteres coincidentes en ambas cadenas.
- |a| → es la longitud de la cadena a.
- |b| → es la longitud de la cadena b.
- t → es el número de transposiciones para convertir una cadena en otra.

Winkler propuso modificar esta similitud para dar peso mayor a las cadenas que tengan prefijos comunes.

$$jk_{a,b} = j_{a,b} + lp(1 - j_{a,b})$$

En la mayoría de los casos los prefijos tienen más valor que los sufijos para identificar similitudes en las cadenas de texto [28].

La similitud Jaro-Winkler introduce una escala que otorga mejores calificaciones a cadenas que coinciden desde el principio. En la figura 38, esta escala se corresponde con la **p**, la **l** es la longitud del prefijo común al comienzo de las cadenas.

Jaro-Winkler en Python

En Python tenemos que hacer uso de la librería jellyfish [29]. Esta librería incluye más métodos además de el que se usa como: la distancia levenshtein, damerau, Hamming. La elección del método de Jaro-Winkler (ver Figura 33) se dió porque mostraba mejores resultados que los demás en la comparación de dos secuencias.

```
# Jaro-Winkler #
score = jellyfish.jaro_winkler_similarity(text1, text2)
```

Figura 33: Implementación de la similitud Jarp-Winkler en Python

2.3.5. Evaluación de algoritmos

Para finalizar esta sección he querido comparar el rendimiento de los algoritmos al comparar dos cadenas de texto. Las Tablas 3,4,5 y 6 hacen una comparación de estos métodos usando dos cadenas de texto.

Cadena 1	Mi nombre es Manuel	
Cadena 2	Manuel es el nombre	
Resultados		
1º	Coseno	86%
2º	BM25	82%
3º	Jaro-Winkler	61%
4º	LCS	42%

Tabla 3: Comparación de algoritmos 1

Cadena 1	Mi llamo Manuel	
Cadena 2	Manuel me llaman	
Resultados		
1º	Coseno	81%
2º	Jaro-Winkler	78%
3º	LCS	58%
4º	BM25	54%

Tabla 4: Comparación de algoritmos 2

Cadena 1	Soy conocido por Manuel	
Cadena 2	Me conocen como Manuel	
Resultados		
1º	Jaro-Winkler	68%
2º	LCS	66%
3º	Coseno	50%
4º	BM25	27%

Tabla 5: Comparación de algoritmos 3

Cadena 1	Yo soy Manuel	
Cadena 2	Mi nombre es Manuel	
Resultados		
1º	Jaro-Winkler	63%
2º	LCS	62%
3º	Coseno	57%
4º	BM25	27%

Tabla 6: Comparación de algoritmos 4

Viendo los resultados obtenidos podemos sacar varias conclusiones:

1. Si usamos palabras parecidas, los métodos coseno y BM25 obtienen mejores resultados. Por lo tanto, si usamos expresiones similares debemos usar estos algoritmos.
2. En cambio, si es al contrario logran los peores resultados, siendo Jaro-Winkler y LCS los que tienen mejores resultados.
3. La similitud Jaro-Winkler tiene mejores resultados si hacemos la media y, en consecuencia, creo que es el algoritmo recomendado para usar en la comparación de ejemplos del chatbot.

2.4. MODELOS DE DIÁLOGO BASADOS EN ÁRBOLES DE DECISIÓN

En la última parte de este proyecto se ha implementado la opción de crear un chatbot que siga una estructura basada en un árbol de decisión. Estos son modelos de predicción que siguen una probabilidad para tomar decisiones.

2.4.1. Árboles de decisión

Un árbol de decisión es un modelo de predicción que es usado en muchos ámbitos. Con un conjunto de datos se pueden construir diagramas lógicos similares a sistemas basados en reglas. Estos diagramas representan una serie de condiciones que ocurren de forma sucesiva [30]. La Figura 34 muestra un ejemplo de las condiciones necesarias para practicar tenis, contando con el tiempo, la humedad y el viento como factores determinantes.

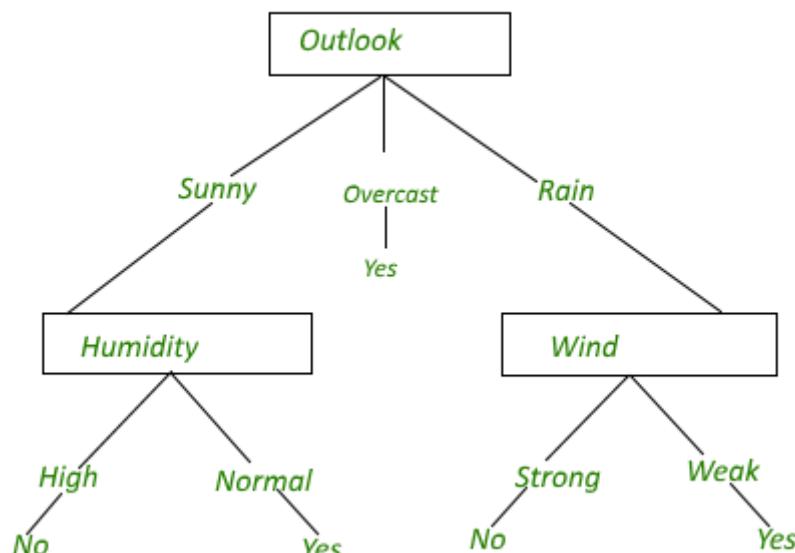


Figura 34: Ejemplo de árbol de decisión para jugar al tenis

Los árboles de decisión están formados por los siguientes elementos:

- **Nodos** → representan el momento en el que se debe tomar una decisión.
- **Flechas** → son las uniones entre los nodos.
- **Etiquetas** → se encuentran entre cada nodo y flecha, representan las acciones.
- **Nodo hoja** → representa el último nodo de una rama. Se llega a él a través de las decisiones tomadas desde el comienzo del árbol.
- **Rama** → camino desde el nodo inicial hasta el nodo hoja. Un árbol de decisión suele tener varias ramas.

Cuando desarrollamos árboles de decisión solemos encontrarnos con ciertos conceptos:

- **Costo** → corresponde con el coste de medir el valor de una propiedad y el coste de clasificar erróneamente. Es decir, decir que el objeto pertenece a la clase X, cuando en realidad corresponde a la clase Y.
- **Sobreajuste/Overfitting** → : se produce cuando hay pocos datos de entrenamiento o contienen incoherencias.
- **Poda/Prunning** → consiste en la eliminación de ramas o nodos. Estos son transformados en hojas y se les da el valor más común considerando los datos de entrenamiento.
- **Validación cruzada** → Es el proceso de construir un árbol con la mayoría de los datos y usar lo que queda para probar la precisión del árbol.

2.5. PLATAFORMAS PARA EL DESARROLLO DE SISTEMAS CONVERSACIONALES

En esta sección haré una pequeña recopilación de aplicaciones o herramientas similares a las de este proyecto que no han sido usadas por: o son de pago, o hay otras más simples o fáciles de usar.

2.5.1. Dialogflow

Dialogflow es una plataforma dedicada al entendimiento del lenguaje natural perteneciente a Google. Incluye una interfaz amigable (ver Figura 35) con la que cualquier persona sin experiencia en programación puede crear un bot sin problemas. Dialogflow no es de código abierto como Rasa y tiene una versión de pago, pero ofrece la posibilidad de usar una versión gratuita, lo que es ideal para hacer pruebas o para pequeñas empresas. También ofrece una amplia gama de APIs tales como Slack, LINE, Telegram, Hangouts, Skype, ...

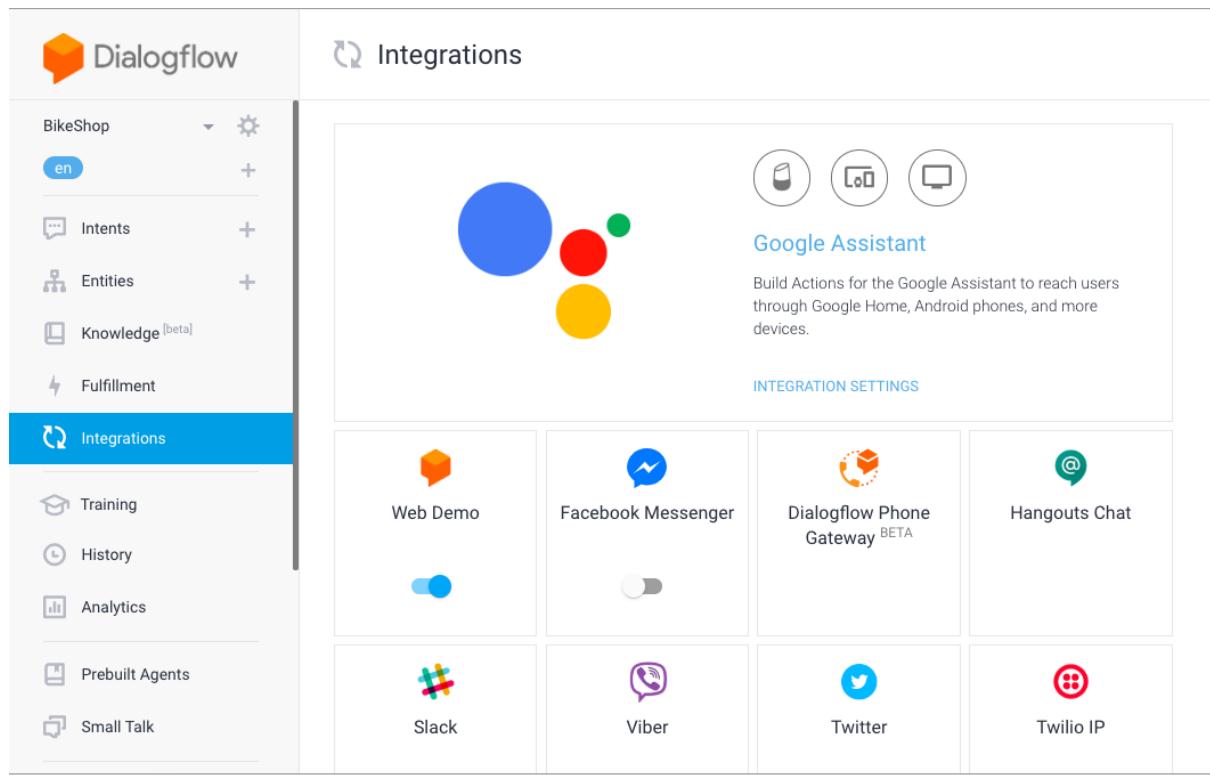


Figura 35: interfaz principal de Dialogflow [33]

Los bots, aquí llamados agentes, contienen intents. La creación de estos es muy sencilla, basta con ir a la pestaña de intents, asignarle un nombre y escribir ejemplos de respuestas que puede dar el usuario (ver Figura 36). El NLU (natural language understanding) es capaz de identificar entidades en las oraciones de muestra, pudiendo hacer que sean obligatorias marcándolo en las opciones. En el caso de no encontrar las entidades requeridas en un intent, puedes programar que pregunte por ellas [33].

The screenshot shows the Dialogflow interface for creating an intent. The intent is named "weather". In the "User says" section, there is a search bar and several examples: "Weather forecast in San Francisco tomorrow", "Weather for tomorrow", "what is the weather today", and "weather forecast". Below this, a table lists parameters with their resolved values:

PARAMETER NAME	ENTITY	RESOLVED VALUE
geo-city	@sys.geo-city	San Francisco
date	@sys.date	tomorrow

Figura 36: Creación de intents en Dialogflow [33]

De la misma manera, podemos crear respuestas del bot en el intent como se muestra en la Figura 37. Solo hay que navegar en las opciones de intent y agregarle tantas respuestas que creamos necesarias [33].

The screenshot shows the "Text response" section for the "weather" intent. It contains four responses:

- Sorry I don't know the weather
- I'm not sure about the weather on \$date
- I don't know the weather for \$date in \$g
- Enter a text response variant... \$geo-city

A cursor is hovering over the fourth response, which has the placeholder "\$geo-city" typed into its input field.

Figura 37: Editando respuestas en Dialogflow [33]

2.5.2. Amazon Lex

Amazon Lex es el servicio provisto por la compañía Amazon para construir interfaces conversacionales. Se sirve del asistente virtual Alexa. Amazon apuesta por una gran cantidad de actividades realizables a través de esta tecnología, donde los chatbots son un sólido punto de entrada para esta tecnología.

Amazon Lex habilita la opción a los desarrolladores de crear bots que convierten el texto introducido por un usuario en intenciones reconocibles, dando como resultado los bots más conversacionales y sofisticados del mercado [34].

Esta aplicación es de pago, variando este por opciones. Por ejemplo: para un bot que procesa 8000 solicitudes de voz y 2000 de texto, el precio sería de 33,5\$ y para conversaciones en streaming el valor del mismo bot sube a 56\$.

En cuanto a funcionamiento, Amazon Lex proporciona tecnologías de reconocimiento del habla y comprensión del lenguaje natural, capaz de aprender las distintas formas en las que los usuarios suelen expresarse. Se admite la gestión del contexto de forma nativa, de forma que puede mantener conversaciones con turno o multturnos, es decir, solicitar la información requerida al usuario para completar su función [35] (Ejemplo: reserva de hotel mostrado en la Figura 38).

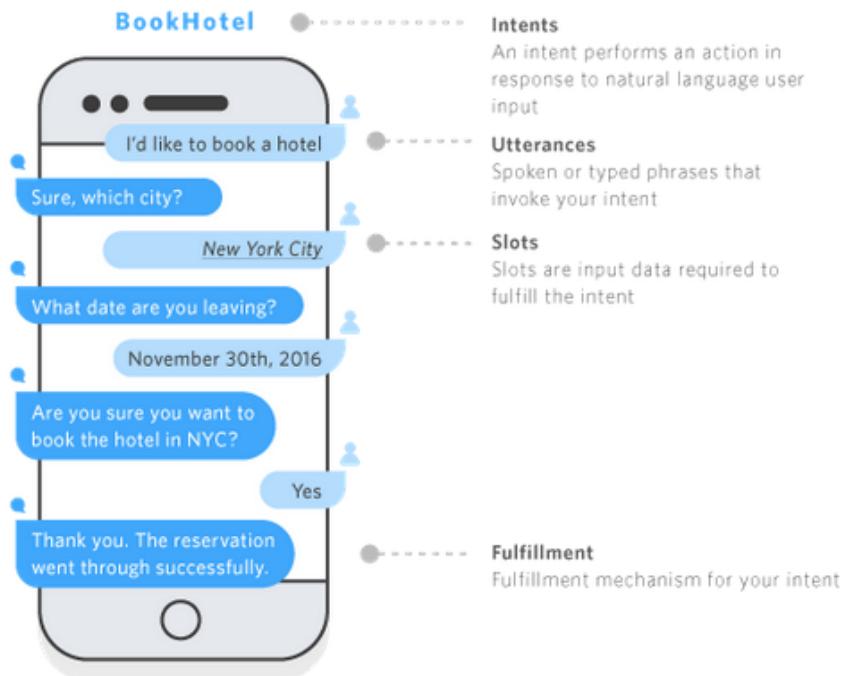


Figura 38: Ejemplo de conversación con Amazon Lex [35]

2.5.3. TKinter

El paquete tkinter es la interfaz por defecto de Python para el kit de herramientas de GUI Tk. Tanto Tk como tkinter están disponibles para la mayoría de plataformas Unix, así como en sistemas Windows.

Tkinter soporta un amplio rango de versiones TCL/TK (toolkit para el desarrollo de interfaces en Python). Su arquitectura más bien consiste en un pocos módulos cada uno separados por funcionalidades [36]:

- TCL: Lenguaje de programación interpretado y dinámico, similar a Python. Su uso normal va ligado a las aplicaciones en C como motor de secuencias de comandos o como interfaz hacia el kit de herramientas tk.
- TK: Es un paquete implementado en C, el cual agrega comandos personalizados capaces de manipular widgets en la GUI.
- TTK: ampliación del paquete tk, con más widgets que proporcionan una mejor experiencia.

En la Figura 39 se muestra el código para crear una ventana con el texto “Hola Mundo!”, que se puede ver en la Figura 40.

```
from tkinter import *

root = Tk()
a = Label(root, text ="Hello World")
a.pack()

root.mainloop()
```

Figura 39: Código de Hello World en tkinter [37]

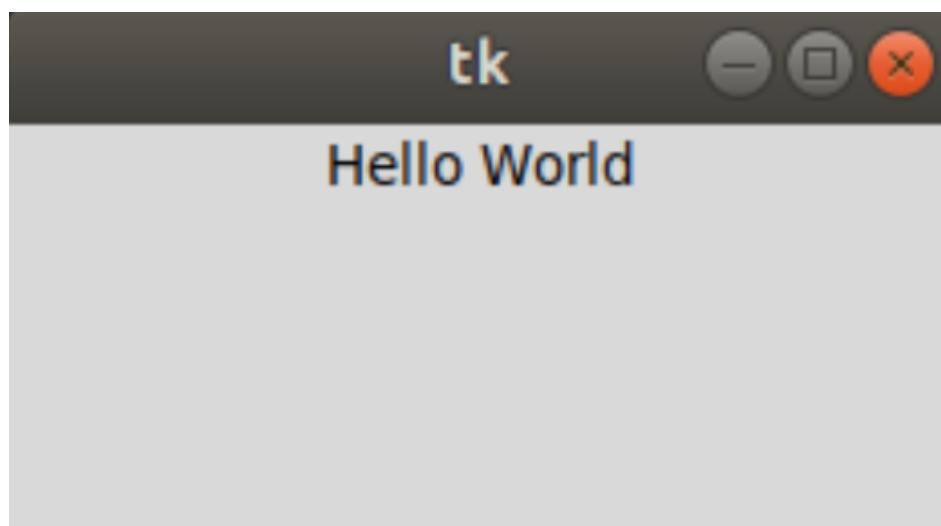


Figura 40: Ventana de Hello World en Tkinter [37]

2.6. Conclusiones sobre las plataformas

Como hemos podido observar, hay más aplicaciones de las que he indicado disponibles para el desarrollo de un chatbot. Muchas de ellas incluyen GUIs, pero son de pago. Otras incluyen versiones gratuitas pero su código es privado.

Tanto Google con su Dialogflow como Amazon con su Amazon Lex, poseen potentes herramientas para creación de chatbots. Dialogflow cuenta con una interfaz limpia y de fácil uso y Amazon Lex cuenta con el soporte del asistente virtual Alexa que le da mucho atractivo.

Rasa nos presenta la opción de “hacer lo mismo” en una plataforma open source, fácil de usar y con mucho potencial para lograr los mismos objetivos. Además cuenta con una documentación actualizada y muy detallada que ayuda a resolver las posibles dudas que puedan surgir. Otra ventaja en la que iguala a sus competidores es en la cantidad de servicios con los que se puede conectar, de entre los cuales voy a usar los servicios de despliegue web de NGROK y de comunicación la API de Telegram.

En la parte de la interfaz, he intentado escoger una librería fácil de usar y de comprender. Por eso me decanté por PySimpleGUI respecto a otras, ya que su código es simple y ocupa mucho menos que Tkinter.

Respecto a los algoritmos usados, la idea de incluir algoritmos surgió a la hora de añadir ejemplos al chatbot automáticamente, es decir, un usuario hace una consulta y el sistema juzga si se parece o no a las que tiene, si no se parece la añade a su lista interna. Los algoritmos utilizados en el proyecto son algunos de los más conocidos en el ámbito de recuperación de información y comparación de muestras, podría haber incluido más pero creo que no hacía falta ya que el proyecto trata de hacer un chatbot.

Capítulo 3: GESTIÓN DEL PROYECTO

En este capítulo se explicarán los métodos y procesos utilizados en la creación del proyecto y los objetivos fijados. Lo principal es el establecimiento de una metodología de trabajo adecuada que controle los tiempos, seguido de los requisitos y funciones ofrecidos por el sistema. Por último, se explicará el proceso seguido en el desarrollo de la aplicación.

3.1. PLANIFICACIÓN Y PRESUPUESTO

3.1.1. Metodología

En esta sección se explican las fases por la que ha pasado el desarrollo del proyecto, desde la fase inicial buscando información sobre la plataforma Rasa, hasta la fase final donde se han añadido pequeñas correcciones al chatbot y detección de errores sobre las plantillas usadas.

La duración del proyecto ha sido de 282 días en total, en este tiempo se dividió la planificación en cuatro fases diferenciadas. Cada una de ellas contiene subsecciones en las cuales comento las tareas realizadas en ellas como podemos ver en la Figura 41.

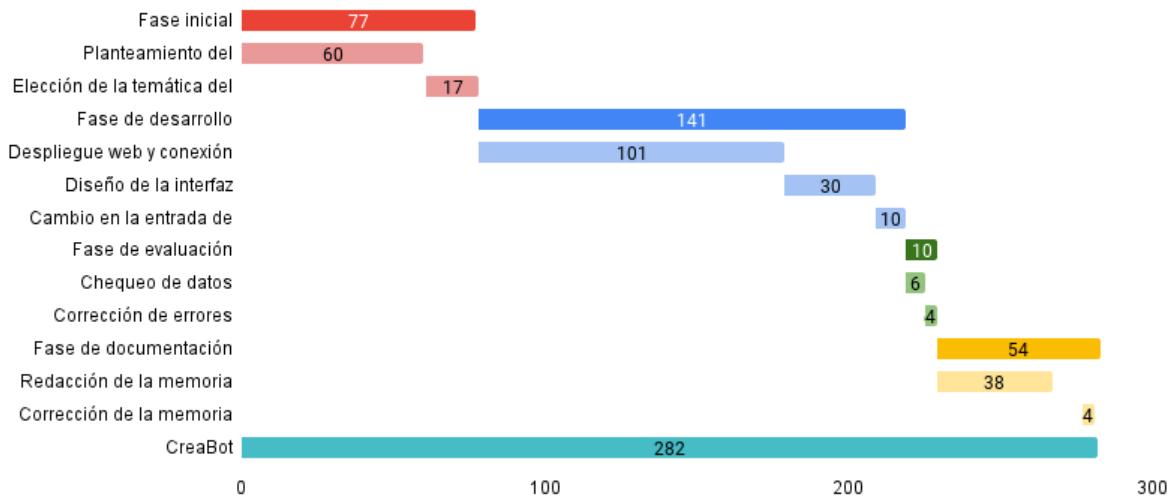


Figura 41: Diagrama de las fases del proyecto

3.1.1.1. Fase inicial

La fase inicial es la de mayor importancia ya que representa los cimientos para construir el resto del proyecto. En esta fase se presenta la idea del proyecto, al ámbito al cual está dirigido, las herramientas y las tecnologías que se van a usar para la obtención del objetivo.

El inicio de esta fase fué el 16 de noviembre de 2021, día en el que tuve la primera reunión con mi tutor. A partir de esta primera reunión empezamos a quedar cada dos o tres semanas, cuando los objetivos de la anterior reunión estaban cumplidos.

Planteamiento del problema

Tras la primera reunión, mi tutor me encomendó la tarea de familiarizarme con la plataforma elegida para realizar el proyecto: Rasa. Con la instalación surgieron los primeros problemas ya que Rasa requiere las versiones Python 3.7 o 3.8 y un entorno propio. Solucionados estos, comencé con el estudio de los elementos y ficheros que posee Rasa, después empecé a experimentar con el código, crear pequeños chatbots que pedían datos, desarrollar funciones custom para validar datos, hasta a probar el módulo de aprendizaje que posee Rasa.

Elección de la temática del proyecto

En las siguientes reuniones se empezó a discutir sobre la temática que debería tener el chatbot. Mi tutor me proveyó con unos ejemplos que habían desarrollado con anterioridad, como por ejemplo un chatbot dedicado a hacer preguntas a usuario que teman haberse contagiado de Covid19, o ejemplos de chatbot para gestionar reservas de viaje en tren.

Algunas de mis propuestas, no iban muy alejadas de los sistemas expuestos anteriormente, por ejemplo: un sistema para encontrar coches de segunda mano, donde el chatbot preguntaba por la localización, marca o modelo y devolvía una lista de las opciones dadas por el usuario; otro muy parecido para buscar piso, o para encontrar un restaurante, o un vuelo. Todos estos chatbots tenían un factor común, eran casi el mismo tipo de chatbot en el que solo cambiaba el tópico sobre el que hace las preguntas.

Entonces fue cuando mi tutor propuso la idea de hacer un programa que creara todos estos tipos de chatbots automáticamente, solo con el uso de una plantilla. Esta plantilla debería contener las preguntas y todo lo necesario para crear un chatbot funcional sobre el tema que sea.

De esta manera se eligió la temática del proyecto y se propusieron los objetivos de que el programa pudiese crear varios tipos de chatbot: el de respuesta a preguntas frecuentes, el de modelo de diálogo basado en una secuencia de pasos y el de estructura en árbol de decisión.

3.1.1.2. Fase de desarrollo

Esta fase se inició en febrero. Esta ha sido la fase más amplia del proyecto. Durante la cuál mantenía reuniones con mi tutor cada dos o tres semanas, donde se trataban los temas discutidos en las anteriores reuniones y se proponían nuevas ideas y objetivos para el proyecto, o se notificaron errores en el diseño y posibles soluciones.

En esta etapa se fueron añadiendo conceptos y funcionalidades que no estaban en el esquema inicial del proyecto. Además de discutir nuevas funciones con mi tutor, yo también iba añadiendo funciones o características que consideraba importantes. Un ejemplo es la inclusión de algoritmos para añadir ejemplos al chatbot o incluir una interfaz gráfica al programa.

Despliegue web y conexión con Telegram

Acabado el primer tipo de chatbot era necesario desarrollar un sistema para poder probarlo, ya que el que usa Rasa por línea de comandos es bastante lento y no es adecuado para el uso por parte de un cliente. Se diseñó una página web que contenía un webchat con el que se podía hablar con el chatbot.

No hubo mucho problema para desplegarlo en un entorno web, ya que con tener un servidor apache local, la herramienta NGROK puede crear un enlace público para el chatbot accesible desde cualquier parte. La parte de la conexión con Telegram fue igual de sencilla, gracias al diseño de Rasa, se pudo conectar este servicio con mi chatbot sin dificultades.

Diseño de la interfaz

Su desarrollo comenzó a la par de las últimas etapas del proyecto. Al aumentar los parámetros que usaba el programa se hacía cada vez más necesario el uso de una interfaz donde poder controlar el valor de estos.

Tras mostrar un prototipo de interfaz en una reunión a mi tutor, él me animó a seguir con esta idea. Le añadí más funcionalidades a la interfaz, como la de cargar en un panel la plantilla y, poder editar y guardar los nuevos datos. Cada versión daba más posibilidades de edición sobre el chatbot: elegir el tipo de chatbot, el algoritmo usado, etc. Sobre el diseño de colores elegidos, la librería trae varios temas predefinidos, y elegí una opción que me gustaba más y parecía más seria.

Cambio en la entrada de datos

Durante casi todo el proyecto la entrada de datos correspondía con un fichero XML. El formato de esta plantilla cambió varias veces durante el desarrollo para adaptarse a las necesidades del tipo de chatbot. Intenté hacerlo lo más sencillo posible de cara a un usuario normal sin conocimientos en programación.

El cambio se produjo a partir de una reunión en la mitad de junio, donde mi tutor me comentó la posibilidad de dejar atrás el modelo que estaba usando y usar un fichero de

texto normal. Se me propuso un formato bastante específico, usando separadores, paréntesis y llaves que fué adaptado en mi código. Costó alrededor de unos cinco días la transformación del modo de entrada del chatbot, ya que había que cambiar todo el código donde extraía de XML al nuevo formato.

3.1.1.3. Fase de evaluación

Chequeo de datos

Este fue el último añadido al programa. Es una función que fué añadida tras comentar en una reunión con mi tutor un error que se producía al crear los chatbots si las plantillas estaban mal escritas.

La función chequea línea por línea y decide según el tipo de chatbot si la línea es correcta o no. Por ejemplo, el único tipo de chatbot que usa entidades es el de secuencia de pasos, por lo tanto si las detecta en otro tipo de chatbot saltaría un error y no dejaría crear el chatbot. Y así con multitud de parámetros más. Al detectar un error en la plantilla lo anota en una variable que es consultada al final de la función, si esta variable contiene algo no deja crear el chatbot.

Corrección de errores

Un error muy común en los chatbots de preguntas frecuentes es que no responda bien a lo que preguntamos, eso es debido a la falta de ejemplos. Suministrando suficientes ejemplos el chatbot no tendrá problemas en contestar correctamente.

Para el chatbot basado en un árbol de decisión, suele ser habitual llegar a estado donde no se recuperé bien porque la respuesta no está contemplada en el chatbot o porque está mal expresada. Por eso la solución es añadir todas las posibles ramas y contemplar más ejemplos de respuestas que el árbol pueda tener.

3.1.1.4. Fase de documentación

Redacción de la memoria

Durante la realización de esta memoria, aproveché que había hecho una especie de diario durante el desarrollo del proyecto donde guardaba las referencias y los pasos que había seguido en la implementación de los distintos servicios de los que consta este proyecto.

Para la estructura de la memoria, fue necesario mantener un par de reuniones con mi tutor para que me orientará sobre ella, saber qué elementos incluir y cómo redactarlos.

Corrección de la memoria

Hecha la primera versión, esperé la corrección de mi tutor. Tras esto comentamos los cambios que había que hacer para terminar la versión final de la memoria.

3.1.2. Presupuesto

En esta sección se describen los distintos recursos hardware y software empleados en el desarrollo del proyecto, explicando su uso y el coste de estos.

3.1.2.1. Recursos hardware

Aquí se muestran los recursos hardware usados, tanto su uso en la Tabla 7 como su precio en la Tabla 8. Para calcular el precio usaré la vida útil de cada componente y lo multiplicaré por el tiempo de realización del proyecto.

Dispositivo	Modelo	Uso
Ordenador personal	Ryzen 5 3600 + 32GB RAM DDR4 3200MHz + 1050TI 4GB VRAM	Desarrollo y prueba de la aplicación
Smartphone	Realme 7	Prueba de la aplicación

Tabla 7: Recursos hardware utilizados

Modelo	Precio	Vida útil	Total
Ryzen 5 3600 + 32GB RAM DDR4 3200MHz + 1050TI 4GB VRAM	1200€	8 años	115.89€
Realme 7	230€	5 años	35.54€
Total			151.43€

Tabla 8: Presupuesto de los recursos hardware

3.1.2.2. Recursos software

En este apartado se describen los recursos hardware utilizados, tanto su uso (ver Tabla 9) como su precio (ver Tabla 10). El precio de estos recursos va ligado al precio de su licencia.

Herramienta	Uso
Windows 10	Sistema Operativo
Rasa Open Source	Plataforma para la creación de chatbots
Visual Studio Code	Desarrollo del código de la aplicación
XAMPP	Servidor apache
NGROK	Herramienta para el despliegue web
PySimpleGUI	Librería para crear interfaces en Python
Google Docs y Sheets	Elaboración del borrador de la memoria.
Microsoft Office Word	Procesador de texto usado para elaborar la versión final de la memoria
Google Draws	Diseño de diagramas, casos de uso y arquitectura del proyecto.

Tabla 9: Recursos software utilizados

Herramienta	Precio
Windows 10	145€
Rasa Open Source	0€
Visual Studio Code	0€
XAMPP	0€
NGROK	0€
PySimpleGUI	0€
Google Docs y Sheets	0€
Microsoft Office Word	149€
Total	294€

Tabla 10: Presupuesto de los recursos software

3.1.2.3. Recursos personales

Aquí se describirán los miembros del proyecto teniendo en cuenta el puesto, tiempo invertido en el proyecto contando las horas, precio por hora y el coste total (ver Tabla 11). Las horas realizadas por el tutor del proyecto se han calculado teniendo en cuenta una media de 40 minutos por reunión y han sido alrededor de 15 reuniones. Dan un total de 600 minutos que equivalen a 25 horas.

En cuanto al coste de un profesor de universidad, contando un salario bruto que ronda los 31500€ anuales es decir alrededor de 24500€ netos anuales contando con el IPRF de 2022. El coste por horas teniendo en cuenta un jornada laboral de 40 horas semanales, sabiendo que un mes tiene 4.34 semanas, asciende a 11.76€/h.

En el cálculo de mis horas, teniendo en cuenta los 282 días a una media de 2.5 horas al día, podemos calcular un total de 700 horas. El coste de un programador junior es de 1500€ al mes, si tenemos en cuenta que una jornada laboral consta de 8 horas, el precio por hora es de 8.33€/h.

Nombre	Puesto	Tiempo (H)	Coste (€)	Total
David Griol Barres	Profesor de universidad	25 h	11.76€/h	294€
Pablo Valenzuela Álvarez	Programador Junior	700 h	8.33€/h	5.381€
Total				6125€

Tabla 11: Presupuesto de los recursos personales

3.1.2.4. Coste total

Como conclusión del apartado, queda la suma total de todos los costes parciales del proyecto, el cual asciende a 6570.43€

Tipo	Coste
Coste del hardware	151.43€
Coste del software	294€
Coste del personal	6125€
Total	6570.43€

Tabla 12: Coste total del proyecto

Capítulo 4: DESARROLLO DEL PROYECTO

Este capítulo va dedicado a comentar todos los aspectos relacionados con la implementación y la programación del proyecto. En las subsecciones se explicarán las distintas fases por las que ha ido transcurriendo el desarrollo, contando desde el diseño y funcionamiento de las interfaces, la elaboración de los tipos de chatbot que soporta el programa, y la detección de errores al usar la aplicación.

4.1. INTRODUCCIÓN

Comenzamos con una pequeña introducción donde explicaré qué son los tres tipos de chatbot que permite hacer el programa (ver Figura 42) y una breve explicación de cómo se han implementado que será ampliada en la sección 4.3 de esta memoria.

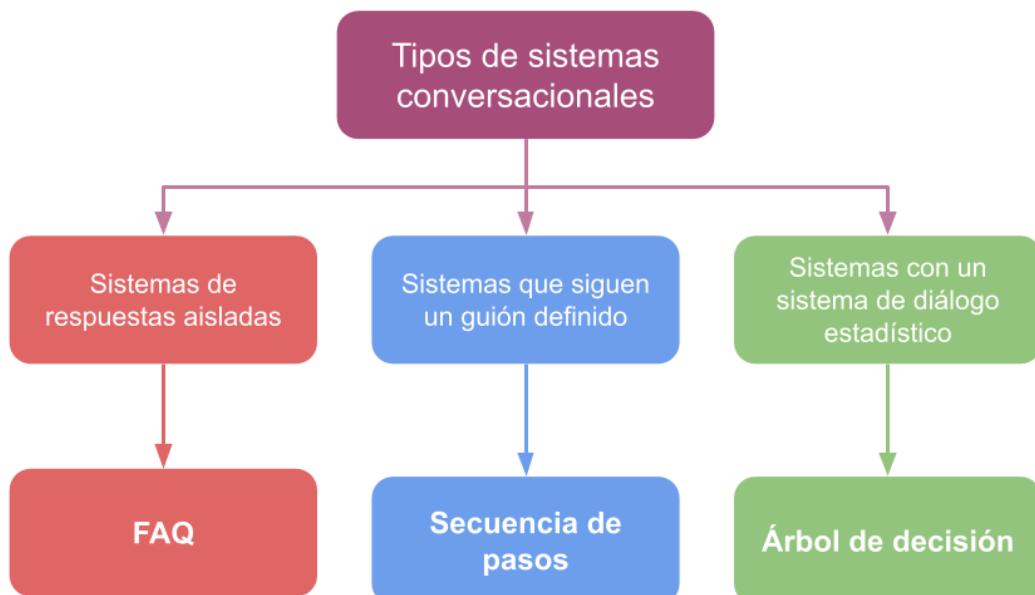


Figura 42: Tipos de sistemas conversacionales del proyecto

Preguntas frecuentes

Corresponde con los sistemas que responden preguntas sueltas y que no tienen relación entre ellas en la mayoría de los casos. Suele ser habitual ver en las páginas web de las grandes superficies secciones llamadas FAQs o preguntas frecuentes, donde hay una batería de preguntas con sus respectivas respuestas.

La implementación en este caso es sencilla, al ser un sistema que solo admite una respuesta a un tipo de pregunta, nuestro programa solo tiene que identificar el intent con la pregunta y responder. El formato de las historias como se ve la Figura 50, es muy simple y a cada pregunta (intent) le asigna una respuesta (action).



Figura 43: Formato de las stories en el chatbot FAQ

Por lo tanto la plantilla para hacer este chatbot debe ofrecer en cada línea un intent con preguntas y asociadas, y nuestro programa debe ser capaz de extraerlas.

Secuencia de pasos

Estos sistemas siguen un guión fijo de preguntas, que son siempre las mismas o parecidas. Se suelen ver este tipo de chatbots en sistemas de reserva de viajes o similares, donde el sistema los datos en un orden concreto, por ejemplo, para reservar un vuelo necesita lugar y destino u otros datos más como fechas.

Para el desarrollo de este tipo de chatbot, es necesario guardar la información que el usuario proporcione. Necesitamos hacer uso de entidades y slots (ver sección 2.2.1.3 y Figura 13). La plantilla debe contener las entidades necesarias y las respuestas relacionadas a estos. Por ejemplo, si una entidad es nombre, las respuestas deberían ser “mi nombre es tal” o “me llamo tal” como se muestra en el ejemplo de la Figura 51.

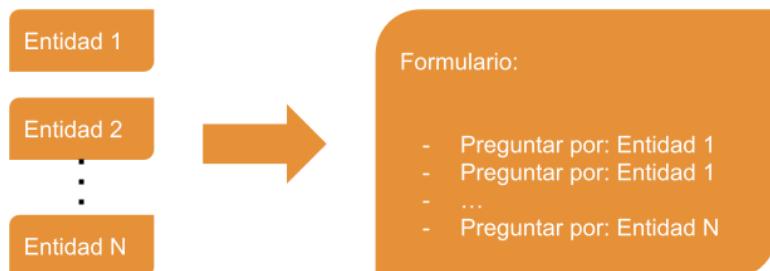


Figura 44: Diagrama de diseño de un formulario en Rasa

El código del programa está diseñado para identificar las entidades y registrar una pregunta automática para cada una de ellas. No son necesarias historias ni preguntas porque este tipo de chatbot activa de manera automática las preguntas de las entidades por medio de un formulario de Rasa (ver sección 2.2.1.3 y Figura 13). El propio código del programa se encarga de llenar ese formulario con todas las entidades proporcionadas en la plantilla.

Árbol de decisión

Este es el modelo más complejo y laborioso. Se basa en un sistema en el que hay que tomar decisiones y cada una de ellas implica nuevos caminos y más profundos. Hay bastantes maneras de implementar un chatbot basado en un árbol de decisión: usar entidades dinámicas, es decir, entidades que son activadas o desactivadas según el camino que se siga; puede ser la posibilidad más adecuada, pero este modo es complicado de desarrollar en un programa que genera chatbots automáticamente.

El modo en que lo implemente fué el siguiente: consideré cada rama del árbol como una historia (ver Figura 45). Este método es similar a el chatbot de preguntas frecuentes por la recopilación de preguntas y respuestas, pero las historias son de mayor tamaño. En esta versión no hacen falta entidades.

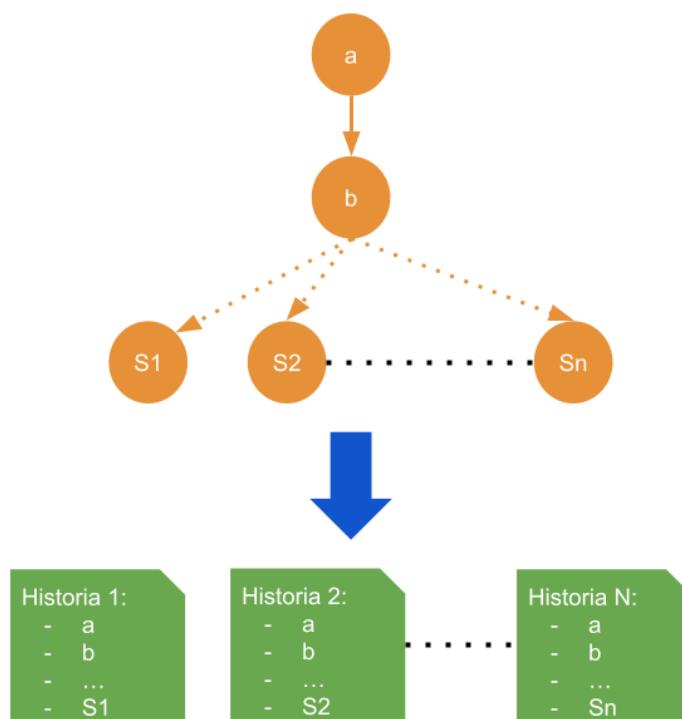


Figura 45: Diagrama de diseño del modelo de chatbot basado en árboles de decisión

Por lo tanto la plantilla deberá contener las preguntas y respuestas para los intents, además de incluir una lista detallando los intents o estados por los que pasa cada rama. El código está diseñado para identificar líneas con preguntas y respuestas, con solo preguntas o solo respuestas, y las historias (en este caso las ramas del árbol). Esto último se explica mejor en la sección 4.4.2.c de la memoria.

4.2. ANÁLISIS Y DISEÑO DE LA APLICACIÓN

En esta sección mostraré las etapas por las que ha ido pasando tanto el diseño de la interfaz web para comunicarse con el chatbot como la propia interfaz del script del proyecto, además de mostrar los detalles en profundidad de ambos.

4.2.1. Interfaz web

En este proyecto se han usado dos diseños para la implementación de la interfaz web. Fué necesaria la búsqueda de un segundo diseño porque el que se estaba usando dejó de dar servicio de repente y por lo tanto no podía usarse para el cometido que necesitaba.

Primer diseño

Basado en el diseño de *chatroom* de Scalableminds [39], este diseño es bastante simple y nos permite añadir reconocimiento del idioma e integrar una respuesta por voz a las preguntas hechas por el usuario (ver Figura 46).

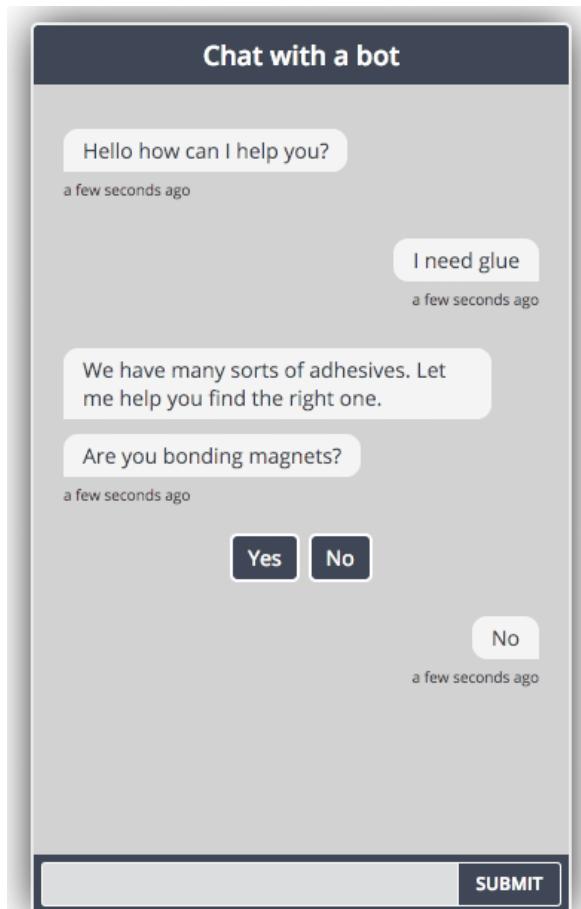


Figura 46: Ejemplo de conversación en la interfaz de Scalableminds

```

1 <html>
2   <head>
3     <link rel="stylesheet" href="https://npm-scalableminds.s3.eu-central-1.amazonaws.com/@scalableminds/chatroom@master/dist/Chatroom.css" />
4   </head>
5   <body>
6     <div class="chat-container"></div>
7
8     <script src="https://npm-scalableminds.s3.eu-central-1.amazonaws.com/@scalableminds/chatroom@master/dist/Chatroom.js"></script>
9     <script type="text/javascript">
10    var chatroom = new window.Chatroom({
11      host: "http://localhost:8085",
12      title: "Chatbot",
13      container: document.querySelector(".chat-container"),
14      welcomeMessage: "",
15      speechRecognition: "es-ES",
16      voicelang: "es-ES"
17    });
18    chatroom.openChat();
19  </script>
20 </body>
21 </html>

```

Figura 47: Código html interfaz web Scalableminds

Como se puede observar en la Figura 47, el código es bastante escueto y eso es posible gracias a que es importado (líneas 3 y 8 de la Figura 47) desde una web externa. De esta manera se ahorra un poco de espacio local, las secciones donde irían los ficheros CSS y Javascript.

En las líneas 15 y 16 de la Figura 47 podemos ver como se hace el reconocimiento del lenguaje y como se cambia el idioma con el que responde. Originalmente estaba establecido en inglés de Estados Unidos (en-US) pero una rápida búsqueda del código del idioma me permitió cambiarlo a español de España (es-ES) [40].

Este diseño iba a ser el definitivo para el proyecto pero los enlaces antes mencionados (ver Figura 47) dejaron de responder, y eso hacía imposible el funcionamiento de la interfaz con lo que tuve que buscar otras alternativas.

Segundo diseño y final

Para el segundo diseño use el *Chatbot-Widget* creado Jitesh Gaikwad [41]. Esta implementación permite uso de texto, botones, imágenes, videos, etc, básicamente está diseñada para usarse con chatbots creados con Rasa. La Figura 48 muestra la interfaz sin modificar que nos ofrece este diseño.

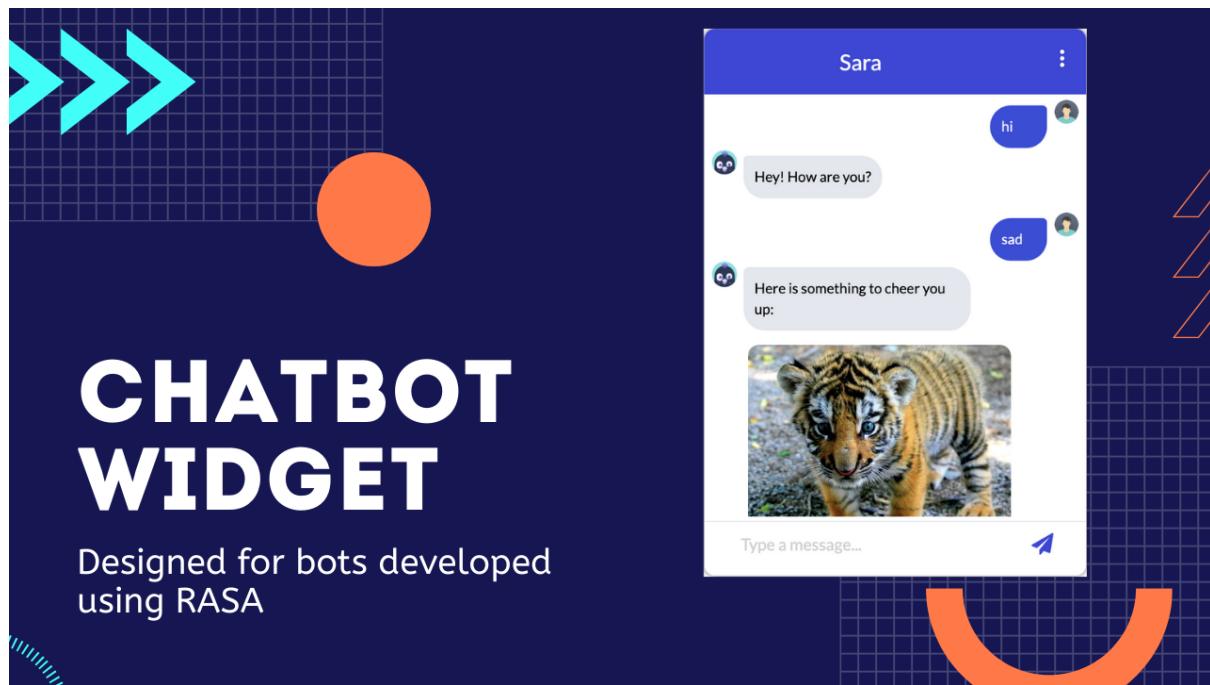


Figura 48: Interfaz de ejemplo de Chatbot-Widget

El código más importante de esta aplicación: css, javascript y demás funciones; viene incluido en carpetas locales por lo que nos quita la preocupación de que deje de estar disponible y no podamos usar nuestro chatbot.

Las modificaciones que hice sobre el código fueron traducir opciones y mensajes que se muestran en la ventana. También modifiqué un par de líneas de código en el archivo CSS, referidas a la apariencia del fondo de pantalla y un efecto de parpadeo que se produce al cargar página.



Figura 49: Página inicial de la interfaz web del chatbot

La Figura 49 muestra la pantalla con la que se encuentra cualquiera al cargar el enlace del chatbot ya habiendo sido modificado. Podemos ver un fondo simple en blanco, y en la esquina inferior derecha una zona coloreada de morado donde hay un mensaje de saludo y un ícono con un búho correspondiente a nuestro chatbot. Esa zona coloreada desaparece a los tres segundos dejando solo el ícono.

La idea es que esto es una muestra del chatbot, por eso no se ha coloreado ni elegido un fondo para la pantalla. Cada chatbot creado puede ser muy diferente del anterior y al ser generado en un archivo html, es muy sencillo tanto cambiar tanto el fondo, el ícono del chatbot, el nombre y los íconos usados dentro del chat. Es ampliamente personalizable.

En cuanto a la elección de colores, nombre e ícono, he mantenido los que había por defecto y estas son las razones:

- La web oficial de Rasa usa un color morado parecido.
- El nombre de Sara se corresponde a Rasa escrito al revés.
- El ícono o “mascota” es usado para referirse a la plataforma de Rasa en otras páginas webs y es usado en la propia web de Rasa.

La razón principal por la que no los he modificado es que, al estar desarrollado con la plataforma Rasa, he intentado mantener la identidad de esta. Esto como ya he indicado antes, es la versión de muestra y puede ser modificada por el cliente a su gusto.

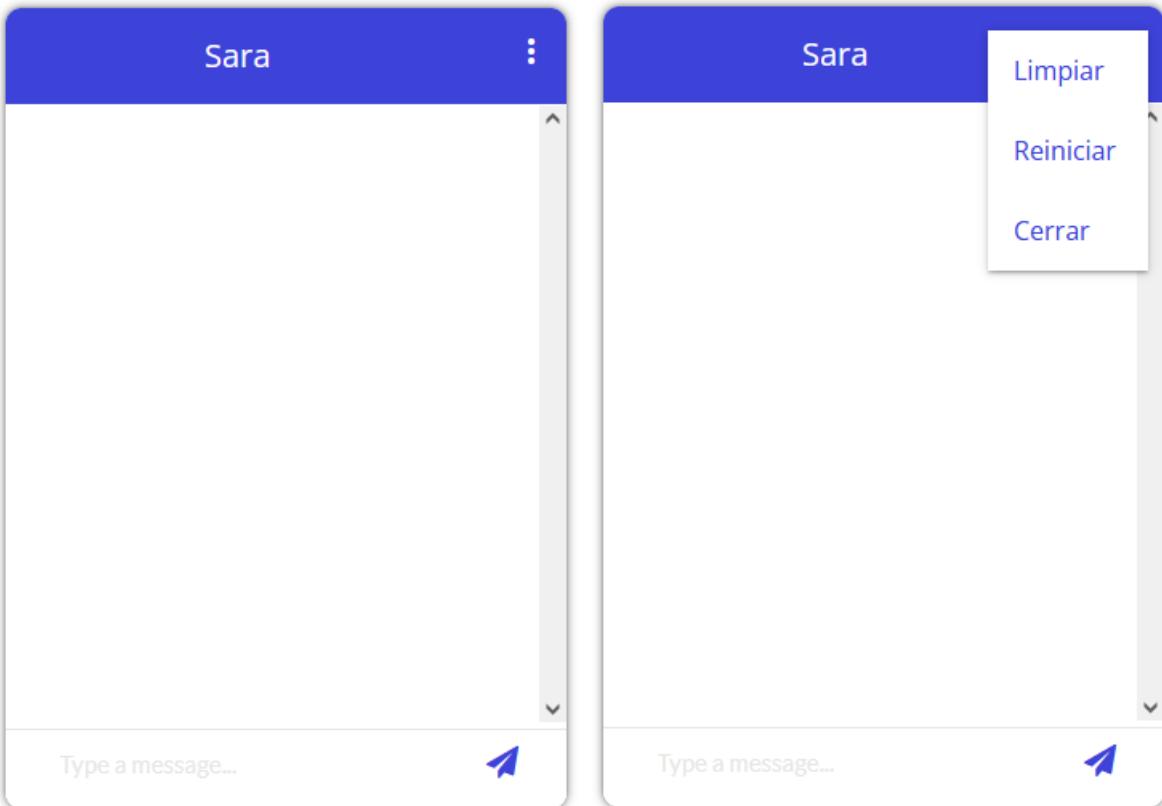


Figura 50:: Webchat desplegado a la izquierda, opciones del chatbot a la derecha

Volviendo a la pantalla de inicio, si queremos empezar una conversación con el chatbot hay que pulsar el ícono del chatbot en la esquina inferior derecha (ver Figura 49). Una vez desplegado (ver Figura 50) el usuario puede escribir en la parte de abajo y la conversación se mostrará arriba, en el cuadro de diálogo.

Si se pulsa sobre los tres puntitos al lado del nombre del chatbot se despliega una lista con tres opciones:

- **Limpiar:** Borra el historial de conversaciones dentro del cuadro de diálogo. Permite seguir con la conversación anterior sin problemas.
- **Reiniciar:** Reinicia el chatbot, es decir, inicia una nueva conversación. Al reiniciar el chatbot se borra la memoria que este tiene y es necesario proporcionar de nuevo los datos que el chatbot requiera para cumplir su objetivo.
- **Cerrar:** Cierra el cuadro de diálogo y deja la pantalla como estaba antes de desplegar el webchat, con el ícono en la esquina inferior derecha.

Varios ejemplos de funcionamiento del webchat están en las secciones 4.3.2.a, 4.3.2.b y 4.3.2.c de la memoria.

4.2.2. Interfaz del programa

La interfaz propia del programa se hizo necesaria dada la complejidad que estaba alcanzando el código del mismo. Se ideó para facilitar la personalización de un chatbot por parte de un usuario sin tocar el código interno del programa. La Figura 51 muestra su apariencia final.

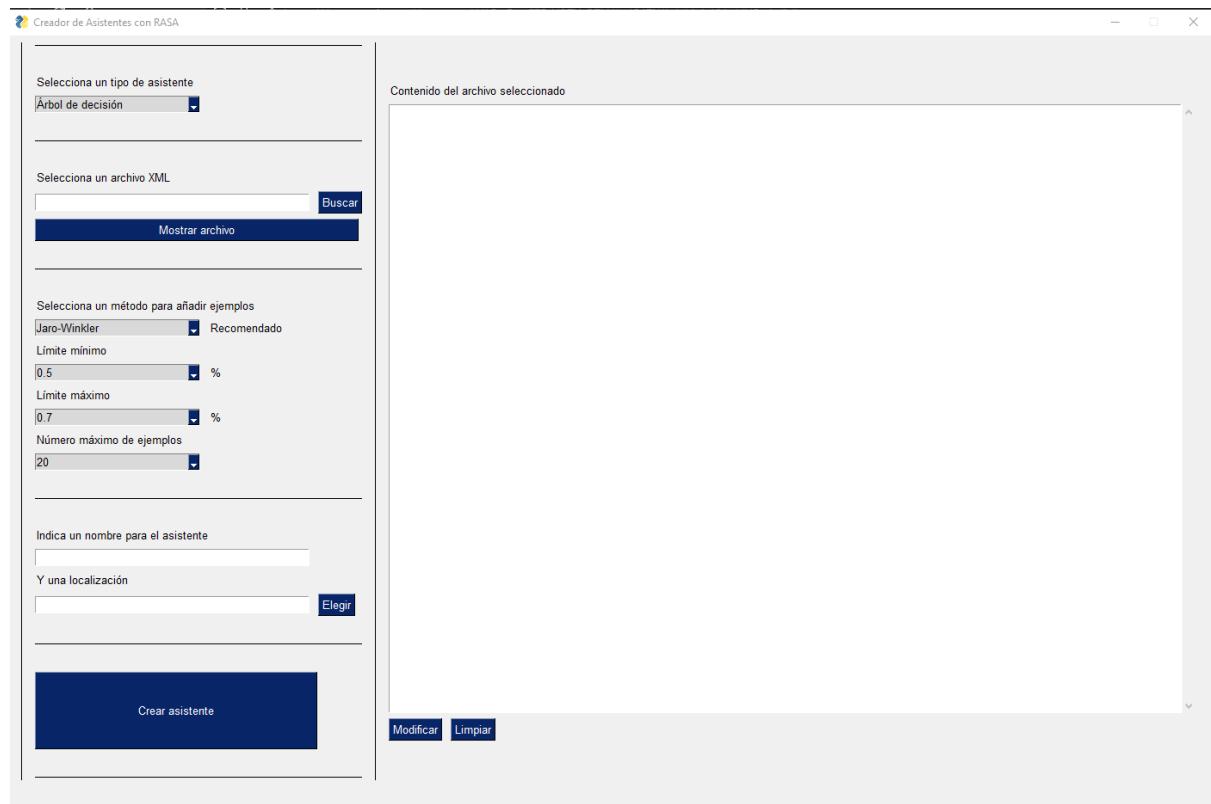


Figura 51: Interfaz principal del programa

Esta interfaz permite elegir el tipo de chatbot que se quiere crear, seleccionar la plantilla necesaria para crearlo, el algoritmo con distintos parámetros relacionados con los ejemplos a añadir, el nombre del chatbot y la ubicación del mismo. También permite la visualización de la plantilla que se usará para creación del chatbot e incluso su modificación en caso de que contenga errores. Con una plantilla con errores no es posible crear un chatbot (ver sección 4.4.2 de esta memoria).

La interfaz está creada con PySimpleGUI que permite una amplia selección de temas (ver Figura 52). En las etapas de desarrollo se usó un tema más oscuro (“Dark Amber”) con toques naranjas, pero al final se decidió cambiarlo por otro tema que proporciona un aspecto más “profesional” (“DefaultNoMoreNagging”).



Figura 52: Gama de temas que ofrece PySimpleGUI

A continuación se explicarán las distintas opciones y el funcionamiento de cada una de las secciones de la interfaz.

Selección de tipo de chatbot y plantilla de texto

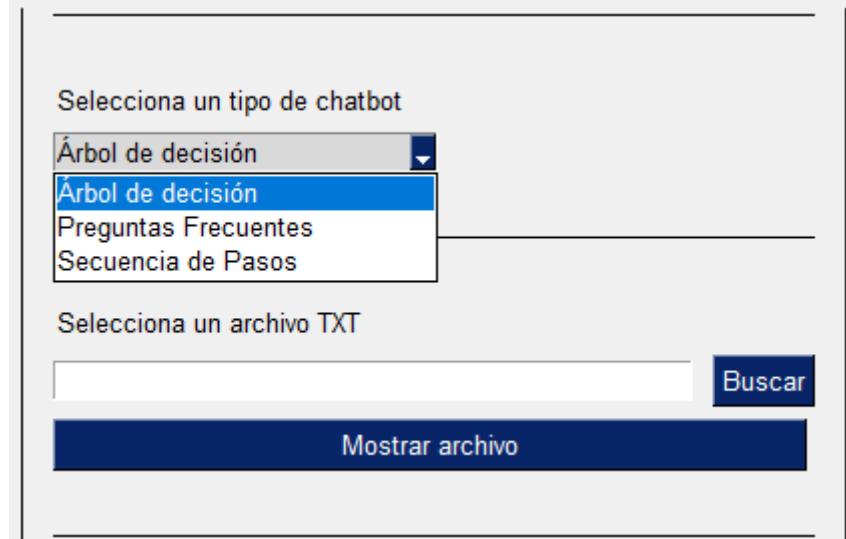


Figura 53: Selección de tipo de chatbot y plantilla de texto

En la parte de arriba de la Figura 53, tenemos una lista de selección de los tres tipos de chatbots del proyecto. La parte de abajo muestra dos botones:

1. **Buscar** → Abre una ventana para seleccionar la plantilla. Está configurado para buscar archivos TXT.
2. **Mostrar archivo** → Carga la plantilla seleccionada en el panel grande de la derecha de la interfaz (ver Figura 54).

Panel de edición de plantilla

The screenshot shows a software interface for editing templates. At the top, there's a header bar with the title 'Panel de edición de plantilla'. Below this is a large text area labeled 'Contenido del archivo seleccionado' (Content of the selected file). The text area contains several sections of template code, each starting with '###' and followed by descriptive comments:

- Section 1: '### primero, si hay, se ponen los intents con respuestas y ejemplos ###'
Content: intent:PreguntaInicial;(Respuesta1|Otra respuesta1);{Pregunta1|Otra pregunta1}
- Section 2: '### seguido, van los intents con respuestas ###'
Content: intent:Respuesta1;{Esto es la respuesta1|Esto es otra respuesta1}
intent:Respuesta2;{Esto es la respuesta2|Esto es otra respuesta2}
- Section 3: '### ahora, las preguntas a las respuestas anteriores ###'
Content: response:Pregunta;({Pregunta?}|{Esto es una pregunta?})
- Section 4: '### penultimo, los responses con las respuestas finales (nodos hoja) ###'
Content: response:Opcion1;(Es la opcion1)
response:Opcion2;(Es la opcion2)
- Section 5: '### Y por ultimo, las stories ###'
Content: story:PreguntaInicial;Pregunta(Respuesta1);*Opcion1
story:PreguntaInicial;Pregunta(Respuesta2);*Opcion2

At the bottom of the panel, there are two buttons: 'Modificar' (Modify) and 'Limpiar' (Clear).

Figura 54: Panel de edición de plantilla

La Figura 54 corresponde con la parte derecha de la interfaz del programa. Consta de un gran panel donde se puede cargar el contenido de la plantilla seleccionada y de dos botones en la parte inferior.

El panel es totalmente editable y la funcionalidad de los botones es la siguiente:

- Limpiar: Limpia el panel, borra el contenido sin modificar la plantilla
- Modificar: Sobrescribe el contenido de la plantilla usando las modificaciones hechas sobre el panel superior.

Selección de algoritmo y parámetros relacionados

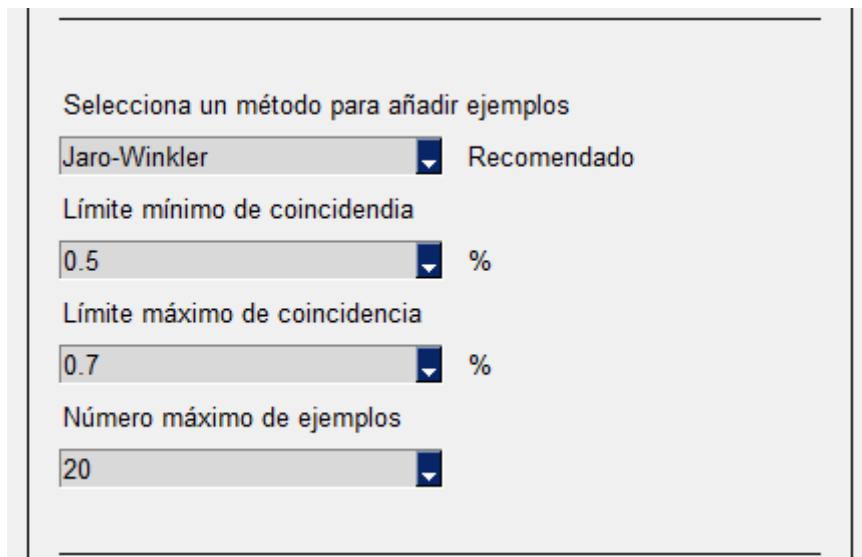


Figura 55: Selección de algoritmo y parámetros relacionados

En la sección que muestra la Figura 55, podemos seleccionar el algoritmo que queremos que el chatbot use, los parámetros de coincidencia para añadir ejemplos y el número máximo que puede albergar.

Los cuatro parámetros son listas y estos son los valores que contienen (se describe desde arriba hacia abajo):

- Los algoritmos son los usados en el proyecto:
 - **Jaro-Winkler.**
 - **LCS para SequenceMatcher.**
 - **Coseno para la similitud coseno.**
 - **BM25.**
- Límite mínimo de coincidencia representa el porcentaje mínimo que un nuevo ejemplo debe parecerse a uno existente en el chatbot. De esta manera se asegura un mínimo de relación entre los ejemplos, es decir, que se parezca un mínimo para ser considerado. El intervalo de valores es el siguiente: [0.1, 0.2, 0.3, 0.4, 0.5].
- Límite máximo de coincidencia establece lo máximo que un nuevo ejemplo puede parecerse a uno ya existente. Así nos aseguramos de no llenar el chatbot de ejemplos prácticamente iguales. El intervalo es el siguiente: [0.6, 0.7, 0.8, 0.9].
- Número máximo de ejemplos constituye el máximo de ejemplos que puede contener cada intent del chatbot. Su intervalo: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100].

Los valores por defecto son los mostrados en la Figura 62. Si por algún casual queremos usar estos para crear un chatbot: a la hora de añadir ejemplos, usando el algoritmo Jaro-Winkler, chequeará si el ejemplo se parece entre un 50% y 70% a cualquiera de los que hay disponibles y por último si dispone de más de 20 ejemplos en el intent predicho.

Nombre, localización y creación del chatbot

The screenshot shows a user interface for creating a chatbot. At the top, there is a text input field with the placeholder "Indica un nombre para el asistente". Below it is another text input field with the placeholder "Y una localización" and a blue "Elegir" button to its right. At the bottom of the interface is a large blue rectangular button with the white text "Crear chatbot".

Figura 56: Nombre, localización y creación del chatbot

La Figura 56 muestra la última sección de la interfaz. En ella podemos indicar el nombre del chatbot, la localización y por último existe un botón para crear el chatbot.

Una de las particularidades es que el nombre se rellena automáticamente una vez se haya seleccionado una plantilla (ver Figura 53), seleccionando el nombre de esta. Esto no impide que se pueda editar luego y cambiar por cualquier nombre deseado.

El botón elegir abre una ventana donde se debe elegir la localización en la que será ubicado el chatbot similar al de la figura 60.

Por último, el botón crear chatbot hace lo que su nombre indica. Hay que indicar que hay un control de fallos que se explica más adelante en la sección 4.4 de esta memoria, y que advierte de cosas tan simples como por ejemplo:

- No se puede crear un chatbot sin plantilla
- No hay plantilla que editar
- Indique un nombre para el chatbot
- ...

Y aparte de un control de fallos en el contenido de las plantillas en la misma sección (4.4).

```

primeras_columnas = [
    [sg.HSeparator()],
    [sg.T()],
    [sg.Text("Selecciona un tipo de chatbot")],
    [sg.Combo(size=(25,1), values=["Árbol de decisión", "Preguntas Frecuentes", "Secuencia de Pasos"], default_value="Árbol de decisión", readonly=True, key="-TIPO-", enable_events=True)],
    [sg.T()],
    [sg.HSeparator()],
    [sg.T()],
    [sg.Text("Selecciona un archivo TXT")],
    [sg.Input(key="-FILE-", enable_events=True), sg.FileBrowse(button_text="Buscar", file_types=((("TXT Files", "*.txt"), ("All Files", "*.*"))))],
    [sg.Button("Mostrar archivo", size=(46,1), key="-SHOW-")],
    [sg.T()],
    [sg.HSeparator()],
    [sg.T()],
    [sg.Text("Selecciona un método para añadir ejemplos")],
    [sg.Combo(size=(25,1), values=["Jaro-Winkler", "LCS", "Coseno", "BM25"], default_value="Jaro-Winkler", readonly=True, key="-ALG-", enable_events=True), sg.Text("Recomendado", key="-RECOMENDADO-")],
    [sg.Text("Límite mínimo de coincidencia")],
    [sg.Combo(size=(25,1), values=[0.1, "0.2", "0.3", "0.4", "0.5"], default_value="0.5", readonly=True, key="-MINLIM-", enable_events=True), sg.Text("%")],
    [sg.Text("Límite máximo de coincidencia")],
    [sg.Combo(size=(25,1), values=[0.6, "0.7", "0.8", "0.9"], default_value="0.7", readonly=True, key="-MAXLIM-", enable_events=True), sg.Text("%")],
    [sg.Text("Número máximo de ejemplos")],
    [sg.Combo(size=(25,1), values=[10, "20", "30", "40", "50", "60", "70", "80", "90", "100"], default_value="20", readonly=True, key="-EXAMPLES-", enable_events=True)],
    [sg.T()],
    [sg.HSeparator()],
    [sg.T()],
    [sg.Text("Indica un nombre para el asistente")],
    [sg.Input(key="-NAME-", enable_events=True)],
    [sg.Text("Y una localización")],
    [sg.Input(key="-FOLDER-", enable_events=True), sg.FolderBrowse(button_text="Elegir")],
    [sg.T()],
    [sg.HSeparator()],
    [sg.T()],
    [sg.Submit(size=(40,5), button_text="Crear chatbot", key="-OK-")],
    [sg.T()],
    [sg.HSeparator()]
]

segunda_columna = [
    [sg.Text("Contenido del archivo seleccionado")],
    [sg.MLine(size=(40,44), key="-CONTENT-")],
    [sg.Button(button_text="Modificar", key="-MOD-"), sg.Button(button_text="Limpiar", key="-CLEAN-")]
]

```

Figura 57: Código de la interfaz del programa

En la figura 57 se puede observar el código necesario para generar la interfaz.

4.3. IMPLEMENTACIÓN

Esta sección está destinada a explicar el proceso de desarrollo del proyecto, comentando desde las primeras etapas del proyecto hasta su versión final. Mostrando la evolución de las plantillas, código y acabando con ejemplos de funcionamiento apoyados con algunos diagramas casos de uso. La plantilla usada para los casos de uso es la mostrada en la Tabla 13.

Caso de uso	<< Nombre del CU >>	<< ID >>
Actores	<< Listado de los actores participantes en el CU >>	
Precondiciones	<< Condiciones que se tienen que dar para la realización del CU >>	
Postcondiciones	<< Efectos tras la realización del CU >>	

Tabla 13: Plantilla para la definición de los casos de uso

4.3.1. Entrada XML

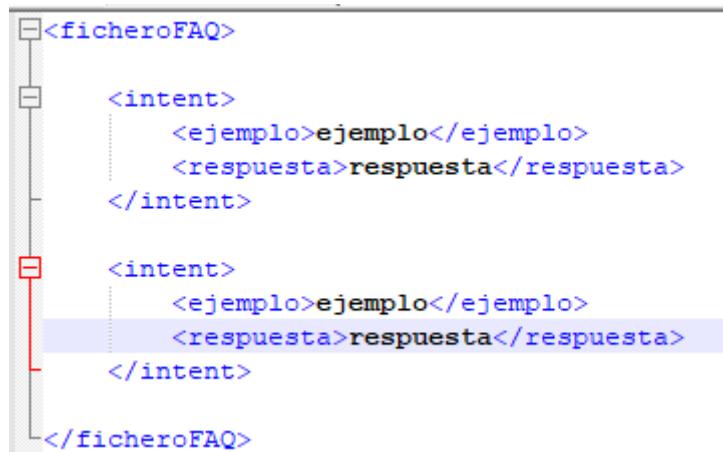
En las primeras etapas del desarrollo la entrada de texto se hacía mediante un fichero XML. Usando la librería Minidom se puede extraer fácilmente el contenido de una etiqueta en concreto (ver Figura 28).

Esta subsección será breve y me limitaré a comentar la estructura de las plantillas usadas y como el programa lo interpreta. Quiero incluir esta mini-sección ya que el cambio al otro modelo de entrada es bastante posterior en el tiempo que he dedicado al desarrollo del proyecto.

Por último, quiero comentar que esta versión no dispone corrección de estructura de plantillas como si tiene la final. Esto es debido a que esta versión dejó de actualizarse en favor de la versión final con entrada en texto plano (TXT).

Preguntas Frecuentes

Como se observa en la Figura 58, la estructura posee etiquetas llamadas *intent*, y en ellas se alojan otras dos etiquetas llamadas *ejemplo* y la *respuesta*. *Ejemplo* contiene los ejemplos dados al *intent* y *respuesta* corresponde con las respuestas que da el chatbot a ese *intent*.



```
<ficheroFAQ>
  <intent>
    <ejemplo>ejemplo</ejemplo>
    <respuesta>respuesta</respuesta>
  </intent>

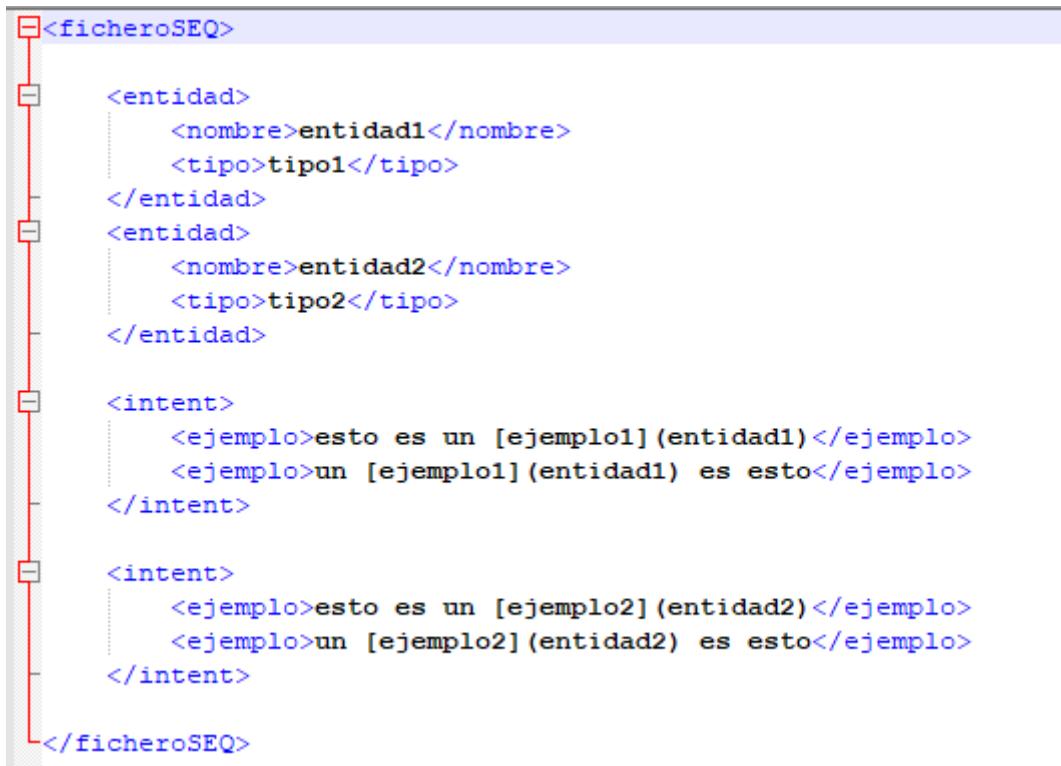
  <intent>
    <ejemplo>ejemplo</ejemplo>
    <respuesta>respuesta</respuesta>
  </intent>
</ficheroFAQ>
```

Figura 58: Ejemplo de plantilla FAQ en formato XML

La implementación del programa es la siguiente:

1. Busca la etiqueta *intent* y devuelve la lista de ellos.
2. Para cada *intent* busca sus ejemplos y respuestas. La plantilla mostrada en la figura 65 cuenta con solo uno de cada, pero el programa está diseñado para devolver una lista de ellos.
3. Por último, escribe la lista de ejemplos en nlu.yml y la lista de respuestas en domain.yml

Pasos en secuencia



```
<ficheroSEQ>
  <entidad>
    <nombre>entidad1</nombre>
    <tipo>tipo1</tipo>
  </entidad>
  <entidad>
    <nombre>entidad2</nombre>
    <tipo>tipo2</tipo>
  </entidad>

  <intent>
    <ejemplo>esto es un [ejemplo1] (entidad1)</ejemplo>
    <ejemplo>un [ejemplo1] (entidad1) es esto</ejemplo>
  </intent>

  <intent>
    <ejemplo>esto es un [ejemplo2] (entidad2)</ejemplo>
    <ejemplo>un [ejemplo2] (entidad2) es esto</ejemplo>
  </intent>
</ficheroSEQ>
```

Figura 59: Ejemplo de plantilla de Pasos en Secuencia en XML

Esta plantilla (Figura 59) incluye la etiqueta *entidad*, que contiene su nombre y tipo. El tipo de una entidad se puede mirar en la documentación oficial de Rasa, los más comunes son “text” para texto normal y “float” para números.

También incluye la etiqueta *intent* como en la anterior plantilla (ver Figura 58), pero en este caso solo contiene ejemplos. Esto es debido a que el programa está configurado para extraer las entidades en una lista y hacer una pregunta automática para cada una de ellas. Por ejemplo, teniendo las entidades: nombre y teléfono; el programa escribirá cuando y donde corresponda las preguntas: “Cuál es tu nombre?” y “Cuál es tu teléfono?”, usando la estructura “Cual es tu <nombre_entidad>?” para cada una de las entidades.

Este tipo de chatbot no necesita historias dado a que la única conversación que conoce se activa mediante un formulario. Al saludar al chatbot este reconoce el intent saludo y su respuesta es activar el formulario donde pregunta por las entidades.

Explicado todo, el funcionamiento del programa es el siguiente:

1. Recoge la lista de entidades proporcionada por la plantilla.
2. Escribe los ejemplos a los intents en nlu.yml
3. Escribe los datos necesarios en domain.yml y las preguntas automáticas de las entidades.

Y el funcionamiento del chatbot:

1. El usuario saluda, el programa reconoce el saludo y activa el formulario.

2. Pregunta por las entidades en secuencia.
3. Recoge los datos y muestra en pantalla los resultados.

Árbol de decisión

```

<!-- INSTENTS -->
<!-- RESPONSES -->
<!-- STORIES -->

<intent>
  <nombre>saludo</nombre>
  <ejemplo>holo</ejemplo>
  <ejemplo>buenas</ejemplo>
</intent>

<intent>
  <nombre>estado_feliz</nombre>
  <ejemplo>estoy feliz</ejemplo>
  <ejemplo>feliz</ejemplo>
  <ejemplo>contento</ejemplo>
  <ejemplo>alegre</ejemplo>
  <ejemplo>con una sonrisa</ejemplo>
</intent>

<intent>
  <nombre>estado_triste</nombre>
  <ejemplo>estoy triste</ejemplo>
  <ejemplo>triste</ejemplo>
  <ejemplo>enfadado</ejemplo>
  <ejemplo>llorando</ejemplo>
  <ejemplo>cabreado</ejemplo>
</intent>

```

```

<response>
  <nombre>utter_saludo</nombre>
  <respuesta>¡Hola!</respuesta>
  <respuesta>¡Hey!</respuesta>
  <respuesta>¡Muy Buenas!</respuesta>
</response>

<response>
  <nombre>utter_preguntar_estado</nombre>
  <respuesta>¿Como te sientes?</respuesta>
  <respuesta>¿Me puedes decir como te sientes?</respuesta>
</response>

<response>
  <nombre>utter_feliz</nombre>
  <respuesta>¡Que bien!</respuesta>
  <respuesta>¡Fenomenal!</respuesta>
  <respuesta>¡Espléndido!</respuesta>
</response>

<response>
  <nombre>utter_triste</nombre>
  <respuesta>¡Vaya :(</respuesta>
  <respuesta>¡Eso no esta bien!</respuesta>
  <respuesta>¡Hay que arreglar eso!</respuesta>
</response>

```

```

<story>
  <nombre>happy path</nombre>
  <user>saludo</user>
  <bot>utter_saludo</bot>
  <bot>utter_preguntar_estado</bot>
  <user>estado_feliz</user>
  <bot>utter_feliz</bot>
</story>

<story>
  <nombre>sad path</nombre>
  <user>saludo</user>
  <bot>utter_saludo</bot>
  <bot>utter_preguntar_estado</bot>
  <user>estado_triste</user>
  <bot>utter_triste</bot>
</story>

```

Figura 60: Ejemplo de plantilla de árbol de decisión en XML

La plantilla usada para este chatbot (ver Figura 60) es un poco más amplia y diferente a las demás. Los intents ahora contienen el nombre y sus los ejemplos, las responses son las interacciones del chatbot y contienen lo mismo que los intents.

Para las historias, las ramas del árbol, hay un formato nuevo:

- Nombre: Rasa no necesita nombres para las historias, pero siempre viene bien poner uno para poder identificarlas.
- La etiqueta *user* viene a indicar las interacciones realizadas por el usuario y se refiere a la línea “intent” de la story (ver Figura 12).
- La etiqueta *bot* indica cuando el chatbot responde y corresponde con la línea “action” de la story (ver Figura 12).

El funcionamiento del programa es similar al chatbot de preguntas frecuentes: escribe los intents en nlu.yml y las responses en domain.yml. Para las stories (como se puede observar en la Figura 61) va identificando las etiquetas y escribe su contenido añadiendo lo necesario para que Rasa sepa cómo tiene que mantener la conversación.

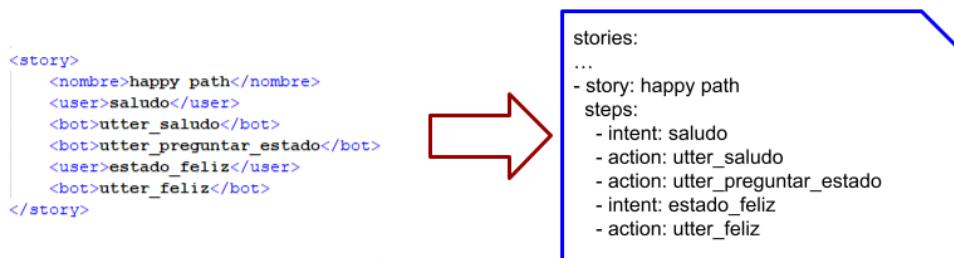


Figura 61: Ejemplo de identificación de las stories en XML

4.3.2. Entrada TXT

Esta es la versión final, introducida en las etapas finales del desarrollo. El funcionamiento es igual que el visto en la sección anterior, solo cambia la forma en la que procesan los datos antes de escribirlos en el formato que Rasa reconoce.

Esta sección ofrece una explicación sobre el funcionamiento de ese procesado mostrando las plantillas y explicando cómo el programa lo interpreta, apoyándose en fragmentos del propio código del programa. Para terminar, se añadirán algunos ejemplos de uso del webchat.

4.3.2.a Preguntas Frecuentes

Plantilla

La plantilla mantiene una estructura similar a la de la Figura 58 en cuanto a los intents, pero en esta se añaden líneas para las historias.

```
### primero van los intents con respuestas y ejemplos ###

intent:Preguntal; (Respuestal|Respuestaal);{Preguntal|Preguntaal}
intent:Preguntal; (Respuesta2lRespuestaa2);{Pregunta2|Preguntaa2}
intent:Preguntal; (Respuesta3lRespuestaa3);{Pregunta3|Preguntaa3}

### y segundo y último, las stories ###

story:Preguntal
story:Pregunta2
story:Pregunta3
```

Figura 62: Ejemplo plantilla FAQ en texto plano (TXT)

El formato se puede observar en la Figura 69, y es el siguiente:

1. Intent: tiene tres elementos separados por el carácter “;”
 - a. **PreguntaN**: equivale al nombre del intent.
 - b. **RespuestaN|...**: son las interacciones del chatbot. Van entre paréntesis.
 - c. **PreguntaN|...**: corresponde a los ejemplos del intent. Se alojan entre llaves.
2. story: contiene un solo elemento.
 - a. **PreguntaN**: Las ramas tienen una sola hoja.

El formato que se usa para los separadores se debe respetar para que el programa sepa cómo separar los elementos y el chatbot se cree correctamente. En esta plantilla se usa el carácter “;” para separar los elementos de los intents (nombre, preguntas y respuestas), no

siendo necesario para las stories al tener solo un elemento. Para separar las muestras o ejemplos que haya en los elementos **preguntas** y **respuestas** se usa el carácter “|”.

El funcionamiento del procesado de la entrada de texto es el siguiente. El programa va leyendo línea por línea y:

1. Si la línea empieza por intent → introduce la línea entera borrando el comienzo (borra “intent:”) en el array de intents (ver Figura 63).
2. Si empieza por story → hace lo mismo pero introduciendo la línea en el array de stories (ver Figura 63).

```
entities_array = []
intents_array = []
responses_array = []
stories_array = []
```

Figura 63: Variables para alojar datos procesados

Una vez terminados los datos de entrada, los arrays son procesados otra vez para incluir los datos en los ficheros de Rasa. La Figura 64 muestra el fragmento del código dónde se incluyen los datos en el fichero nlu.yml.

Ya habiéndose separado por “;”, en la línea 110 se identifica el elemento (las llaves) como ejemplo del intent. En la línea 114 se forma otra lista ahora usando como separador el mencionado “|”, y se va escribiendo cada uno en las líneas del fichero.

```
110      elif(a.startswith('{')):
111          # se escribe en nlu
112          nlu.write('- intent: {}\\n'.format(nombre))
113          nlu.write('  examples: |\\n')
114          res = a.replace('{','').replace('}','').split('|')
115          for r in res:
116              nlu.write('    - \"{}\"\\n'.format(r))
```

Figura 64: Fragmento de código donde se escriben los ejemplos en nlu.yml

El código que añade las respuestas del chatbot es similar, solo cambia el identificador de la línea 110 de la Figura 64 por el carácter ““.

```
stories.write('  - intent: {}\\n'.format(g))
stories.write('  - action: action_mirar_ejemplos\\n')
stories.write('  - action: utter_{}\\n'.format(g))
```

Figura 65: Fragmento de código donde se añaden las historias en el chatbot FAQ

En la Figura 65 se puede ver como el programa escribe cada historia en su fichero correspondiente, siendo *g* un elemento del array de historias visto en la Figura 70. La línea de enmedio correspondiente a “*action_mirar_ejemplos*” se refiere a la función que se usa para añadir ejemplos al chatbot, que será cubierta en la sección 4.3.3 de esta memoria.

Ejemplo CorreoUGR

Para este ejemplo he usado la página de preguntas frecuentes sobre el correo de la ugr [42] (ver Figura 66). Esta página web contiene nueve preguntas con sus respectivas respuestas, y es un ejemplo simple para implementar en nuestro programa.

Preguntas Frecuentes sobre Correo electrónico

- ▼ ¿Cuál es el número máximo de destinatarios en un único mensaje?
 - El número máximo es 150. Si necesita enviar asiduamente mensajes a más destinatarios debe usar una lista de distribución.
- ¿Cuál es el número máximo de correos electrónicos que puedo enviar al día?
- ¿De cuánto espacio dispongo para mi cuenta de correo?
- ¿Puedo consultar la cuota que estoy usando en el correo electrónico?
- ¿Cómo puedo recuperar espacio libre en el correo electrónico una vez sobrepasada la cuota permitida?
- ¿Cuál es el tamaño máximo de los archivos adjuntos a enviar con un correo?
- ¿Para qué sirve la carpeta BUZONdeEntradaUGR?
- ¿Puedo avisar automáticamente que estoy de vacaciones a mis contactos cuando me manden un correo?
- ¿Qué es webmail?

Figura 66: Página de preguntas frecuentes sobre el correo de la ugr

```

1 intent:Pregunta1;(El número máximo es 150. Si necesita enviar asiduamente mensajes a más destinatarios debe usar una lista de distribución);{¿Cuál
2
3 intent:Pregunta2;(A través de las plataformas de Webmail y desde el resto de formas de acceso al correo, el número máximo de correos electrónicos
4
5 intent:Pregunta3;(Las cuentas del tipo @correo.ugr.es tienen espacio para correo: 500 MB);{¿De cuánto espacio dispongo para mi cuenta de correo?}
6
7 intent:Pregunta4;(Está disponible a su izquierda en el programa Webmail. Podrá ver qué espacio de su cuenta está usado y los límites de cuota que p
8
9 intent:Pregunta5;(Cuando se sobrepasa la cuota permitida el usuario recibirá un correo avisándole de que ha superado su cuota y que debe intenta
10
11 intent:Pregunta6;(Para el PAS/PDI de la Universidad, desde Webmail es posible enviar en un correo archivos adjuntos de hasta 10MB de tamaño);{¿Cuál
12
13 intent:Pregunta7;(Con objeto de mejorar el rendimiento de su correo electrónico, existe una funcionalidad en las estafetas centrales de correo UGR
14
15 intent:Pregunta8;(Si, a través de la configuración de webmail, podemos activar una notificación a nuestros contactos.);{¿Puedo avisar automáticamente
16
17 intent:Pregunta9;(Se trata de una herramienta para consultar correo electrónico mediante un navegador de Internet);{¿Qué es webmail?}
18
19 story:Pregunta1
20 story:Pregunta2
21 story:Pregunta3
22 story:Pregunta4
23 story:Pregunta5
24 story:Pregunta6
25 story:Pregunta7
26 story:Pregunta8
27 story:Pregunta9
28 story:Pregunta9

```

Figura 67: Plantilla de las preguntas frecuentes del correo de la ugr

En la Figura 67 he trasladado las preguntas y respuestas de la Figura 66 al formato de entrada correcto para el chatbot. En este caso sólo contiene un ejemplo para ambos, una pregunta y una respuesta para cada intent, lo que hará que el chatbot intente añadirle

ejemplos a cada uno hasta llenar el espacio que le hemos definido en los parámetros (sección 4.2.2, Figura 55).

A continuación se muestran tablas con algunos casos de uso de este ejemplo:

Caso de uso	Pregunta: “¿Qué es webmail?”	CU_01
Actores	Usuario/Cliente	
Precondiciones	Ninguna	
Postcondiciones	Respuesta a la pregunta: ¿Qué es webmail?	

Tabla 14: Ejemplo caso de uso 01. Chatbot FAQ

Caso de uso	Pregunta: “¿De cuánto espacio dispongo para mi cuenta de correo?”	CU_02
Actores	Usuario/Cliente	
Precondiciones	Ninguna	
Postcondiciones	Respuesta a la pregunta: “¿De cuánto espacio dispongo para mi cuenta de correo?”	

Tabla 15: Ejemplo caso de uso 02. Chatbot FAQ

Con un par de ejemplos se puede observar que todos los casos de uso son iguales. Cada uno se basa en responder a la pregunta correspondiente, y hay tantos como preguntas tenga el chatbot (en este caso nueve).



Figura 68: Ejemplo conversaciones. Chatbot FAQ.

Para terminar, podemos observar como la Figura 68 contiene varios de los ejemplos mencionados en los casos de uso y un par más usando la interfaz web diseñada para el programa.

4.3.2.b Secuencia de pasos

Plantilla

La plantilla de la Figura 69 contiene tres etiquetas:

1. **entities**: una forma de poner un conjunto de entidades en una línea, usando como separador el carácter “;”.
2. **entity**: otra forma de poner las entidades, pero de una forma individual
3. **intent**: el intent correspondiente a las entidades definidas. Este tipo de chatbot contiene dos elementos.
 - a. el nombre del intent
 - b. los ejemplos. Van entre llaves y usan el separador habitual “|”.

```

# primero definimos las entidades

entities:entidad1;entidad2
entity:entidad3


# seguido de los intent con respuestas para las entidades

intent:pedir_entidad1;{Es [entidad1] (entidad1) | Es otra [entidad1] (entidad1)}
intent:pedir_entidad2;{Es [entidad1] (entidad2) | Es otra [entidad1] (entidad2)}
intent:pedir_entidad3;{Es [entidad1] (entidad3) | Es otra [entidad1] (entidad3)}

```

Figura 69: Plantilla usada para el chatbot de pasos en secuencia en texto plano (TXT)

Las historias en este tipo de chatbot se agrupan en una, preguntar por el valor de las entidades. Esto se hace mediante un formulario y su funcionamiento será explicado más adelante en esta sección.

El funcionamiento para los intents es el mismo que el mostrado en la Figura 64. Podemos observar que el formato de los ejemplos de estos contienen los caracteres "[]" y "()", esto es debido a que Rasa marca así los ejemplos de entidades para que el chatbot pueda reconocerlos. Es decir, teniendo la entidad *nombre*, un ejemplo sería: "Me llamo [Pablo](nombre)", así estamos diciéndole a Rasa que el valor de la entidad *nombre* es "Pablo".

En cuanto a las entidades, el programa las procesa como puede verse en la Figura 70. Siguiendo el formato que usa Rasa para el fichero domain.yml (ver Figura 13), y sabiendo que ya están mapeadas las entidades en su array correspondiente, el programa:

- en la línea 101, escribe la lista de entidades.
- en la línea 107, escribe un slot para cada una.
- en la línea 115 en adelante, crea el formulario contando con todos los slots.

```

100      # entidades en domain
101      domain.write('entities:\n')
102      for e in entities_array:
103          domain.write(' - {}\\n'.format(e))
104
105      # slots en domain
106      domain.write('slots:\n')
107      for e in entities_array:
108          domain.write(' - {}:\\n'.format(e))
109          domain.write('     type: text\\n')
110          domain.write('     mappings:\\n')
111          domain.write('         - type: from_entity\\n')
112          domain.write('             entity: {}\\n'.format(e))
113
114      # form en domain
115      domain.write('forms:\n')
116      domain.write('    data_form:\\n')
117      domain.write('    required_slots:\\n')
118      for e in entities_array:
119          domain.write('        - {}\\n'.format(e))

```

Figura 70: Procesado de array de entidades en el chatbot de pasos en secuencia

Las respuestas automáticas se crean de la siguiente manera (ver Figura 71). Recorre la lista de entidades y para todas hace la misma pregunta. De esta manera evitamos poner los ejemplos de respuestas en la plantilla de entrada.

```

# responses en domain
domain.write('responses:\n')
for e in entities_array:
    domain.write('    utter_ask_{}:\\n'.format(e))
    domain.write('        - text: "¿Cuál es tu {}?"\\n'.format(e))

```

Figura 71: Fragmento de código donde añade las preguntas automáticas en el chatbot de secuencia de pasos.

Para activar estas preguntas necesitamos activar el formulario, esto se hace en el fichero rules.yml. En la Figura 72 podemos ver el código necesario para crear este fichero y para activarlo:

- En la línea 161 se crea un intent llamado “Saludo”, que es necesario para la activación del formulario. Este intent se crea siempre independientemente de la plantilla que se use.
- A partir de la línea 169 se escribe en el fichero rules el código necesario para activar el formulario, usando el intent creado anteriormente para su activación
- Las últimas líneas del código corresponden a las funciones personalizadas usadas para validar los slots. Estas serán explicadas con más detalle en la sección 4.3.3 de la memoria.

```

160
161     nlu.write('- intent: Saludo\n')
162     nlu.write('  examples: |\n')
163     nlu.write('    - "Hola"\n')
164     nlu.write('    - "Buenas"\n')
165     nlu.write('    - "Buenos dias"\n')
166     nlu.write('    - "Buenas noches"\n')
167
168 # archivo rules, donde se activa y desactiva el form
169 rules.write('- rule: Activate Data Form\n')
170 rules.write('  steps:\n')
171 rules.write('    - intent: Saludo\n')
172 rules.write('    - action: data_form\n')
173 rules.write('    - active_loop: data_form\n')
174 rules.write('- rule: Submit Data Form\n')
175 rules.write('  condition:\n')
176 rules.write('    - active_loop: data_form\n')
177 rules.write('  steps:\n')
178 rules.write('    - action: data_form\n')
179 rules.write('    - active_loop: null\n')
180 rules.write('    - slot_was_set:\n')
181 rules.write('      - requested_slot: null\n')
182 rules.write('      - action: utter_submit\n')
183 rules.write('      - action: utter_slots_values\n')
184
185 domain.write('actions:\n')
186 domain.write('  - validate_data_form\n')
187

```

Figura 72: Fragmento de código donde crea el fichero rules.yml con los datos necesarios para activar el formulario.

Una vez respondidas, es necesario saber el valor de estas. Se puede ver en la Figura 73 el proceso necesario para escribir en el fichero domain.yml las respuestas con las que debe terminar la conversación el chatbot, mostrando la información recogida.

```

127 # submit --> cuando has introducido todos los slots
128 domain.write('  utter_submit:\n')
129 domain.write('    - text: "Gracias por la información."\n')
130
131 # muestro la informacion recaudada
132 domain.write('  utter_slots_values:\n')
133 domain.write('    - text: "Valores:'")
134 for e in entities_array:
135     domain.write('      {} --> {{}}'.format(e,e))
136 domain.write("\n")

```

Figura 73: Fragmento de código donde se muestran la información recogida por el chatbot de pasos en secuencia.

Ejemplo Recogida de Datos

Para este ejemplo he usado un chatbot que pide datos personales al usuario. El chatbot pregunta el nombre, teléfono y ciudad y muestra los resultados obtenidos. En este caso las

entidades serían tres: nombre, teléfono y ciudad; y la conversación se terminaría cuando tenga el valor de esas tres entidades. La Figura 74 muestra cómo sería este proceso.

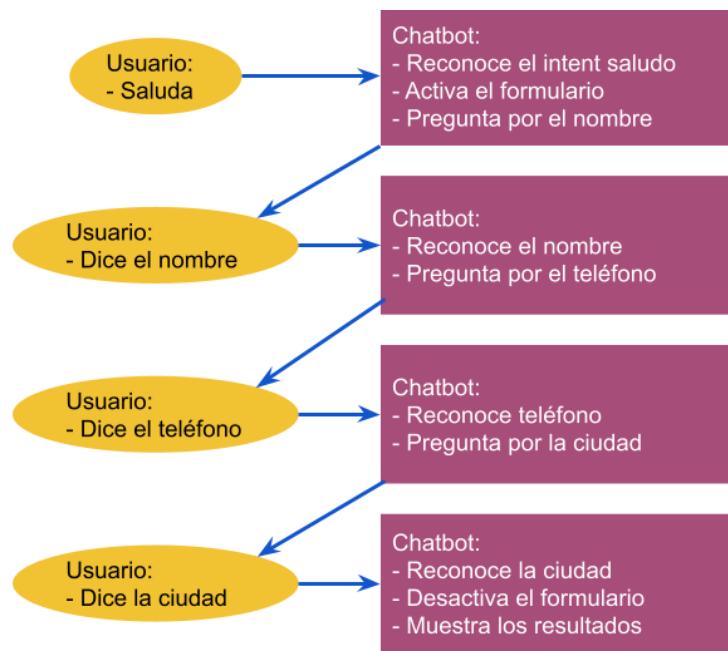


Figura 74: Funcionamiento del chatbot de recogida de datos

```
1 entities:nombre;telefono
2 entity:ciudad
3
4
5 intent:pedir_nombre;(Me llamo [Pablo] (nombre) |Me llamo [Ines] (nombre) |
6 intent:pedir_telefono;(Mi telefono es [987645374] (telefono) |Mi numero
7 intent:pedir_ciudad;(Soy de [Granada] (ciudad) |Soy de [Huelva] (ciudad) |
8
```

Figura 75: Plantilla del chatbot de recogida de datos

```

1  version: "3.0"
2  nlu:
3    - intent: pedir_nombre
4      examples: |
5        - "Me llamo [Pablo] (nombre)"
6        - "Me llamo [Ines] (nombre)"
7        - "Mi nombre es [Susana] (nombre)"
8        - "Mi nombre es [Oscar] (nombre)"
9        - "Soy [Antonio] (nombre)"
10       - "Soy [Manuela] (nombre)"
11       - "[Raquel] (nombre)"
12       - "[Macarena] (nombre)"
13       - "Es [Carlos] (nombre)"
14       - "Es [Elena] (nombre)"
15       - "[Yurena] (nombre) es mi nombre"
16       - "[Raul] (nombre) es mi nombre"
17       - "[Olga] (nombre) me llaman"
18       - "[Hernan] (nombre) me llaman"
19    - intent: pedir_telefono
20      examples: |
21        - Es [6574957983476983] (telefono)
22        - Mi telefono es [987645374] (telefono)"
23        - Mi numero es [897463728] (telefono)"
24        - Es [978124367] (telefono)"
25        - "[676357968] (telefono)"
26        - El numero de telefono es [677119826] (telefono)"
27    - intent: pedir_ciudad
28      examples: |
29        - "Soy de [Granada] (ciudad)"
30        - "Soy de [Huelva] (ciudad)"
31        - "Mi ciudad es [Sevilla] (ciudad)"
32        - "Mi ciudad es [Cáceres] (ciudad)"
33        - "Es [Guadalajara] (ciudad)"
34        - "Es [Jaén] (ciudad)"
35        - "[Almería] (ciudad)"
36        - "[Málaga] (ciudad)"
37        - ""
38    - intent: Saludo
39      examples: |
40        - "Hola"
41        - "Buenas"
42        - "Buenos días"
43        - "Buenas noches"

```

Figura 76: Archivo nlu.yml correspondiente a la plantilla del chatbot de recogida de dato

En la Figura 75 se puede ver la plantilla usada y en la figura 76 está todo el archivo generado por el programa usando la plantilla.

El siguiente caso de uso (Tabla 16) muestra el funcionamiento del programa, se limita a seguir un camino fijo. La Figura 77 presenta un ejemplo correspondiente a la conversación que este chatbot está programado para hacer.

Caso de uso	Recabar datos del usuario	CU_01
Actores	Usuario/Cliente	
Precondiciones	Haber obtenido las entidades: nombre, teléfono y ciudad	
Postcondiciones	Mostrar los resultados obtenidos.	

Tabla 16: Ejemplo caso de uso 01. Chatbot pasos en secuencia.



Figura 77: Ejemplo de conversación con el chatbot de recogida de datos

4.3.2.c. Árbol de decisión

Plantilla

La plantilla usada para crear estos chatbots contiene más elementos que las anteriores pero es más simple que su versión homónima en formato XML. El contenido se puede observar en la Figura 85 y su estructura se resume en:

- **intents** → soporta dos casos:
 - con tres elementos: iguales a los vistos en la plantilla del chatbot de preguntas frecuentes (ver Figura 62)
 - con dos elementos: el nombre y los ejemplos del intent
- **response** → contiene dos elementos: el nombre y las respuestas del chatbot
- **story** → donde hay ilimitados elementos, pero solo pueden tener tres tipos de formato:
 - Elemento normal: se toma como en la Figura 65, es decir, se escribe la línea del intent y de la acción usando el nombre del elemento (línea 157-159 de la Figura 78).
 - Elemento con paréntesis. En este caso se extraen dos parámetros: el que está fuera (se toma como una acción) y el que está dentro (como un intent) de los paréntesis. La implementación se muestra en las líneas 146-151 de la Figura 78.
 - Elemento que empieza con “*”: Se corresponde al final de la rama. He decidido marcarlo de esta manera para “decirle” al programa que es la última línea de la story. Se puede ver en las líneas 154-155 de la Figura 78.

```

1  ### primero, si hay, se ponen los intents con respuestas y ejemplos ####
2
3 intent:PreguntaInicial;(Respuestal|Otra respuestal);{Preguntal|Otra preguntal}
4
5
6
7
8  ### seguido, van los intents con respuestas ####
9
10 intent:Respuestal;{Esto es la respuestal|Esto es otra respuestal}
11 intent:Respuesta2;{Esto es la respuesta2|Esto es otra respuesta2}
12
13
14
15  ### ahora, las preguntas a las respuestas anteriores ####
16
17 response:Pregunta;({Pregunta?}|{Esto es una pregunta?})
18
19
20
21  ### penultimo, los responses con las respuestas finales (nodos hoja) ####
22
23 response:Opcion1;{Es la opcion1}
24 response:Opcion2;{Es la opcion2}
25
26
27
28  ### Y por ultimo, las stories ####
29
30 story:PreguntaInicial;Pregunta(Respuestal);*Opcion1
31 story:PreguntaInicial;Pregunta(Respuesta2);*Opcion2

```

Figura 78: Plantilla del chatbot basado en árboles de decisión

En la Figura 79 se puede ver el fragmento de código donde los stories son escritos en el archivo correspondiente.

```

139     # ramas en stories
140     for sto in stories_array:
141         ggg = sto.split(';')
142         stories.write('- story:\n')
143         stories.write('  steps:\n')
144         for g in ggg:
145             if '(' in g:
146                 params = g[g.find("(")+1:g.find(")")]
147                 tex = g.replace(params,'').replace('(','').replace(')', '')
148
149                 stories.write('    - action: utter_{}\n'.format(tex))
150                 stories.write('    - intent: {}\n'.format(params))
151                 stories.write('    - action: action_mirar_ejemplos\n')
152
153             elif g.startswith('*'):
154                 fin = g.replace('*', '')
155                 stories.write('    - action: utter_{}\n'.format(fin))
156             else:
157                 stories.write('    - intent: {}\n'.format(g))
158                 stories.write('    - action: action_mirar_ejemplos\n')
159                 stories.write('    - action: utter_{}\n'.format(g))

```

Figura 79: Fragmento del código donde se escriben las historias del chatbot basado en árboles de decisión.

Ejemplo Árbol de Fruta

Para probar este tipo de chatbot se ha usado un ejemplo sencillo con unas nueve ramas y distintas opciones de decisión.

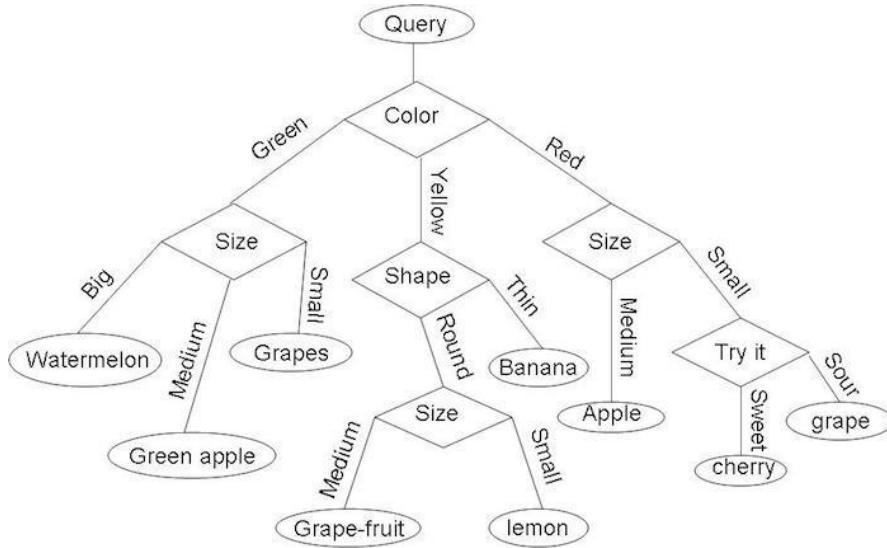


Figura 80: Diagrama usado en el chatbot basado en árboles de decisión

El ejemplo reconoce tres colores para las frutas, varios tamaños y formas. Es imprescindible seguir los caminos marcados por el grafo de la Figura 80 para que llegue a las hojas finales, por ejemplo: diciendo que la fruta es amarilla el chatbot pregunta qué forma tiene, si la persona contesta otra cosa que no sea redonda o alargada el chatbot no encontrará el camino y será necesario reiniciarlo. Hay que destacar que estas excepciones van ligadas al diseño del árbol usado para este ejemplo, otro modelo más completo, o que considere más opciones y contenga más ejemplos no tendrá tantas limitaciones.

```

1 intent:Saludo;(;Hola!|;Saludos!);{Hola,Muy buenas|Buenas|Buenos dias|Buenas noches}
2
3
4 intent:Color_Verde;{Es verde|Es de color verde|Su color es verde}
5 intent:Color_Amarillo;{Es amarilla|Es de color amarillo|Su color es amarillo}
6 intent:Color_Rojo;{Es roja|Es de color rojo|Su color es rojo}
7 intent:Tamano_Grande;{Es grande|Es de gran tamaño|Su tamaño es grande}
8 intent:Tamano_Medio;{Es medio|Es mediano|Es de tamaño medio|Su tamaño es mediano}
9 intent:Tamano_Pequeno;{Es Pequeño|Es de pequeño tamaño|Su tamaño es pequeño}
10 intent:Forma_Redonda;{Es redondo|Redondeado|Su forma es redonda|Forma redondeada}
11 intent:Forma_Alargada;{Es largo|alargado|Su forma es larga|Forma alargada}
12 intent:Sabor_Dulce;{Sabe dulce|Tiene un sabor dulce|Es dulce}
13 intent:Sabor_Amargo;{Es amargo|Tiene un sabor amargo|Sabe amargo}
14
15 response:Color;{¿De que color es tu fruta?|¿Que color tiene tu fruta?}
16 response:Tamano;{¿Cual es su tamaño?|¿Que tamaño tiene?}
17 response:Forma;{¿Cual es su forma?|¿Que forma tiene?}
18 response:Sabor;{¿Que sabor tiene?|¿Cual es su sabor?}
19 response:Sandia;{Es una sandia}
20 response:Manzana;{Es una manzana}
21 response:Uvas;{Son uvas}
22 response:Pomelo;{Es un pomelo}
23 response:Limon;{Es un limón}
24 response:Platano;{Es un plátano}
25 response:Cerezas;{Son cerezas}
26
27 story:Saludo;Color(Color_Verde);Tamano(Tamano_Grande);*Sandia
28 story:Saludo;Color(Color_Verde);Tamano(Tamano_Medio);*Manzana
29 story:Saludo;Color(Color_Verde);Tamano(Tamano_Pequeno);*Uvas
30 story:Saludo;Color(Color_Amarillo);Forma(Forma_Redonda);Tamano(Tamano_Medio);*Pomelo
31 story:Saludo;Color(Color_Amarillo);Forma(Forma_Redonda);Tamano(Tamano_Pequeno);*Limon
32 story:Saludo;Color(Color_Amarillo);Forma(Forma_Alargada);*Platano
33 story:Saludo;Color(Color_Rojo);Tamano(Tamano_Medio);*Manzana
34 story:Saludo;Color(Color_Rojo);Tamano(Tamano_Pequeno);Sabor(Sabor_Dulce);*Cerezas
35 story:Saludo;Color(Color_Rojo);Tamano(Tamano_Pequeno);Sabor(Sabor_Amargo);*Uvas
36

```

Figura 81: Plantilla del ejemplo usado en el chatbot basado en árboles de decisión

La Figura 81 muestra como es la plantilla basada en el diseño del grafo visto en la Figura 80. Observando las líneas de los story (las ramas), se puede comprobar que (tomando como ejemplo la línea 30 de la Figura 81):

1. Los elementos normales (Saludo) se corresponden a los intents con tres elementos.
2. Los elementos con paréntesis tienen la respuesta/acción (Color, Forma, Tamano) fuera del paréntesis y el intent (Color_amarillo, Forma_Redonda, Tamano_Medio) dentro de ellos.
3. Los elementos que empiezan con el carácter "*" son las hojas finales de la rama.

El resultado final de una historia se muestra en la Figura 82.

```

]- story:
  steps:
    - intent: Saludo
    - action: action_mirar_ejemplos
    - action: utter_Saludo
    - action: utter_Color
    - intent: Color_Amarillo
    - action: action_mirar_ejemplos
    - action: utter_Forma
    - intent: Forma_Redonda
    - action: action_mirar_ejemplos
    - action: utter_Tamano
    - intent: Tamano_Medio
    - action: action_mirar_ejemplos
    - action: utter_Pomelo

```

Figura 82: Muestra de cómo se hace el story correspondiente a la línea 30 de la Figura 81.

Se probaron otros diseños usando entidades y otra forma de hacer las historias pero no logré hacerlos funcionar.

Los casos de uso siguientes (Tabla 17, 18 y 19) son un ejemplo de algunas ramas de este árbol. Y las Figuras 83, 84 y 85 son los ejemplos probados en el webchat.

Caso de uso	Mi fruta es una uva	CU_01
Actores	Usuario/Cliente	
Precondiciones	Su color es verde y tamaño pequeño	
Postcondiciones	Mostrar el mensaje: "Son uvas"	

Tabla 17: Ejemplo caso de uso 01. Chatbot basado en árbol de decisión,



Figura 83: Ejemplo del caso de uso de la tabla 17 usando el webchat.

Caso de uso	Mi fruta es un pomelo	CU_02
Actores	Usuario/Cliente	
Precondiciones	Tener un color amarillo, una forma redonda y ser de tamaño medio	
Postcondiciones	Mostrar el mensaje: "Es un pomelo"	

Tabla 18: Ejemplo caso de uso 02. Chatbot basado en árbol de decisión,



Figura 84: Ejemplo del caso de uso de la tabla 18 usando el webchat.

Caso de uso	Mi fruta son cerezas	CU_03
Actores	Usuario/Cliente	
Precondiciones	Su color rojo, tamaño pequeño y sabe dulce	
Postcondiciones	Mostrar el mensaje: "Son cerezas"	

Tabla 19: Ejemplo caso de uso 03. Chatbot pasos en secuencia.



Figura 85: Ejemplo del caso de uso de la tabla 19 usando el webchat.

4.3.3. Actions.py

En esta sección se explicará brevemente la función que dicta si y un nuevo ejemplo provisto por el usuario se añade al chatbot, siempre y cuando se cumplan los parámetros dispuestos para el chatbot (sección 4.2.2, Figura 55).

```
# segun el algoritmo ...
if opcion == "Jaro-Winkler":
    actions.write("                ratio = otra.score_JW(ejemplo, comparacion)
if opcion == "LCS":
    actions.write("                ratio = otra.score_LCS(ejemplo, comparacion)
if opcion == "Coseno":
    actions.write("                ratio = otra.score_Cosine(ejemplo, comparacion)
if opcion == "BM25":
    actions.write("                ratio = otra.score_BM25(ejemplo, comparacion)
```

Figura 86: Selección del algoritmo en el constructor del fichero actions.py

En la Figura 86 podemos ver como el programa selecciona la opción del algoritmo según lo que se haya seleccionado en la interfaz. Las funciones: `score_JW`, `score_LCS`, `score_Cosine`, `score_BM25`; correspondientes a los algoritmos usados en el programa

(sección 2.3); devuelven un porcentaje con la similitud de los parámetros *ejemplo* y *comparación*.

Preguntas frecuentes y árbol de decisión

Los chatbot correspondientes a preguntas frecuentes y los basados en árboles de decisión comparten el mismo generador del fichero *actions.py*, debido a que no contienen entidades la inclusión de ejemplos se hace de la misma forma. Un fragmento de la implementación se puede ver en la Figura siguiente (ver Figura 87):

```

110     while not intent in contenido[c]:
111         c = c + 1
112
113         c = c+2
114         d = c
115         k = 0
116
117         while d<len(contenido) and not 'intent' in contenido[d]:
118             k = k + 1
119             comparacion = contenido[d] [6:]
120
121             ratio = otra.score_LCS(ejemplo, comparacion)
122
123             if ratio > limiteMAX:
124                 parecido = True
125                 break
126             elif ratio > limiteMIN and ratio < limiteMAX:
127                 parecido = False
128             else:
129                 parecido = True
130
131             d = d+1
132
133             if parecido == False and k<=maxEjem:
134                 contenido.insert(c, ' - {}\\n'.format(ejemplo))
135                 nlu.seek(0)
136                 nlu.writelines(contenido)
137                 incluido = True

```

Figura 87: Fragmento del fichero *actions.py* correspondiente a los chatbots de FAQs y árbol de decisión.

Es un bucle que empieza a buscar en la línea del intent predicho por Rasa del fichero *nlu.yml* del chatbot y acaba cuando finalizan los ejemplos del intent, o se llega al final del fichero. Las variables importantes observadas en esta Figura son:

- **intent**: Intent predicho por Rasa
- **contenido**: todo fichero *nlu.yml*
- **comparación**: ejemplos del fichero *nlu.yml*. Los que ya contiene el chatbot.
- **ejemplo**: frase dada por el usuario.
- **ratio**: porcentaje de similitud entre ejemplo y comparacion
- **limiteMIN**: límite mínimo de similitud
- **limiteMAX**: límite máximo de similitud
- **parecido**: variable boolean. Indicador principal para la añadidura del ejemplo.
- **maxEjem**: máximo de ejemplos que tiene el intent del chatbot.

Pasos en secuencia

Para el chatbot de pasos en secuencia el funcionamiento al añadir ejemplos es el mismo, pero al contener entidades hay que modificar el ejemplo dado por el usuario. Esto es debido (ver Figura 69) a que Rasa marca con “[]” y “()” las entidades, por lo tanto es necesario incluir estos caracteres a la hora de comparar.

```
ejemplo = ejemplo.replace(value, '[' + value + '](' + entity + ')')
```

Figura 88: Modificación del ejemplo para los chatbots de pasos en secuencia

En el ejemplo de la Figura 88, Rasa ya ha reconocido la entidad (guardada en entity) y su valor (value) en la frase realizada por el usuario, por lo que nos facilita su edición para incluir los caracteres antes mencionados.

Cambiando de tema, este chatbot contiene formularios, slots y entidades, que son validadas también en este archivo. El programa genera una validación para cada entidad/slots simple.

```
# función para validar cada slot
for e in entities_array:
    actions.write('\n')
    actions.write('    def validate_{}(self, slot_value: Any, dispatcher: CollectingDispatcher,
    actions.write("        intent = tracker.latest_message['intent'].get('name')\n")
    actions.write("        ejemplo = tracker.latest_message.get('text')\n")
    actions.write("        entidad = tracker.latest_message['entities'][0]['entity']\n")
    actions.write("        valor = tracker.latest_message['entities'][0]['value']\n")
    actions.write("        miClase.comprobar_ejemplo(intent, entidad, valor, ejemplo)\n")
    actions.write("    return {{'{}':slot_value}}\n\n'.format(e))
```

Figura 89: Generación de cada validación de slots

```

class ValidateDataForm(FormValidationAction):
    def name(self) -> Text:
        return 'validate_data_form'

    def validate_nombre(self, slot_value: Any, dispatcher: CollectingDispatcher):
        intent = tracker.latest_message['intent'].get('name')
        ejemplo = tracker.latest_message.get('text')
        entidad = tracker.latest_message['entities'][0]['entity']
        valor = tracker.latest_message['entities'][0]['value']
        miClase.comprobar_ejemplo(intent, entidad, valor, ejemplo)
        return {'nombre':slot_value}

    def validate_telefono(self, slot_value: Any, dispatcher: CollectingDispatcher):
        intent = tracker.latest_message['intent'].get('name')
        ejemplo = tracker.latest_message.get('text')
        entidad = tracker.latest_message['entities'][0]['entity']
        valor = tracker.latest_message['entities'][0]['value']
        miClase.comprobar_ejemplo(intent, entidad, valor, ejemplo)
        return {'telefono':slot_value}

    def validate_ciudad(self, slot_value: Any, dispatcher: CollectingDispatcher):
        intent = tracker.latest_message['intent'].get('name')
        ejemplo = tracker.latest_message.get('text')
        entidad = tracker.latest_message['entities'][0]['entity']
        valor = tracker.latest_message['entities'][0]['value']
        miClase.comprobar_ejemplo(intent, entidad, valor, ejemplo)
        return {'ciudad':slot_value}

```

Figura 90: Ejemplo de fichero actions.py correspondiente al ejemplo usado en la sección 4.3.2.b

En las Figuras 89 y 90 podemos observar cómo se genera y cómo acaba siendo la validación de un chatbot generado con el programa. La validación se limita a dar el valor recogido del usuario a la entidad correspondiente.

4.3.4. Estructura del programa

Sección dedicada a mostrar la estructura que tiene el proyecto. La carpeta que lo contiene tiene la estructura vista en la Figura 91.

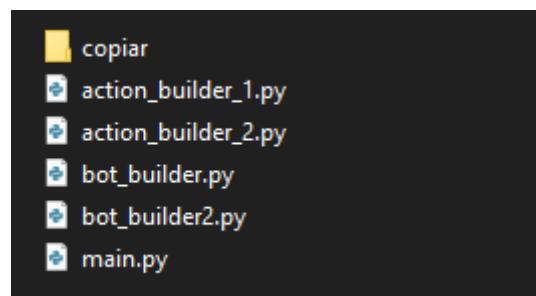


Figura 91: Estructura del programa

La interfaz se corresponde con el archivo *main.py* y es el único archivo que se debe ejecutar desde consola. Una vez abierta la interfaz el resto de archivos son usados por el programa sin que el usuario se entere, por ejemplo:

- Si seleccionamos los chatbots de preguntas frecuentes o de árbol de decisión, el programa usará el fichero *bot_builder.py* para construir la estructura del chatbot y luego empleará *action_builder1.py* para generar el fichero *actions.py* de estos chatbots.
- Eligiendo el otro tipo, el de pasos en secuencia, usará los otros dos ficheros, *bot_builder2.py* para la estructura y *action_builder2.py* para el archivo de *actions.py*.

Hay archivos que son iguales para todos los tipos de chatbots. Contenidos en la carpeta *copiar* de la Figura 91, estos archivos se copian directamente a la raíz donde se ubica el chatbot que se va a crear.

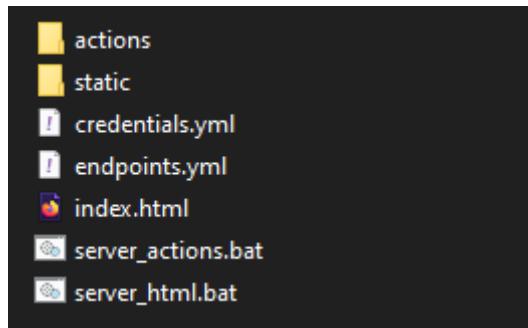


Figura 92: Composición de la carpeta copiar

Los archivos de la carpeta *copiar* son los que muestra la Figura 92:

- **actions**: es la carpeta donde se guardan las acciones que usa el chatbot, esta carpeta contiene una versión incompleta de dicho archivo. Se rellena como se ve en la sección anterior (sección 4.3.3 de esta memoria).
- **static**: son los archivos correspondientes a las funciones javascript y css del webchat.
- **credentials.yml** y **endpoints.yml**: son archivos de Rasa.
- **index.html**: archivo que contiene el webchat.
- **server_actions.bat** y **server_html.bat**: archivos de programación por lotes que ejecutan dos comandos necesarios para activar el chatbot y que funcionen correctamente. En la sección 4.4 se explica con más detallamiento su funcionamiento.

4.3.5. Uso en Telegram

Se ha implementado para el proyecto la conexión con Telegram haciendo uso de la API explicada en la sección 2.2.6 de esta memoria.

El bot creado para telegram tiene los datos que se pueden ver en la Figura 93. El nombre es **pvalenz_bot**, el access_token se genera automáticamente al crearlo y es facilitado por la herramienta @botfather.

Para la dirección url necesitamos hacer uso de la herramienta NGROK usando la orden ngrok http 5005. El puerto 5005 es necesario para activar su funcionamiento en la API de Telegram.

```
telegram:  
  access_token: "5524883291:AAHYxMscidoJ25TX_hX_-u1oLoMucaMyKxE"  
  verify: "pvalenz_bot"  
  webhook_url: "https://1d6b-85-136-44-1.ngrok.io/webhooks/telegram/webhook"
```

Figura 93: Fragmento de código necesario para activar el chatbot en Telegram

Las siguientes Figuras (94, 95 y 96) muestran ejemplos de conversaciones usando los tres chatbot creados como ejemplo (secciones 4.3.2.a, 4.3.2.b y 4.3.2.c).

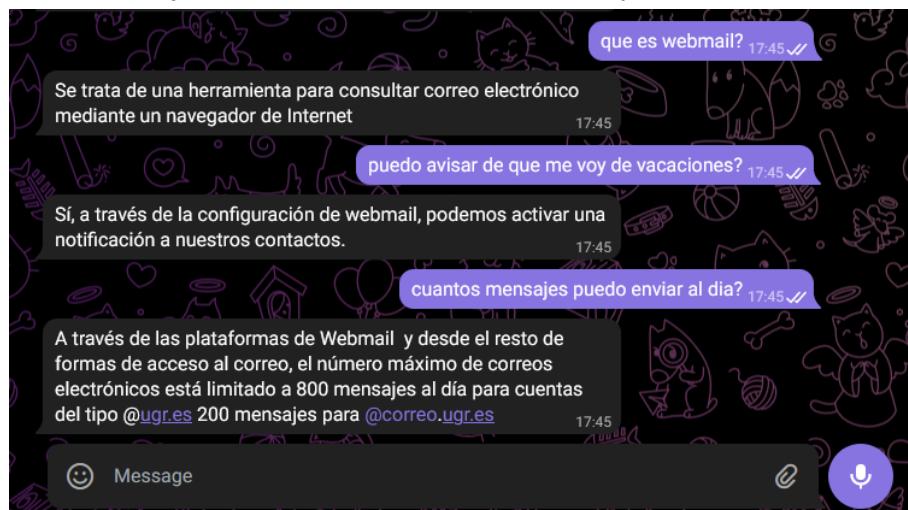


Figura 94: Conversación en Telegram Web usando el ejemplo de chatbot de Correo Ugr

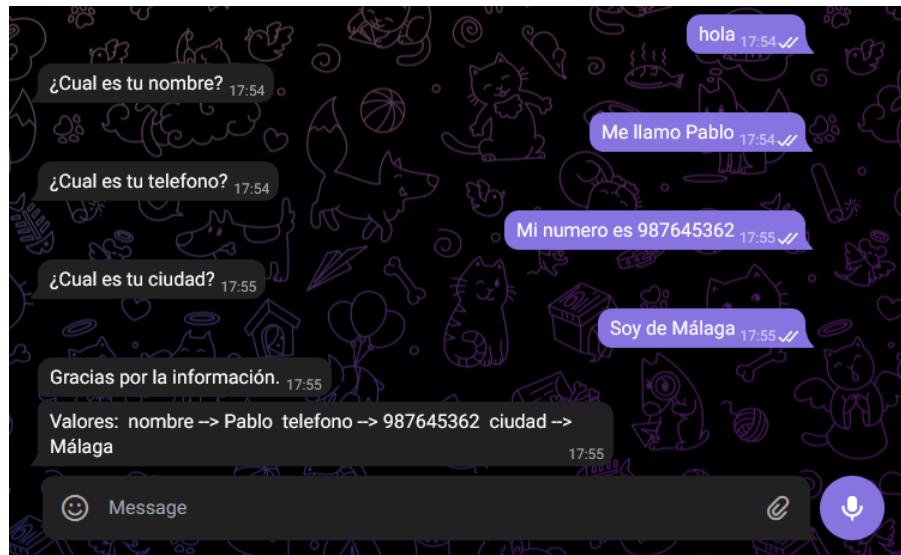


Figura 95: Conversación en Telegram Web usando el chatbot de recogida de datos

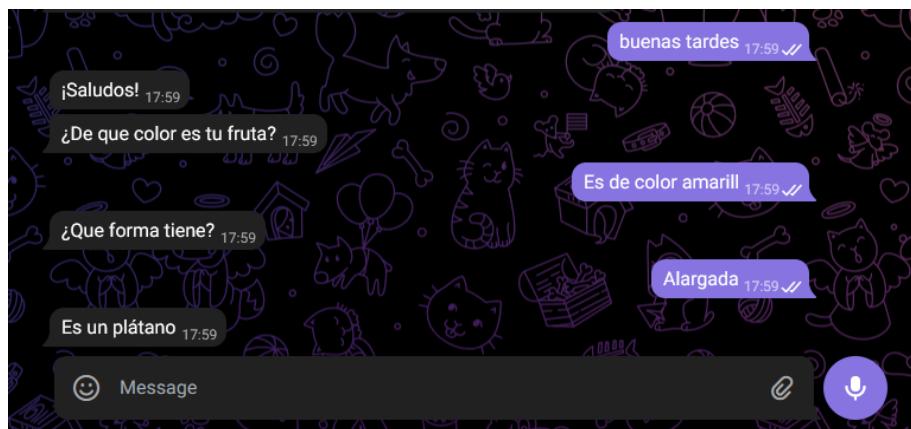


Figura 96: Conversación en Telegram Web usando el chatbot del árbol de fruta

4.4. COMPILACIÓN Y DETECCIÓN DE ERRORES

En este apartado comentaré brevemente las órdenes y comandos necesarios para hacer funcionar tanto el webchat del chatbot como el de Telegram, y más extensamente se explicará la mecánica de chequeo de plantillas implementada para la versión final del proyecto.

4.4.1. Funcionamiento

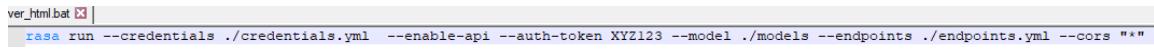
Siguiendo con el funcionamiento del chatbot, en la Figura 92 se pueden ver dos ficheros batch (*server_actions.bat* y *server_html.bat*). Cada archivo contiene una orden de línea de comandos que activan una parte necesaria para que el chatbot y el webchat funcionen correctamente. Deben ser ejecutados desde consola obviamente.



```
1 rasa run actions
```

Figura 97: Contenido archivo *server_actions.bat*

El archivo *server_actions.bat* (ver Figura 97) activa el servidor de acciones de Rasa. Este servidor hace posible que Rasa use las funciones incluidas en el fichero *actions.py*, en este caso las funciones para añadir ejemplos al chatbot y validar entidades.



```
ver_html.bat |  
rasa run --credentials ./credentials.yml --enable-api --auth-token XYZ123 --model ./models --endpoints ./endpoints.yml --cors "*"
```

Figura 98: Contenido del archivo *server_html.bat*

Por su parte, el archivo *server_html.bat* (Figura 98) activa el chatbot en sí, es decir, para que el chatbot conteste es necesario ejecutar este archivo.

Se decidió crear estos archivos para que sea más cómodo para el usuario final activar el chatbot sin necesidad de conocer las línea de comando. Cuando este quiera activar el chatbot debe ejecutar ambos archivos a la vez para que funcione correctamente.

4.4.2. Chequeo de datos

Esta actualización se añadió en la etapa final del desarrollo y solo afecta a las plantillas de entrada con texto plano. Se decidió así por que la versión de entrada XMI era más compleja para un usuario final.

La actualización se basa en una función que controla el contenido de la plantilla según el tipo de chatbot seleccionado, por ejemplo: seleccionamos el tipo "preguntas frecuentes" y la plantilla contiene líneas de entidades y responses, entonces la función detectará estas líneas y avisará del error; por consiguiente no creará el chatbot hasta que el error sea subsanado.

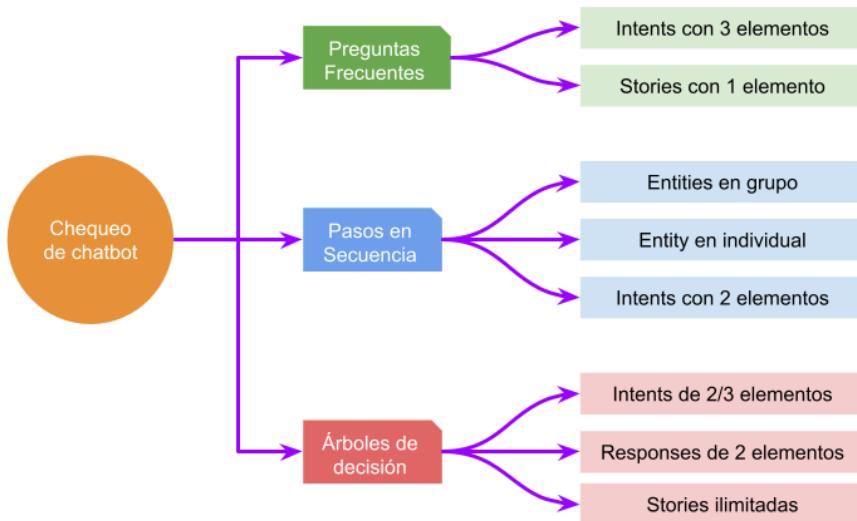


Figura 99: Diagrama con los datos que chequea la función

La función lee la plantilla proporcionada línea por línea y, según el tipo de chatbot seleccionado procede de distinta forma. En la Figura 99 se puede ver las condiciones que revisa la función para cada tipo de chatbot.

Funcionamiento para preguntas frecuentes

Primero comprueba si tiene líneas con “entities”, “entity” o “responses”. Si los detecta incluye una línea advirtiendo del error en la variable *errores* (se puede ver en la Figura 100).

```

if linea.startswith("entities") or linea.startswith("entity"):
    errores += "Error: linea {}. En el bot de Preguntas Frecuentes no son necesarias entities\n\n".format(c)

elif linea.startswith("response"):
    errores += "Error: linea {}. En el bot de Preguntas Frecuentes no son necesarios responses\n\n".format(c)

```

Figura 100: Errores al detectar entidades y responses en la plantilla de preguntas frecuentes

Para los intents comprueba (ver Figura 101):

1. Si tiene tres elementos.
2. Si contiene un elemento con paréntesis (las respuestas del chabot).
 - a. si no está vacío.
3. Si contiene otro elemento con llaves (los ejemplos del intent).
 - a. si no está vacío.
4. Si un elemento no tiene valor.

Todo lo que no cumpla la lista anterior incluirá una línea en la variable de errores.

```

elif linea.startswith("response"):
    errores += "Error: linea {}. En el bot de Preguntas Frecuentes no son necesarios responses\n\n".format(c)

elif linea.startswith("intent"):
    linea = linea.replace("intent:", "")

if len(linea.split(";")) == 3:           # para este bot, necesito que tengan tres argumentos los intents
    error = False

    for item in linea.split(";" ):
        #busco los parentesis
        if item.startswith("("):
            para = item.replace("(", "").replace(")", "")
            # miro si esta vacio
            if para == "":
                error = True
                errores += "Error: Linea{}. Parametro entre parentesis sin valor\n\n".format(c)
        #busco las llaves
        elif item.startswith("{"):
            para = item.replace("{", "").replace("}", "")
            # miro si esta vacio
            if para == "":
                error = True
                errores += "Error: Linea{}. Parametro entre llaves sin valor\n\n".format(c)
        else:
            if item == "":
                error = True
                errores += "Error: Linea{}. No hay valor\n\n".format(c)

    if not error:      # si no ha habido ningun fallo en la linea, tiene tres argumentos, uno normal, otro e
        tiene_intents = True

else:
    errores += "Error: linea {}. Error en los argumentos, chequee los separadores. El bot de Preguntas Frecuentes"

```

Figura 101: Chequeo de errores para los intents en el chatbot de preguntas frecuentes

Las historias para este chatbot contienen solo un elemento. La función, como se ve en la Figura 102, comprueba que esta condición se cumpla y no esté vacío su contenido.

```

elif linea.startswith("story"):
    linea = linea.replace("story:", "")

    if len(linea.split(";")) == 1:
        if linea == "":
            errores += "Error: Linea{}. No hay valor\n\n".format(c)
        else:
            tiene_stories = True
    else:
        errores += "Error: linea {}. Error en los argumentos, chequee los separadores."

```

Figura 102: Chequeo de errores para los stories en el chatbot de preguntas frecuentes.

En las Figuras 101 y 102 se pueden observar las variables booleanas *tiene_intents* y *tiene_stories*. Ambas se comprueban al final de la función y, según el tipo de chatbot elegido y el valor de ellas pueden añadir errores a la variable donde se guardan los errores. En este caso, habiendo seleccionado el chatbot de preguntas frecuentes, tienen que tener el valor verdadero.

Funcionamiento para secuencia de pasos

Este tipo de chatbot no contiene responses ni stories, así que si detecta líneas con ese contenido avisará del error de forma similar al de la Figura 100.

El funcionamiento para las entidades, que recordemos que hay dos formatos para su introducción en la plantilla es el mostrado en la Figura 103.

```
if linea.startswith("entities") or linea.startswith("entity"):
    if linea.startswith("entities"):
        linea = linea.replace("entities:", "")
    if not ";" in linea:
        errores += "Error: Linea{}. Error en los separadores, asegurese de que son ';'\\n\\n".format(c)
    else:
        tiene_entities = True
else:
    linea = linea.replace("entity", "")
    if ";" in linea:
        errores += "Error: Linea{}. No hacen falta separadores en esta linea\\n\\n".format(c)
    else:
        tiene_entities = True
```

Figura 103: Chequeo de errores para las entidades en secuencia de pasos

En caso de agruparlas todas las entidades en la misma línea, el programa revisa si contiene separadores necesarios, en el caso contrario mira si los tiene.

```
elif linea.startswith("intent"):
    linea = linea.replace("intent:","")

    if len(linea.split(";")) == 2:                                     # para este bot, ne
        error = False

        for item in linea.split(";"):
            #busco las llaves
            if item.startswith("{"):
                para = item.replace("{","",).replace("}", "")
                # miro si esta vacio
                if para == "":
                    error = True
                    errores += "Error: Linea{}. Parametro entre llaves sin valor\\n\\n".format(c)
                else:
                    if item == "":
                        error = True
                        errores += "Error: Linea{}. No hay valor\\n\\n".format(c)

            if not error:          # si no ha habido ningun fallo en la linea, tiene dos argumentos, uno normal y otro
                tiene_intents = True
            else:
                errores += "Error: linea {}. Error en los argumentos, chequee los separadores. El bot de Secuencia de P
```

Figura 104: Chequeo de errores para los intents en secuencia de pasos

Los intents en este chatbot contienen dos elementos, el nombre y los que vienen entre llaves. La validación es igual a la vista en la Figura 103.

Funcionamiento para árboles de decisión

Para este tipo de chatbot tiene que comprobar más cosas, pero el funcionamiento no deja de ser el mismo que los vistos en las figuras anteriores.

Los intents que reconoce el chatbot tienen 2 y 3 elementos, y el funcionamiento es similar al de las Figuras 101 y 104. Para las “responses” funciona como se ve en la Figura 105, buscando que no falten separadores y que tengan valor los elementos.

```
elif linea.startswith("response"):
    linea = linea.replace("response:", "")

    if len(linea.split(";")) == 2:
        error = False

        for item in linea.split(";"):
            #busco los parentesis
            if item.startswith("("):
                para = item.replace("(, "").replace(")", "")
                # miro si esta vacio
                if para == "":
                    error = True
                    errores += "Error: Linea{}. Parametro entre parentesis sin valor\n\n".format(c)
                else:
                    if item == "":
                        error = True
                        errores += "Error: Linea{}. No hay valor\n\n".format(c)

            if not error:      # si no ha habido ningun fallo en la linea, tiene dos argumentos,
                tiene_responses = True
            else:
                errores += "Error: linea {}. Error en los argumentos, chequee los separadores. El bot"


```

Figura 105: Chequeo de responses en el chatbot de árbol de decisión

Y para las stories, al ser ilimitadas, busca que tengan separadores. Así implica que no haya un único elemento en la rama (Figura 106).

```
elif linea.startswith("story"):
    linea = linea.replace("story:", "")

    if not ";" in linea:
        errores += "Error: Linea{}. Error en los separadores, chequee los separadores"
    else:
        tiene_stories = True
```

Figura 106: Cheque de stories en el chatbot de árbol de decisión

```

# comprobamos por ultimo si se cumplen las ultimas condiciones
if tipo == "Preguntas Frecuentes" and not tiene_intents and not tiene_stories:
    errores += "El bot de Preguntas Frecuentes necesita intents y stories"
elif tipo == "Secuencia de Pasos" and not tiene_entities and not tiene_intents:
    errores += "El bot de Secuencia de Pasos necesita entities e intents"
elif tipo == "Árbol de decisión" and not tiene_intents and not tiene_responses and not tiene_stories:
    errores += "El bot de Arbol de decision necesita intents, responses y stories"

# si no hay errores, se puede crear el bot
if errores == "":
    estado=True
else:
    sg.popup_scrolled(errores, title="Aviso")

return estado

```

Figura 107: Comprobaciones finales del chequeo de datos

Para finalizar explicar el contenido de la Figura 107, diciendo que la función es bastante estricta con las líneas que contiene la plantilla. Por ejemplo, si la plantilla de un chatbot basado en árboles de decisión tiene entidades, no deja crear el chatbot.

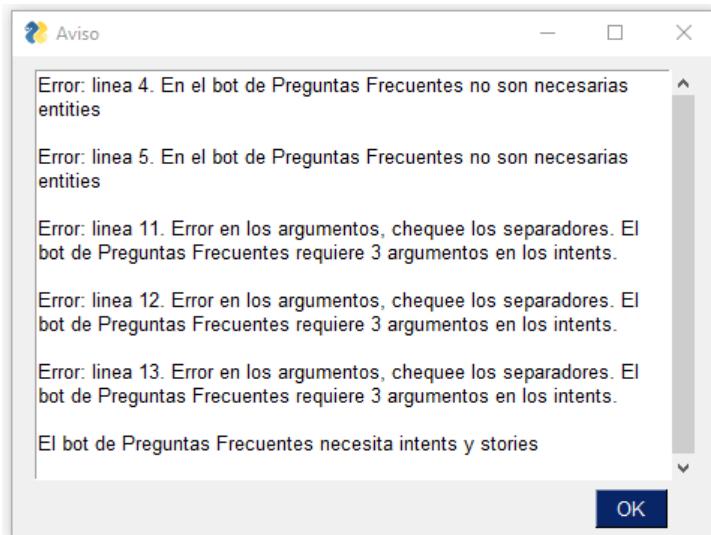


Figura 108: Aviso de errores pasando la plantilla vista en la Figura 67 a un chatbot de Preguntas frecuentes

La Figura 108 muestra una batería de fallos dados al intentar crear un chatbot de preguntas frecuentes con una plantilla para hacer chatbot de pasos en secuencia. Como contiene entidades e intents con dos elementos saltan los errores que se pueden ver.

El ejemplo siguiente (Figura 109) muestra los errores al crear con la plantilla de preguntas frecuentes (vista en la Figura 67) un chatbot basado en árboles de decisión. Al contener las historias un solo elemento muestra el error.

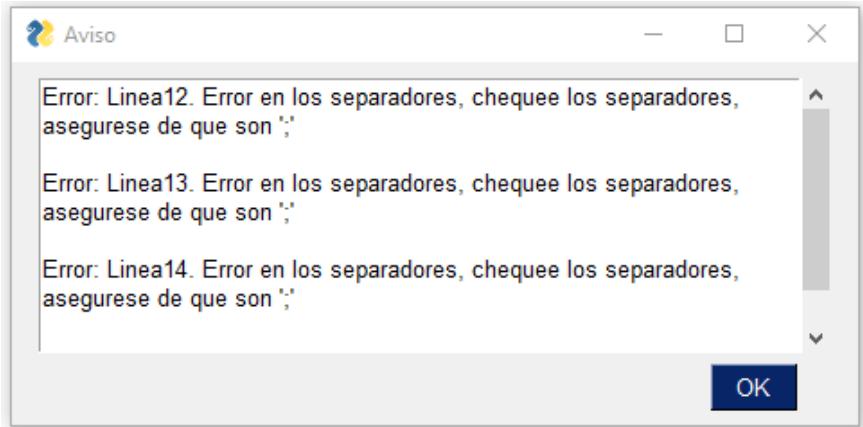


Figura 109: Aviso de errores al crear un chatbot basado en árboles de decisión con la plantilla de la Figura 67

Capítulo 5: CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo final se exponen las conclusiones alcanzadas tras la realización del proyecto, y también tras finalizar el grado de ingeniería informática. Además explicaré las posibles vías de desarrollo que podría seguir mi proyecto.

5.1. Conclusiones

Tomar la decisión de elegir este proyecto no fue fácil, había otros que me interesaban más en campos donde había hecho pequeños proyectos de clase, pero estos ya estaban asignados. Entonces escogí este, crear un chatbot. Al principio tenía un poco de miedo, esto era nuevo para mí y no sabía ni cómo empezar. Fué a partir de la primera reunión con mi tutor, donde me explicó la plataforma que usaría cuando empecé a descubrir las posibilidades que podía ofrecer y el alcance al que podría llegar.

Con la realización de este proyecto me he dado cuenta de que esta tecnología de sistemas conversacionales está empezando a ganar mucha popularidad en el mercado dado el amplio entorno al que se puede aplicar. Muchas de las más grandes compañías están invirtiendo en esto ya sea desarrollando software para la creación de sistemas de comunicación o creando dispositivos inteligentes con los que puedes mantener una conversación.

El objetivo más importante de los sistemas conversacionales es lograr “humanizarse” tanto que el usuario medio no note la ausencia de una persona al otro lado. Un objetivo que es difícil y en el que se va a trabajar en los próximos años, según las predicciones, esto significará un futuro en este nuevo sector para los desarrolladores.

Puedo concluir diciendo que empecé teniendo dudas respecto al proyecto, pero que con el tiempo me ha acabado gustando, incluso queriendo tener un futuro en este sector.

5.2. Líneas futuras

En este apartado voy a explicar ideas, o modificaciones o ampliaciones que se podrían realizar en un futuro sobre el proyecto que acabo de realizar. Estas ideas han ido apareciendo durante la realización de él y no han sido implementadas ya sea por falta de tiempo o porque ampliaban demasiado partes que no eran muy necesarias dado el tema del proyecto.

Un aspecto de lo más evidente es mejorar las interacciones del chatbot, ya sea introduciendo más estados en los que el chatbot pueda responder, o evitando que el mismo llegue a estados “raros” donde hay que reiniciarlo.

Implementar mejor los tipos de chatbot de los que dispone o ampliarlos.

En cuanto a la interfaz, se puede hacer una versión 2.0, en la que el usuario vaya introduciendo uno por uno los elementos del chatbot y pasando de una ventana a otra, el programa los irá guardando en una especie de memoria y cuando el usuario quiera, crear el chatbot. En esta versión no habría necesidad de entrada de texto.

Ampliar más las opciones de entrada de texto, por ejemplo, aceptar otro tipo de ficheros como csv.

Respecto a los algoritmos, podemos incluir más. Pero algo que creo que es más importante es el preprocesado de texto. Podemos eliminar palabras vacías de los ejemplos y/o quedarnos con la raíz de la palabra, estos cambios hacen más precisas las búsquedas en sistemas de recuperación de información, lo que adaptándolo a mi proyecto permitiría comparar mejor los ejemplos que se van añadiendo.

Pero si hablamos de cambios inmediatos, se puede realizar un chequeo de plantillas también para la versión con entrada XML y, crear manuales en español e inglés para el uso del programa.

BIBLIOGRAFÍA

- [1] Quesada Moreno, J. F., Z. Callejas Carrión, and D. Griol Barres. 2019. “Informe sobre sistemas multimodales y multilingües.”
<https://plantl.mineco.gob.es/tecnologias-lenguaje/actividades/estudios/Paginas/sistemas-conversacionales.aspx>.
- [2] Turing, A. 1950. *Computing machinery and intelligence*. Mind ed. Vol. 236. pp 433-460.
- [3] Mendez, M. A. 2014. “Un ordenador supera por primera vez en Test de Turing.”
<https://es.gizmodo.com/un-ordenador-supera-por-primera-vez-el-test-de-turing-1587840827>.
- [4] MarketsandMarkets. 2022. “Speech and Voice Recognition Market | Global forecast by 2027.” MarketsandMarkets.
<https://www.marketsandmarkets.com/Market-Reports/speech-voice-recognition-market-202401714.html>.
- [5] Ibeas, David. 2022. “Humanizar los chatbots: el reto de 2022.” Contact Center Hub.
<https://contactcenterhub.es/humanizar-los-chatbots-el-reto-de-2022-2022-15-35169/>.
- [6] Rasa, O. S. 2022. Rasa: Open source conversational AI. <https://rasa.com/>.
- [7] Rasa NLP. 2022. “Open source natural language processing (NLP).” Rasa.
<https://rasa.com/solutions/open-source-nlu-nlp/>.

[8] Rasa Docs. 2022. “Introduction to Rasa Open Source.” Rasa.

<https://rasa.com/docs/rasa/>.

[9] Rasa Integration. 2022. “Connecting to Messaging and Voice Channels.” Rasa.

<https://rasa.com/docs/rasa/messaging-and-voice-channels/>.

[10] Visual Studio Code. 2022. Visual Studio Code - Code Editing. Redefined.

<https://code.visualstudio.com/>.

[11] Lardinois, F. 2015. “Microsoft Launches Visual Studio Code, A Free Cross-Platform

Code Editor For OS X, Linux And Windows.” TechCrunch.

<https://techcrunch.com/2015/04/29/microsoft-shocks-the-world-with-visual-studio-code-a-free-code-editor-for-os-x-linux-and-windows/>.

[12] Most popular IDE. 2022. “Stack Overflow Developer Survey 2022.” Stack Overflow

Annual Developer Survey.

<https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment>.

[13] Apache Friends. 2022. XAMPP Installers and Downloads for Apache Friends.

<https://www.apachefriends.org/index.html>.

[14] NGROK. 2022. ngrok - Online in One Line. <https://ngrok.com/>.

[15] Ponce, Jesús A. 2020. “Ngrok: una herramienta con la que hacer público tu localhost de forma fácil y rápida.” SDOS.

<https://www.sdos.es/blog/ngrok-una-herramienta-con-la-que-hacer-publico-tu-localhost-de-forma-facil-y-rapida>.

[16] PySimpleGUI. 2022. “PySimpleGUI · PyPI.” PyPI. <https://pypi.org/project/PySimpleGUI/>.

[17] Driscoll, Mike. 2021. “PySimpleGUI: The Simple Way to Create a GUI With Python – Real Python.” Real Python. <https://realpython.com/pysimplegui-python/>.

[18] Rasa - Telegram Integration. 2022. “Telegram.” Rasa.
<https://rasa.com/docs/rasa/connectors/telegram/>.

[19] Telegram Bot Platform. 2015. Telegram Bot Platform.
<https://telegram.org/blog/bot-revolution>.

[20] Minidom. 2022. “xml.dom.minidom — Implementación mínima del DOM — documentación de Python - 3.10.5.” Python Docs.
<https://docs.python.org/es/3/library/xml.dom.minidom.html>.

[21] Campbell, Steve. 2022. “Python XML Parser Tutorial: Read xml file example(Minidom, ElementTree).” Guru99. <https://www.guru99.com/manipulating-xml-with-python.html>.

[22] Tan, P. N., M. Steinbach, and V. Kumar. 2005. *Introduction to Data Mining*. Addison-Wesley ed. cap. 8, pag. 500.

[23] Sidorov, G., A. Gelbukh, H. Gómez-Adorno, and D. Pinto. 2014. “Soft Similarity and Soft Cosine Measure: Similarity of Features in Vector Space Model | Sidorov.” Computación y Sistemas.
<https://cys.cic.ipn.mx/ojs/index.php/CyS/article/view/2043/1921>.

[24] Sklearn Cosine Similarity. 2022. “sklearn.metrics.pairwise.cosine_similarity — scikit-learn 1.1.1 documentation.” Scikit-learn.

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html.

[25] TFxIDF Repository. 2018.

<https://web.archive.org/web/20180115181946/http://kak.t0.org/Information-Retrieval/TFxIDF>.

[26] Brown, D. 2018. “dorianbrown/rank_bm25: A Collection of BM25 Algorithms in Python.” GitHub. https://github.com/dorianbrown/rank_bm25.

[27] Jaiswal, Nikhil. 2019. “SequenceMatcher in Python. A human-friendly longest contiguous &... | by Nikhil Jaiswal.” Towards Data Science.

<https://towardsdatascience.com/sequencematcher-in-python-6b1e6f3915fc>.

[28] Rodríguez, Daniel. 2020. “La similitud de Jaro–Winkler.” Analytics Lane.

<https://www.analyticslane.com/2020/06/24/la-similitud-de-jaro-winkler/>.

[29] Turk, J. 2020. “jamesturk/jellyfish: 🐠 a python library for doing approximate and phonetic matching of strings.” GitHub. <https://github.com/jamesturk/jellyfish>.

[30] Rokach, Lior, and Oded Maimon. 2008. *Data Mining with Decision Trees: Theory and Applications*. N.p.: World Scientific.

[31] Bayes, T. 1763. *Philosophical Transactions of the Royal Society of London. "An Essay towards solving a Problem in the Doctrine of Chances"*.

[32] Wikipedia. 2022. “Teorema de Bayes.” Wikipedia.

https://es.wikipedia.org/wiki/Teorema_de_Bayes.

[33] Chen, Hao. 2018. “A brief introduction to Chatbots with Dialogflow.” Margo.

<https://www.margo-group.com/en/news/a-brief-introduction-to-chatbots-with-dialogflow/>.

[34] Perez, S. 2017. “Amazon Lex, the technology behind Alexa, opens up to developers.”

TechCrunch.

<https://techcrunch.com/2017/04/20/amazon-lex-the-technology-behind-alexa-opens-up-to-developers/>.

[35] Amazon Lex Features. 2022. “Características de Amazon Lex: Amazon Web Services.”

Amazon AWS. <https://aws.amazon.com/es/lex/features/>.

[36] Tkinter. 2022. “tkinter — Interface de Python para Tcl/Tk — documentación de Python -

3.10.6.” Python Docs. <https://docs.python.org/es/3/library/tkinter.html>.

[37] dipesh99kumar. 2020. “Hello World in Tkinter.” GeeksforGeeks.

<https://www.geeksforgeeks.org/hello-world-in-tkinter/>.

[39] hotzenklotz. 2019. “scalableminds/chatroom: React-based Chatroom Component for

Rasa Stack.” GitHub. <https://github.com/scalableminds/chatroom>.

[40] Speech-to-text. 2022. “Language support | Cloud Speech-to-Text Documentation.”

Google Cloud. <https://cloud.google.com/speech-to-text/docs/languages>.

[41] JiteshGaikwad. 2021. “JiteshGaikwad/Chatbot-Widget.” GitHub.

<https://github.com/JiteshGaikwad/Chatbot-Widget>.

[42] FAQ correo ugr. 2022. "Preguntas Frecuentes sobre Correo electrónico | Centro de Servicios Informáticos y Redes de Comunicación." CSIRC.

<https://csirc.ugr.es/informacion/servicios/correo-electronico/preguntas-frecuentes>.