UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Análisis y aplicación de plataformas de desarrollo de chatbots

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Domínguez Coronado, Carlos

Tutor/a: Molina Marco, Antonio

CURSO ACADÉMICO: 2022/2023

Resumen

Un *chatbot* es un *software* capaz de simular la interacción entre personas gracias a una serie de patrones definidos previamente, los cuales permiten a este responder al usuario o realizar una serie de acciones en función de la conversación.

A lo largo de los años, han ido surgiendo *chatbots* de múltiples tipos para satisfacer las necesidades del momento y situación. Esto ha llevado a que, actualmente, existan una gran variedad de tipos distintos, desde los más sencillos, como pueden ser aquellos que responden a unas entradas definidas, hasta los más complejos, aquellos que hacen uso de *Machine Learning* y Procesamiento del Lenguaje Natural (PLN). El hecho de que existan múltiples tipos de *chatbots* implica que haya un gran abanico de herramientas disponibles para desarrollarlos y, a su vez, esto complica decidir que herramienta seleccionar para comenzar su desarrollo.

Es por ello por lo que, en este trabajo, se buscará realizar un análisis de las herramientas y plataformas existentes que permitan la creación de *chatbots*, estudiar tanto la posibilidad de aplicación de técnicas de PLN a estas como la integración de herramientas del mismo tipo y, por último, se seleccionará un caso de estudio para poner a prueba una de las herramientas o plataformas escogidas.

Palabras clave: *chatbot*, *software*, herramientas, *Machine Learning*, lenguaje natural.

Abstract

A *chatbot* is a *software* capable of simulating the interaction between people thanks to a series of previously defined patterns, which allow it to respond to the user or perform a series of actions depending on the conversation.

Over the years, *chatbots* of multiple types have emerged to meet the needs of the moment and situation. This has led to the current existence of a wide variety of different types, from the simplest, such as those that respond to defined inputs, to the most complex, those that make use of *Machine Learning* and Natural Language Processing (*NLP*). The fact that there are multiple types of *chatbots* means that there is a wide range of tools available to develop them and, at the same time, this makes it difficult to decide which tool to choose to start developing them.

This is the reason why, in this work, an analysis of the existing tools and platforms that allow the creation of *chatbots* will be carried out, the possibility of applying *NLP* techniques to these tools and the integration of tools of the same type will be studied and, finally, a case study will be selected to test one of the chosen tools or platforms.

Keywords: *chatbot*, *software*, tools, *Machine Learning*, natural language.

Tabla de contenidos

Tabla de ilustraciones

1. Introducción

La repetición de tareas sencillas, y no tan sencillas, lleva en nuestra sociedad desde el inicio de la civilización, lastrando o interponiéndose en su avance de múltiples maneras. Esto cambio con la invención de la máquina de vapor a manos de James Watt, ingeniero mecánico e inventor. Esto permitió automatizar procesos mediante el uso del agua, transformando la energía térmica en energía mecánica.

A partir de esta, se fueron precediendo una serie de eventos históricos que conforman la Segunda Revolución Industrial, naciendo de esta la automatización industrial, la cual permitió aplicar los conocimientos que se habían adquirido acerca de la creación de máquinas para crear sistemas en cadena capaces de emular las acciones de múltiples trabajadores trabajando de manera síncrona, relegando a estos, por otra parte, de sus puestos de trabajo, pero suponiendo un claro avance para la época y mejorando la producción de objetos, reduciendo tiempo de fabricación y ahorrando costes en el proceso.

A mediados del siglo XX, se da la Tercera Revolución Industrial, la cual se basa en la automatización y ro*boti*zación por medio de dispositivos electrónicos. Con la puesta a punto de los microprocesadores, esta tecnología se convirtió en el buque insignia del desarrollo de las tecnologías de la Tercera Revolución Industrial, al contribuir decisivamente a la disminución de los gastos energéticos y de otros materiales, así como a facilitar el desarrollo impetuoso de nuevos conocimientos [1].

Por último, a finales de la primera década de los 2000, surge lo que ahora es conocida como la Cuarta Revolución Industrial o Industria 4.0, término acuñado por el economista y empresario alemán Klaus Schwab en el Foro Económico Mundial del año 2016. Esta se caracteriza por la adopción de las Tecnologías de la Información y Comunicación (*TIC*), el Internet de las Cosas (*IoT, Internet of Things*), el Internet de los Servicios (*IoS, Internet of Services*), así como una gran cantidad de avances en otros campos relacionados con la tecnología.

La aparición de los *chatbots* marca un punto de inflexión significativo en la llamada Cuarta Revolución Industrial, particularmente en el contexto del avance tecnológico y las revoluciones industriales anteriores. Estos *chatbots*, también conocidos como *bots* (robots) conversacionales, son un avance sin precedentes en la incorporación de la automatización y la inteligencia artificial en los procesos comerciales. Su llegada, en el contexto de la transformación digital, despliega una amplia gama de aplicaciones, tanto dentro como fuera de las organizaciones actuales. Gracias a su versatilidad para gestionar diversos trámites, los *chatbots* se han convertido en una solución flexible y efectiva para la conseguir la interacción entre seres humanos y tecnología en la era de la Industria 4.0.

1.1 Motivación

Actualmente existe una gran variedad de herramientas que permiten la creación de *chatbots* más a menos complejos, todo dependiendo del uso que se le quiera dar a estos. Es por ello por lo que pueden surgir varias dudas, si bien existen muchas herramientas para crear *chatbots*, ¿cuál se debería escoger de entre todas ellas? ¿en qué se debería fijar una persona para elegir una por encima de otra?

La motivación detrás de este trabajo surge de una tarea universitaria relacionada con el procesamiento del lenguaje natural. Junto con un compañero, desarrollamos un prototipo de aplicación que permitía la gestión del correo electrónico por medio de la voz, todo ello creado gracias a la *API* de *Google*. En primera instancia, no pensamos que aquel programa que estábamos realizando fuese, a todas luces, un *chatbot*, únicamente quisimos desarrollar una forma de utilizar el correo electrónico de *Google*, *Gmail*, mediante el uso de voz para casos en los que fuese necesario leer o escribir sin tener presente un teclado físicamente.

A raíz de esto comencé a investigar acerca del tema y pude observar que el "fenómeno *chatbot*" había contribuido a la creación de todo tipo de *frameworks*, o herramientas, por parte de múltiples empresas, que permitían desarrollar estos *chatbots* de manera más o menos sencilla, con un menor o mayor grado de profundidad dependiendo de para qué se quisiese utilizar e, incluso, integrarlos en otras plataformas o webs como blogs personales e inclusive en aplicaciones de uso masivo como por ejemplo la red social Facebook. Sin embargo, el hecho de que existan muchos frameworks también puede complicar su proceso de selección y más aún si se tiene en cuenta que cada uno permite la programación en múltiples lenguajes de programación.

A lo largo de estos meses se ha podido comprobar como este fenómeno se ha visto acrecentado debido a la irrupción de *ChatGPT*, desarrollado por *OpenAI*, haciendo que, como resultado, múltiples empresas se unan a esta carrera por poseer un *chatbot* impulsado por inteligencia artificial.

1.2 Objetivos

El objetivo principal de este trabajo de investigación es diseñar una metodología para seleccionar, desarrollar e integrar herramientas de *chatbots* de manera efectiva. Esto se divide en tres subobjetivos: realizar un análisis exhaustivo del estado actual de los *chatbots*, diseñar una metodología detallada y aplicarla en un caso de estudio realista. Al lograr estos subobjetivos, se espera tener una metodología sólida que guíe la implementación exitosa de *chatbots* en diversos escenarios, mejorando la interacción con los usuarios y brindando soluciones eficientes y personalizadas en el ámbito de la atención al cliente y la comunicación automatizada.

2. *Chatbots*: Surgimiento, tipos y componentes

Antes de comenzar a valorar los *frameworks* escogidos, es necesaria una introducción al término de *chatbot*, pero antes de eso, se debe abordar el asunto de qué es un *bot*.

2.1 *Bots*

Según lo mencionado en la página web de Amazon Web Services, un *bot* es una aplicación de *software* automatizada que realiza tareas repetitivas en una red. Dicha aplicación sigue instrucciones específicas para imitar el comportamiento humano, pero es más rápida y precisa. Un *bot* también se puede ejecutar de forma independiente sin intervención humana. Por ejemplo, los *bot*s pueden interactuar con sitios web, hablar con visitantes del sitio o analizar contenidos. Aunque la mayoría de los *bot*s son útiles, hay agentes externos que diseñan *bot*s con intenciones maliciosas. Las organizaciones protegen sus sistemas de los *bot*s maliciosos y utilizan *bot*s útiles para una mayor eficiencia operativa [2].

La distinción de un *bot* de un humano a veces puede conllevar un alto grado de dificultad dependiendo de cómo haya sido diseñado este, es por ello por lo que, actualmente, existen una serie de *tests* que permiten diferenciar al humano del *bot*. Estos tests tuvieron como precursor el llamado Test de Turing, apodado así por el considerado como uno de los padres de la informática y la inteligencia artificial gracias a sus múltiples aportes a estos campos, Alan Turing. Esta prueba constaba de tres actores principales: un interrogador humano, un humano y una máquina. El primero de ellos debe tratar de distinguir al humano de la máquina, todo ello mediante el envío de información a este haciendo uso de texto y, si es incapaz de distinguirlos, se considera que la máquina ha pasado la prueba. Esta prueba ha ido evolucionando con el pasar de los años, perfeccionándose cada vez más mediante el uso de nuevas técnicas y enfoques, como lo ocurrido con los tests de Lovelace o Eugene.

2.2 *Chatbots*

Un *chatbot* es un *software*, o agente conversacional, que emula una conversación entre dos personas, ya sea mediante texto o voz, utilizando técnicas de procesamiento del lenguaje natural (*NLP*) para comprender lo que el usuario trata de transmitir y así poder responder sus preguntas y realizar acciones en consonancia con el contexto de la conversación. Los *chatbots* se rigen por una serie de reglas y algoritmos responsables de convertir la conversación entre el usuario y la máquina en una conversación coherente y fluida.

Esta capacidad de respuesta ha permitido la creación de *chatbots* con múltiples enfoques tales como la asistencia personal, como se ha comprobado con el uso de

Alexa o Siri; la realización de encuestas y atención al cliente para ofrecer contacto a cualquier hora del día, ahorrando tiempo a la administración realizando acciones simples que no necesitan intervención ni comprobación humana; la venta de productos, por medio de *chatbots* integrados en las páginas web de múltiples tipos de comercios, desde la alimentación hasta los servicios; así como un largo etcétera.

2.3 Tipos

Como se ha mencionado anteriormente, existe una gran cantidad de usos para estas tecnologías, lo que ha implicado que haya una gran cantidad de tipos específicos de *chatbots*. Sin embargo, muchos de ellos comparten base entre sí.

2.3.1 Basados en *Pattern Matching*

El *Pattern Matching* es una técnica que se encarga de buscar coincidencias entre los datos ofrecidos y patrones previamente definidos. Por tanto, la utilización de esta técnica sirve para generar respuestas adecuadas a las preguntas del usuario, en función de los tipos de coincidencia, como enunciados simples, lenguaje natural o significado semántico de las preguntas [3].

2.3.2 Basados en *Machine Learning*

El *Machine Learning* (por sus siglas, *ML*) se define como una rama evolutiva de algoritmos computacionales diseñados con el fin de emular la inteligencia humana mediante el uso de su entorno como aprendizaje [4].

2.4 Clasificación

Tantos enfoques han conllevado a la creación de múltiples tipos de *chatbots*, haciendo difícil su categorización. Sin embargo, se ha propuesto una tipología para clasificar los distintos tipos de *chatbots* existentes basada en dos dimensiones [5].

Locus of control es la primera de las dimensiones y hace referencia a quién impulsa el diálogo entre la máquina y el usuario. Por un lado, están los *chatbots* que controlan la interacción con el usuario, por lo que los diálogos están dirigidos por estos mismos (*chatbot-driven dialogue*), los cuales son altamente restrictivos e interactúan con el usuario, comúnmente, mediante el uso de menús o botones con respuestas predefinidas, pero son sencillos de realizar. Por otro lado, están los *chatbots* en los cuales el diálogo es impulsado por el usuario, o diálogos dirigidos por el usuario (*user-driven dialogue*), permitiendo más flexibilidad a la hora de su concepción, pero, a su vez, conllevando una dificultad mayor teniendo que aplicar diferentes técnicas con tal de deducir la intención del usuario.

Duration of relation es la segunda dimensión definida y hace referencia a la duración de la conversación entre el usuario y el *chatbot*. Por un lado, están los *chatbots* de *short-term relation*, o de relación de corto plazo, en los cuales no se recopila información del usuario ni de la conversación mantenida. Por otro lado, están los *chatbots* de *long-term relation*, o de relación de largo plazo, en los cuales se hace uso de la información proporcionada por el usuario para personalizar la interacción que se va a mantener, permitiendo así el uso de información a largo plazo.

En la ilustración [1] puede observarse la clasificación de cuatro ejemplos de *chatbots* realizada teniendo en cuenta ambas dimensiones. Como puede observarse, tanto los *chatbots* de soporte al consumidor como el de asistente personal están clasificados como de diálogo dirigido por el usuario y, por otra parte, los *chatbots* de gestión de contenidos y de orientador se sitúan en el extremo opuesto, en los de diálogo dirigido por el *chatbot*. A su vez, tanto el asistente personal como el orientador forman parte de los *chatbots* de relación de largo plazo, así como los de soporte al consumidor y gestión de contenidos están clasificados principalmente como de relación de corto plazo, pudiendo poseer cierto grado de retención de información.

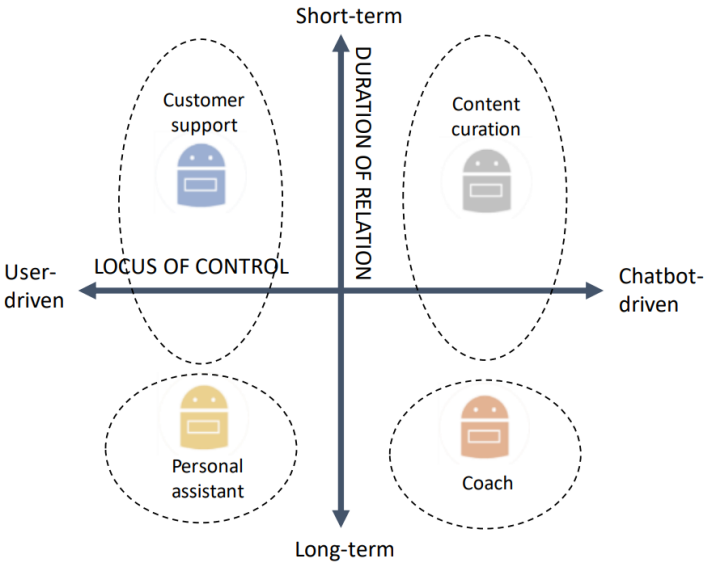

Ilustración 1. Una tipología de *chatbots* con cuatro ejemplos [6]

Nota: La imagen representa una tipología de *chatbots* mediante el uso de cuatro ejemplos, los cuales clasifica en base a *locus of control* y *duration of relation*.

2.5 Elementos principales

La implementación de un *chatbot* requiere considerar el diseño del flujo de conversación y la gestión de las diversas etapas que componen el sistema de diálogo. Los *chatbots* poseen una serie de elementos principales cuyo papel es crucial para el buen funcionamiento de estos y su integración, de forma adecuada, permite desarrollar *chatbots* eficientes y eficaces. En este punto se explorará brevemente cada uno de estos componentes con el fin de darles una descripción

sencilla pero suficientemente intuitiva, así como para aclarar sus funciones principales.

2.6 *Natural Language Processing*

Natural Language Processing (NLP) es una tecnología que se encarga de responder al usuario, comprendiendo el contenido del mensaje y actuando en consecuencia para ofrecer la acción más acertada. Suele utilizarse para entornos en los que no importa tanto como se realice la acción.

2.7 *Natural Language Understanding*

Natural Language Understanding (NLU) parte de *NLP*, pero es más específico que este. Permite entender una conversación mejor que *NLP*, puesto que se centra en estructurar el mensaje que ha recibido, aunque este no se presente de la manera más correcta (faltas de ortografía, acentos, coloquialismos, etc.), además de incluir el análisis de sentimientos para detectar el estado de ánimo el usuario.

2.8 *Natural Language Generation*

Natural Language Generation (NLG) utiliza los datos estructurados que *NLP* o NLU han recogido y se encarga de responder al usuario en lenguaje natural, ya sea mediante texto o voz.

2.9 *Pipeline*

Los *pipelines* son estructuras secuenciales que aplican diferentes etapas de procesamiento y transformación al texto del usuario en los sistemas de *chatbots* y procesamiento del lenguaje natural (*NLP*).

2.10 Tokenización

La tokenización, o segmentación, es el proceso de dividir el texto en unidades más pequeñas llamadas *tokens*. Estos *tokens* pueden ser palabras, caracteres o frases y se utilizan para realizar múltiples acciones como lo son el análisis de sentimiento, detección de entidades o la generación de respuestas. Al segmentar el texto, la tokenización facilita el análisis y procesamiento del lenguaje natural en etapas posteriores. En las ilustraciones 2 y 3 puede observar un ejemplo de como se realiza la tokenización.

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize

text = "La tokenización, o segmentación, es el proceso de" + \
 " dividir el texto en unidades más pequeñas llamadas tokens" + \
 ". Estos tokens pueden ser palabras, caracteres o frases y" + \
 " se utilizan para realizar múltiples acciones como lo son" + \
 " el análisis de sentimiento, detección de entidades o la" + \
 " generación de respuestas. Al segmentar el texto, la" + \
 " tokenización facilita el análisis y procesamiento del" + \
 " lenguaje natural en etapas posteriores."

print(sent_tokenize(text))
print(word_tokenize(text))
```

Ilustración 2. Código de ejemplo de tokenización con nltk [Elaboración propia]

Nota: La imagen muestra un ejemplo de tokenización haciendo uso de la librería nltk.

```
['La tokenización, o segmentación, es el proceso de dividir el texto en unidades más pequeñas
 llamadas tokens.', 'Estos tokens pueden ser palabras, caracteres o frases y se utilizan para
 realizar múltiples acciones como lo son el análisis de sentimiento, detección de entidades o
 la generación de respuestas.', 'Al segmentar el texto, la tokenización facilita el análisis
y procesamiento del lenguaje natural en etapas posteriores.']

['La', 'tokenización', ',', 'o', 'segmentación', ',', 'es', 'el', 'proceso', 'de', 'dividir',
 'el', 'texto', 'en', 'unidades', 'más', 'pequeñas', 'llamadas', 'tokens', '.', 'Estos', 'tok
ens', 'pueden', 'ser', 'palabras', ',', 'caracteres', 'o', 'frases', 'y', 'se', 'utilizan', '
para', 'realizar', 'múltiples', 'acciones', 'como', 'lo', 'son', 'el', 'análisis', 'de', 'sen
timiento', ',', 'detección', 'de', 'entidades', 'o', 'la', 'generación', 'de', 'respuestas',
'.', 'Al', 'segmentar', 'el', 'texto', ',', 'la', 'tokenización', 'facilita', 'el', 'análisis
', 'y', 'procesamiento', 'del', 'lenguaje', 'natural', 'en', 'etapas', 'posteriores', '.']
```

Ilustración 3. Salida de ejemplo de tokenización [Elaboración propia]

Nota: La imagen muestra el resultado de la ejecución del código de la ilustración [2], pudiendo diferenciarse el resultado de sent_tokenize() y word_tokenize().

2.11 *Stemming*

El *stemming* es un proceso lingüístico que se utiliza para reducir las palabras a su forma truncada, eliminando prefijos y sufijos. A través de algoritmos y reglas lingüísticas, el *stemming* busca simplificar las palabras a su forma más simple y común, sin tener en cuenta la estructura gramatical o el contexto. Esta técnica es útil para agrupar palabras similares y reducir la dimensionalidad del texto, lo que facilita el análisis y la comparación de palabras. En las ilustraciones 4 y 5 puede observarse un ejemplo de aplicar *stemming* a un texto.

```python
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('spanish')

text = "El stemming es un proceso lingüístico que se utiliza" + \
    " para reducir las palabras a su forma raíz o base, eliminando" + \
    " prefijos y sufijos. "
words = word_tokenize(text)

print("")
for w in words:
    print(w, " : ", stemmer.stem(w))
print("")
```

Ilustración 4. Código de ejemplo de stemming con nltk [Elaboración propia]

Nota: La imagen muestra un ejemplo de stemming haciendo uso de la librería nltk.

```
El  :  el
stemming  :  stemming
es  :  es
un  :  un
proceso  :  proces
lingüístico  :  lingüist
que  :  que
se  :  se
utiliza  :  utiliz
para  :  par
reducir  :  reduc
las  :  las
palabras  :  palabr
a  :  a
su  :  su
forma  :  form
raíz  :  raiz
o  :  o
base  :  bas
,  :  ,
eliminando  :  elimin
prefijos  :  prefij
y  :  y
sufijos  :  sufij
.  :  .
```

Ilustración 5. Salida de ejemplo de stemming [Elaboración propia]

Nota: La imagen muestra el resultado de la ejecución del código de la ilustración [4]. Puede apreciarse como se realiza el proceso de reducción de una palabra a su forma truncada.

2.12 Lematización

La lematización es un proceso que busca reducir las palabras a su forma base, conocida como lema. A diferencia del *stemming*, que únicamente elimina sufijos para obtener una forma truncada de una palabra, la lematización busca obtener la forma canónica, o raíz, de una palabra. Esto implica tener en cuenta la gramática y la morfología del idioma para obtener el lema adecuado. La lematización es útil para reducir las variaciones de palabras a su forma base, lo que facilita la comparación y el análisis de textos, como puede observarse en las ilustraciones 6 y 7.

```
import stanza
nlp = stanza.Pipeline('es')

doc = nlp("La lematización es un proceso que busca reducir las" + \
    " palabras a su forma base, conocida como lema.")

for sentence in doc.sentences:
    for word in sentence.words:
        print(word.text, ' :' ,word.lemma)
```

Ilustración 6. Código de ejemplo de lematización con stanza [Elaboración propia]

Nota: La imagen muestra un ejemplo de lematización haciendo uso de la librería stanza.

```
La  : el
lematización  : lematización
es  : ser
un  : uno
proceso  : proceso
que  : que
busca  : buscar
reducir  : reducir
las  : el
palabras  : palabra
a  : a
su  : su
forma  : forma
base  : base
,  : ,
conocida  : conocido
como  : como
lema  : lema
.  : .
```

Ilustración 7. Salida de ejemplo de lematización [Elaboración propia]

Nota: La imagen muestra el resultado de la ejecución del código de la ilustración [6], pudiendo observar cómo se realiza el proceso de reducción de las palabras a su forma raíz.

2.13 *Stopwords*

Las *stopwords* son palabras que se consideran comunes y no aportan un valor significativo al texto. Estas palabras, como artículos, preposiciones y conjunciones, se eliminan durante el procesamiento del lenguaje natural para mejorar el rendimiento del análisis de texto, pudiendo verse un ejemplo de esto en las ilustraciones 8 y 9. Eliminar las *stopwords* puede ayudar a mejorar la precisión y eficiencia de los algoritmos de procesamiento de texto, así como su selección puede variar según el idioma y el contexto.

```python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
nltk.download('stopwords')

stop_words = nltk.corpus.stopwords.words('spanish')

text = "Las stopwords son palabras que se consideran comunes y no" + \
    " aportan un valor significativo al texto. "

word_tokens = word_tokenize(text)

filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]
filtered_sentence = []

for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)

print('- Texto original:\n', word_tokens,'\n')
print('- Texto sin stopwords:\n', filtered_sentence)
```

Ilustración 8. Código de ejemplo de stopwords con nltk [Elaboración propia]

Nota: La imagen muestra un ejemplo de eliminación de stopwords haciendo uso de la librería nltk.

```
- Texto original:
 ['Las', 'stopwords', 'son', 'palabras', 'que', 'se', 'consideran', 'comunes',
'y', 'no', 'aportan', 'un', 'valor', 'significativo', 'al', 'texto', '.']

- Texto sin stopwords:
 ['Las', 'stopwords', 'palabras', 'consideran', 'comunes', 'aportan', 'valor',
'significativo', 'texto', '.']
```

Ilustración 9. Salida de ejemplo de stopwords [Elaboración propia]

Nota: La imagen muestra el resultado de la ejecución del código de la ilustración [8]. Puede observarse como se realiza la eliminación de palabras que no aportan un valor significativo.

2.14 Elementos de diálogo

Los elementos de diálogo desempeñan un papel importante en la construcción de interacciones entre el usuario y la máquina. Estos elementos, que incluyen los *intents*, *entities*, *slots*, actions, stories, *rules* y *utterances*, son los pilares que permiten al *chatbot* comprender y responder de manera precisa a las necesidades del usuario.

2.15 Intents

Los *intents*, o intenciones, sirven para representar la intención del usuario detrás del mensaje. Estos son definidos previamente para ajustarse a las necesidades del usuario. Un ejemplo de estos sería el siguiente:

La secretaría de una universidad hace uso de un *chatbot* para formalizar reservas de aulas y para consultar los horarios asignados a cada una. En este caso, el *chatbot* tiene dos *intents*: reservar aula y consultar horario del aula.

Por tanto, si el usuario comenta “Desearía reservar el aula 1” el *chatbot* asociará este mensaje con el *intent* “reservar aula”. Para ello debe haber sido entrenado

previamente con múltiples entradas similares para que el *chatbot* pueda deducir con mayor probabilidad de acierto lo que el usuario necesita.

2.16 Entities

Las *entities*, o entidades, son palabras clave de un mensaje que sirven para dar contexto de la conversación y permiten al *chatbot* responder en base a su valor.

Basándonos en el anterior ejemplo, en la frase “Desearía reservar el aula 1” la *entity* es “aula” (*entity*: tipo de sala).

2.17 Slots

Los *slots*, o ranuras, permiten almacenar el valor de las entidades dentro del contexto de una conversación, por lo que permiten estructurar la información de manera eficiente para así poder utilizarla posteriormente.

Por tanto, el valor de tipo de sala es “aula”.

2.18 Actions

Las *actions*, o acciones, son elementos fundamentales en los sistemas de procesamiento del lenguaje natural, encargados de definir respuestas y ejecutar tareas específicas en función de las intenciones del usuario. Representan componentes programados que permiten generar respuestas personalizadas, realizar consultas en bases de datos, interactuar con servicios externos y completar acciones diversas.

Un ejemplo de *action* es el siguiente:

Usuario: "Desearía reservar el aula 1."

Action: ReservarAula1

2.19 Stories

Las *stories*, o historias, son ejemplos de conversaciones que pueden mantener el usuario con el *chatbot* y que permiten entrenar a este último para responder de manera correcta dependiendo del flujo de la conversación.

Un ejemplo de *story* es el siguiente:

Usuario: "Desearía reservar el aula 1."

Chatbot: "¡Claro! ¿En qué fecha y horario te gustaría reservar el aula?"

Usuario: "Quisiera reservarla para mañana de 10:00 a 12:00."

Chatbot: "Perfecto. El aula 1 ha sido reservada para mañana de 10:00 a 12:00."

En este ejemplo, el usuario pide reservar un aula, el *chatbot* entiende que la acción de reservar está asociada a la entidad "aula", por lo que procede a pedir los datos faltantes al usuario tal y como marca la *story* predefinida.

2.20 Rules

Las *rules*, o reglas, describen detalladamente cual debe ser el flujo de la conversación, en determinadas partes de esta, detallando paso a paso el *intent* principal desde el que se parte y las acciones que este va a desencadenar. Son útiles cuando no se precisa utilizar una lógica de procesamiento más compleja y, en su defecto, se apuesta por una respuesta estática.

Rule: Si el mensaje del usuario contiene la frase "reservar el aula 1".

Action: El *chatbot* confirma la disponibilidad del aula 1 y solicita más detalles sobre la fecha y horario de reserva.

De este modo, regla verifica si el mensaje del usuario contiene la frase "reservar el aula 1" y, si es así, el *chatbot* responde confirmando la disponibilidad y solicita más detalles sobre la fecha y horario de reserva.

2.21 Utterances

Las *utterances*, o expresiones, permiten entrenar al *chatbot* con las diferentes formas posibles de expresar una intención, permitiendo a este comprender y clasificar las intenciones del usuario, identificando posibles patrones.

Por tanto, varios ejemplos con los que entrenar al sistema podrían ser los siguientes:

a) "Quiero hacer una reserva para el aula 1."

b) "¿Es posible reservar el aula 1 para la próxima semana?"

c) "Desearía reservar el aula 1 durante dos horas."

3. Metodología para la creación de *chatbots*

La metodología que se va a presentar a continuación debe servir como una guía para el proceso de desarrollo de *chatbots*. Esta metodología proporciona un marco que abarca desde la definición de objetivos y requisitos hasta la implementación, pruebas y mejora continua. A través de una serie de pasos bien definidos, se busca garantizar la efectividad, funcionalidad y calidad del *chatbot*, optimizando la experiencia del usuario y maximizando el valor que aporta a este.

La Ilustración 10 muestra el ciclo de desarrollo que se debería seguir haciendo uso de la información proporcionada por el usuario, pudiendo observarse un modelo en cascada con la capacidad de volver a fases anteriores en el caso de que sea necesario.

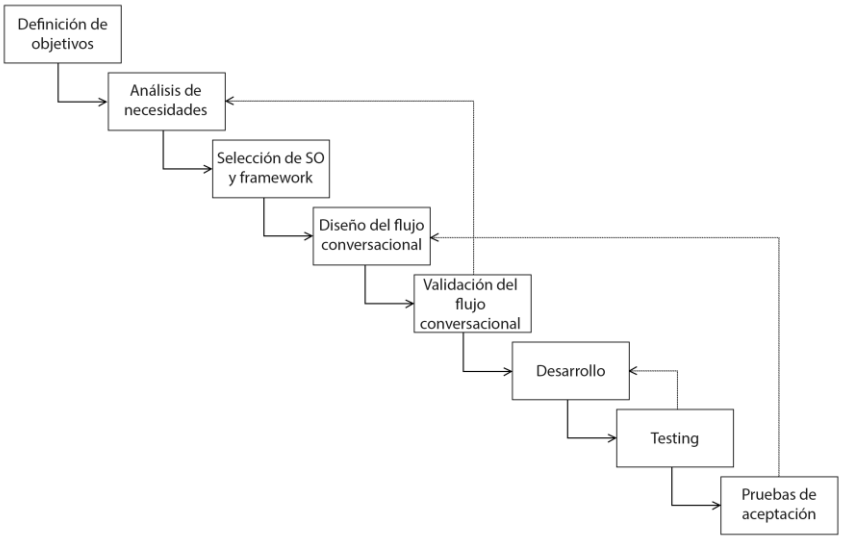

Ilustración 10. Ciclo de desarrollo [Elaboración propia]

Nota: La imagen muestra el ciclo de desarrollo en cascada de un *chatbot*, en el cual pueden observarse las distintas fases en las que se puede realizar un *rollback* con tal de realizar ajustes o corregir problemas de desarrollo o concepción.

3.1 Definición de objetivos

Como paso primordial, es necesaria la definición de los objetivos y propósitos para la creación de un *chatbot*. Se busca identificar el propósito, o idea, central del *chatbot*, así como los problemas que buscará resolver esta solución.

3.2 Análisis de necesidades

Se realizará un análisis de las necesidades del usuario, identificando las posibles funcionalidades que será necesario programar. Para ello, se realizará una recopilación de los requisitos y posibles escenarios que puedan darse, desde el lado del usuario, con respecto a la necesidad de obtener información sobre un punto en específico.

3.3 Selección del sistema operativo y *framework*

Una vez definido el propósito y el tipo del *chatbot* que va a desarrollarse, es necesaria la selección tanto del sistema operativo como del *framework*, debido a que algunos *frameworks* pueden no encontrarse disponibles en determinados sistemas operativos.

Es importante saber qué tipo de *chatbot* se tiene en mente desarrollar (asistente virtual, atención al cliente, etcétera), ya que esto puede condicionar la selección de un *framework* por encima de otro. Como se ha visto anteriormente, es posible la creación de un *chatbot* basado en *Pattern Matching*, *Machine Learning* o un híbrido de ambos.

La selección del *framework* va a ser determinante, ya no solo para la creación del *chatbot*, si no como también para la implementación de componentes, así como la integración con servicios externos tales como bases de datos o sistemas de autenticación.

3.4 Diseño del flujo conversacional y funcionalidades

En base a las necesidades del usuario, se realizará el diseño del flujo conversacional, seleccionando los *intents* necesarios para cumplir con los objetivos, así como también las *entities* asociadas a cada *intent*. Se realizarán, además, historias de ejemplo con tal de comprobar que se está tomando la dirección correcta.

Dependiendo del *framework* seleccionado, se podrá utilizar diferentes lenguajes de programación, *frameworks* y librerías para llevar a cabo la implementación. Por ejemplo, si se opta por un enfoque basado en reglas, será necesario definir las reglas y patrones que permitan reconocer las intenciones del usuario y generar respuestas en bases a estas. Si, por el contrario, se opta por el uso de *Machine Learning*, se deberán entrenar y ajustar los modelos de procesamiento del lenguaje natural utilizando conjuntos de datos de relevancia.

3.5 Validación del flujo conversacional

Tras el diseño del flujo conversacional y funcionalidades, se deberán validar con el fin de comprobar su validez, así como para poder identificar posibles errores, mejoras o adiciones a las funcionalidades establecidas inicialmente. Por tanto, es

posible que durante esta etapa sea necesario realizar un *rollback* a la etapa de análisis de necesidades

3.6 Desarrollo

La siguiente etapa corresponde a la implementación de los distintos elementos propios de los chatbots, funcionalidades programables, preparación de los datos de entrenamiento y entramiento del modelo.

Durante la implementación de los elementos, se hará uso de la información extraída del diseño del flujo de la conversación, así como de su validación, para programar los distintos elementos como los *intents*, *entities*, *rules* y *stories*. Además, se implementarán las distintas funcionalidades que corresponden a acciones externas a la lógica del chatbot.

Una vez escogidos los datos que van a ser mostrados al usuario, es necesaria su preparación, lo cual conlleva la limpieza de estos datos con técnicas de preprocesamiento, como la tokenización o la eliminación de *stopwords*, de forma que el texto resultante siga una estructura clara y adecuada para entrenar el modelo. Tras esto, los datos deben separarse en tres conjuntos de datos que cumplirán funciones específicas tal y como indican su nombre: entrenamiento, validación y prueba. Esta fase no es exclusiva de los *chatbots* basados en *Machine Learning*, puesto que para los basados en *Pattern Matching* también debe prepararse el conjunto de datos, sin embargo, los primeros deberán entrenarse para poder ofrecer respuestas más elaboradas.

A continuación, los datos obtenidos son utilizados para entrenar el modelo de formas diversas, por lo que, dependiendo del *framework* escogido, este entrenamiento suele realizarse de manera supervisada o automática.

3.7 *Testing* y depuración

Es fundamental realizar pruebas y evaluaciones continuas durante el desarrollo para asegurarse de que el *chatbot* funcione correctamente. Esto implica someterlo a diferentes escenarios y casos de uso, con el objetivo de verificar su capacidad. Asimismo, deben llevarse a cabo pruebas de integración con otros sistemas o servicios externos para confirmar que el *chatbot* funcione correctamente.

3.8 Pruebas de aceptación

Una vez realizado el testing en cuestión, con el fin de comprobar la correcta funcionalidad de los elementos y funcionalidades, será necesaria la realización de unas pruebas de aceptación, en las cuales se comprobará que los resultados obtenidos concuerdan con aquellos objetivos que han sido marcados al comienzo del ciclo, pudiendo retroceder a la fase de diseño del flujo conversacional para corregir problemas o desajustes que se hayan podido cometer.

4. Herramientas escogidas

El proceso de selección de herramientas ha consistido en la elección de un conjunto de cuatro herramientas siendo estas Amazon Lex, Dialogflow, Rasa y Chatterbot. El criterio para la selección de estas herramientas se basó en su popularidad y en la disponibilidad de documentación y recursos de apoyo de los cuales se pueden beneficiar los desarrolladores.

Debido a que existen herramientas que hacen uso de interfaces gráficas para ayudar a los desarrolladores de manera visual, así como también existen otras herramientas que disponen únicamente de línea de comandos y editores de texto externos, se decidió escoger dos de cada tipo para poder realizar un análisis parejo.

A continuación, se explicarán brevemente cada una de las herramientas escogidas para introducirlas en el proyecto.

4.1 Amazon Lex

Amazon Lex es una herramienta desarrollada por Amazon que permite la creación de chats conversacionales mediante el uso de texto y/o voz. Amazon Lex hace uso del motor conversacional de Alexa, lo cual le permite realizar análisis profundos sobre la información de entrada para comprender el fin de esta gracias a su comprensión del lenguaje natural (ver Ilustración 11).

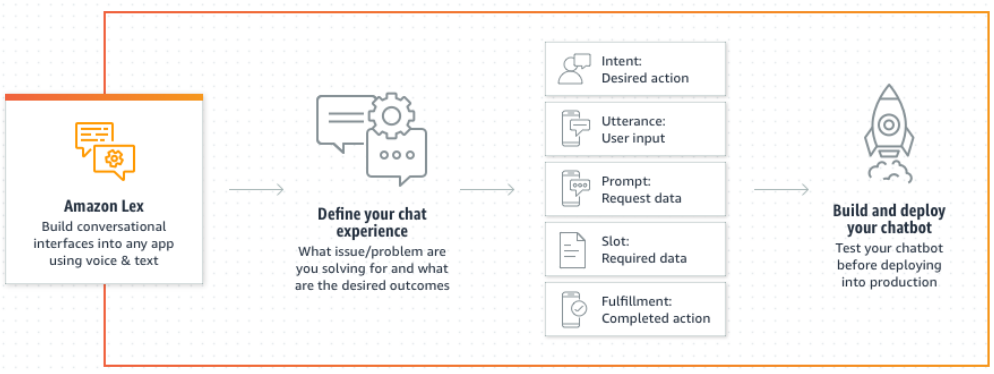

Ilustración 11. Diagrama de flujo de Amazon Lex [7]

Nota: La imagen corresponde a un diagrama de flujo muy simplificado de Amazon Lex y puede observarse el flujo de creación de *chatbots* en esta plataforma.

La particularidad de Amazon Lex es que hace uso de varios servicios de AWS: AWS Lambda, Amazon Cognito y Amazon Polly.

AWS Lambda es un servicio de Amazon que permite ejecutar funciones previamente programadas. Lambda ejecuta el código en una infraestructura de computación de alta disponibilidad y realiza todas las tareas de administración de los recursos de computación, incluido el mantenimiento del servidor y del sistema

operativo, el aprovisionamiento de capacidad y el escalado automático, así como las funciones de registro [8].

Amazon Cognito es un proveedor de autenticación que permite el registro e inicio de sesión de los usuarios, así como llevar un mantenimiento de estos de manera efectiva. Su implementación es sencilla puesto que basta con añadir unas pocas líneas de código al *software* en cuestión, siendo posible su implementación en varias plataformas (Objective C para iOS, Android, Swift para iOS, React Native y Aplicaciones web).

Amazon Polly es un servicio que permite la conversión de texto en voz y que hace uso de técnicas de *Machine Learning* para ofrecer una mayor calidad de voz.

4.2 Rasa

Rasa es un marco de aprendizaje automático de código abierto para crear *chatbots* conversacionales. Se utiliza para automatizar asistentes de texto y voz.

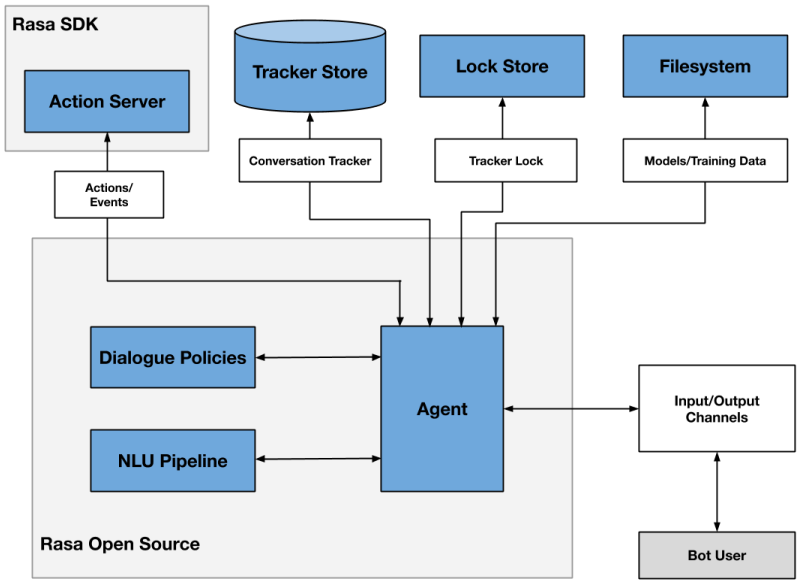

Ilustración 12. Arquitectura de RASA [9]

Nota: La imagen corresponde a la arquitectura de RASA. En dicha imagen, pueden observarse múltiples componentes, entre los cuales destacan los componentes relacionados con NLU y la gestión de diálogo.

Rasa ha sido creado con la intención de ser altamente personalizable para poder ser utilizado a diferentes niveles (ver Ilustración 12), desde un proyecto propio hasta su uso a nivel empresarial, es por ello por lo que un gran número de empresas importantes hacen uso de Rasa de maneras diversas: la compañía telefónica Orange lo utiliza para proporcionar un soporte técnico multicanal y automatizado, el operador T-Mobile para permitir a los clientes ahorrar tiempo y saltarse la cola con una experiencia guiada y personalizada para cada uno de ellos, la compañía de seguros Helvetia para la automatización del proceso de reclamación de seguros, así como un sinfín más de empresas.

4.3 Dialogflow

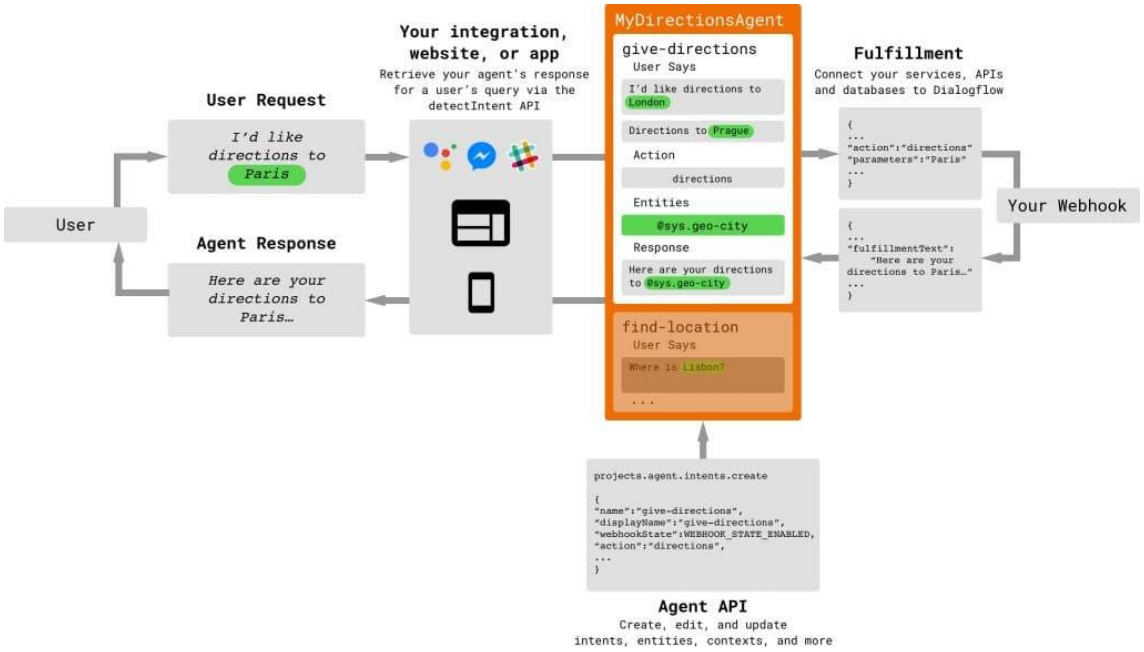

Ilustración 13. Diagrama de flujo de Dialogflow [10]

Nota: La imagen corresponde al diagrama de flujo de Dialogflow. En esta imagen se observa el flujo del proceso que lleva a cabo Dialogflow cuando recibe una respuesta del usuario, destacando el módulo Fulfillment y su interacción con los *webhook*.

Dialogflow es una plataforma con comprensión del lenguaje natural que te facilita el diseño de una interfaz de usuario de conversación y su integración a tu aplicación para dispositivos móviles, aplicaciones web, dispositivos, *bot*s, sistemas de respuesta de voz interactiva y más [11]. En la Ilustración 13 puede observarse el diagrama de flujo de Dialogflow, pudiendo diferenciar claramente las distintas fases de este.

Este se divide en dos servicios de agentes virtuales, Dialogflow ES y Dialogflow CX, estando Dialogflow ES orientado a agentes pequeños y de poca a moderada complejidad y, por otra parte, estando Dialogflow CX orientado a su contraparte, agentes grandes y con una gran complejidad.

4.4 Chatterbot

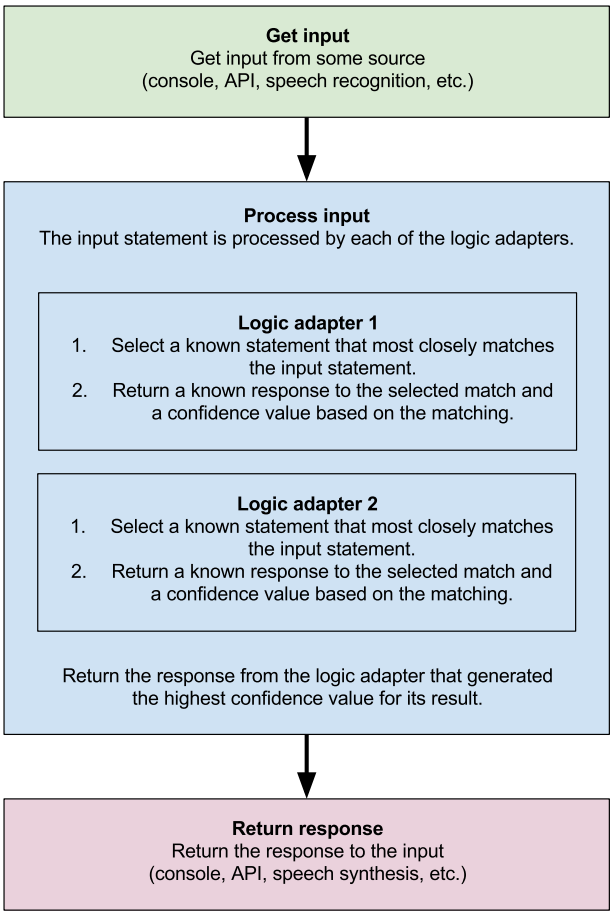

Ilustración 14. Diagrama de flujo de Chatterbot [12]

Nota: La imagen corresponde al diagrama de flujo de Chatterbot. En esta imagen se observa el flujo del proceso de Chatterbot, el cual se divide en tres grupos: obtener entrada, procesar entrada y devolver respuesta.

Chatterbot es una librería de Python concebida para crear *chatbots* basados en reglas, mayormente, o en aprendizaje automático, siendo más adecuada su utilización para proyectos simples debido a su simplicidad (ver Ilustración 14).

Si bien Chatterbot es especialmente adecuada para proyectos de baja complejidad, es importante destacar que también es una librería de código abierto, lo cual significa que los desarrolladores tienen acceso al código fuente de esta y pueden contribuir a su desarrollo [12].

5. Evaluación y análisis de las herramientas

5.1 Evaluación

Dentro del ámbito de la Ingeniería de *Software*, se pueden distinguir diversas características relevantes con respecto a los productos *software*. En este trabajo, se llevará a cabo una selección de las más relevantes, con el objetivo de llevar a cabo un análisis minucioso y detallado de los *frameworks* escogidos. De esta manera, se podrá obtener una aproximación precisa para determinar la herramienta más adecuada en cada caso. Para este propósito, se tomará como referencia el estándar ISO/IEC 25010 de la Ilustración 13, el cual se centra en la calidad de los productos de *software*. Este estándar proporciona un marco sólido que servirá como base para evaluar y comparar las características de los *frameworks* seleccionados.

Ilustración 15. ISO/IEC 25010 [13]

Nota: Diagrama perteneciente al estándar ISO/IEC 25010 relativo a la calidad de los productos *software*. Se encuentra dividido en ocho características y estas, a su vez, anidan sus propias subcaracterísiticas.

Para realizar las mediciones de las métricas para cada *framework*, se otorgará una puntuación de 1 a 5 a cada subcaracterística, donde cada puntuación corresponderá al siguiente grado de satisfacción de esta (ver Tabla 1).

Tabla 1. Puntuación para el grado de satisfacción de subcaracterísiticas

Grado de satisfacción	Puntuación
Muy deficiente	1
Insuficiente	2
Aceptable	3
Bueno	4
Excelente	5

De cara a la realización de este estudio se han escogido las características externas del estándar ISO/IEC 25010, debido a que estas características se centran en los aspectos perceptibles y cuantificables, desde la perspectiva del usuario,

abarcando elementos como la funcionalidad, la facilidad de uso, la seguridad, entre otros. Es por ello por lo que, en el contexto de características externas, las características en las que se va a centrar este trabajo son las siguientes: adecuación funcional, eficiencia de desempeño, compatibilidad, usabilidad, fiabilidad, seguridad y portabilidad.

De estas características externas se han escogido, minuciosamente, una serie de subcaracterísticas debido a que no todas las subcaracterísticas recogidas en las ISO/IEC 25010 son de utilidad real para este estudio, así como otras se consideran imprescindibles en un *software* que se encuentra actualmente en el mercado. Un ejemplo de subcaracterística que no ha sido incluida en el conjunto es madurez, la cuál es la capacidad del sistema de ejecutarse de manera correcta y fiable bajo condiciones normales. Como puede observarse, esta subcaracterística es más propia de estudios realizados sobre *software* que todavía no se encuentra en el mercado, ya que es una métrica esencial para que este pueda ser utilizado.

Por tanto, en la Tabla 2 se ha agrupado toda la información perteneciente a las características que van a ser evaluadas, sus subcaracterísticas, una descripción breve pero representativa de cada una de estas, así como una serie de consideraciones o preguntas para poder realizar la evaluación de las diferentes características que ofrecen cada uno de los distintos *frameworks*.

Tabla 2. Consideraciones a tener en cuenta para la evaluación de herramientas

Características externas ISO/IEC 25010	**Subcaracterísticas**	**Descripción**	**Consideraciones de evaluación**
Adecuación Funcional	Completitud funcional	Grado en que un sistema o *software* cubre todas las funcionalidades requeridas o esperadas por los usuarios.	Ofrece todas las funcionalidades necesarias para satisfacer los requisitos establecidos.
			Permite la creación de flujos de conversación complejos con múltiples opciones.
			Ofrece soporte para la integración de servicios externos y sistemas *backend*.
Fiabilidad	Disponibilidad	Capacidad del sistema de estar operativo cuando el usuario lo precise.	Ofrece mecanismos para garantizar una disponibilidad total del *chatbot*.
			Proporciona opciones de escalado automático para adaptarse a aumentos de carga repentinos y evitar interrupciones en el servicio.
			Incluye herramientas de supervisión y alerta para identificar y abordar problemas de disponibilidad.
Seguridad	Integridad	Capacidad del sistema de prevenir accesos y modificaciones de información de forma no autorizada.	Garantiza que la información y las respuestas proporcionadas por el *chatbot* sean confiables.
			Ofrece mecanismos de detección y prevención de modificación de datos.
			Permite la verificación de integridad de los mensajes enviados y recibidos.
	Autenticidad	Capacidad del sistema de confirmar la integridad de un recurso, así como la identidad de un usuario.	Proporciona métodos para autenticar y verificar la identidad de los usuarios.
			Permite la integración con servicios de autenticación externos.
			Incluye opciones para la implementación de protocolos de seguridad para garantizar la autenticidad de las solicitudes y respuestas.
	Responsabilidad	Capacidad del sistema de rastrear información o acciones llevadas a cabo por un usuario.	Ofrece herramientas y funciones para el seguimiento y registro de las interacciones del *chatbot*.
			Permite la revisión de acciones realizadas tanto por el *chatbot* y como por los usuarios.
			Proporciona opciones para gestionar la privacidad y el consentimiento de los usuarios en cumplimiento con las regulaciones de protección de datos.

5.2 Análisis

A continuación, se presenta una evaluación detallada de las herramientas mencionadas anteriormente en relación con las características mencionadas en la ISO/IEC 25010.

Mediante el uso de la tabla de métricas (ver Tabla 2), se compararán y evaluarán diferentes aspectos de cada herramienta para obtener una visión clara de sus fortalezas y debilidades. Este análisis permitirá tomar decisiones en base a sus características, con tal de seleccionar la herramienta más adecuada para el proyecto en cuestión.

5.2.1 Amazon Lex

5.2.1.1 Completitud funcional

Amazon Lex es una herramienta que permite la creación de *chatbots* gracias a diferentes tecnologías de procesamiento de lenguaje natural y aprendizaje automático, por lo que, a mayor número de usuarios, mejor capacidad de comprender las consultas ofrecidas por el usuario.

Además, gracias a su diseñador visual, es posible crear *bot*s de manera sencilla, además de crear flujos de conversación complejos y tarjetas de respuesta, las cuales permiten mostrar al usuario una tarjeta en la que puede incluir fotos, enlaces o texto, así como una serie de opciones a elegir en función de la pregunta del usuario, pudiendo ser la conversación con este más interactiva [14].

En cuanto a la integración con servicios externos, Amazon Lex no ofrece soporte nativo para *webhooks*, pero mediante AWS Lambda es posible configurar una función Lambda para que actúe como un *webhook*, pudiendo así recibir datos y comunicarse con otras herramientas [15].

5.2.1.2 Disponibilidad

Gracias a la infraestructura de AWS, que se compone de regiones y zonas de disponibilidad, la disponibilidad del servicio está garantizada [16] y, gracias a que Amazon Lex es un servicio totalmente administrador, el escalado es automático, por lo que no precisa de un escalado manual por parte del usuario [17].

Amazon CloudWatch permite la supervisión de las aplicaciones del usuario pudiendo recopilar información referida a la disponibilidad de estas, así como de los recursos de los que hagan uso, con tal de poder eliminar los posibles errores que surjan [18].

5.2.1.3 Integridad

Amazon Lex proporciona herramientas para garantizar la confiabilidad de las respuestas, pero siempre teniendo en cuenta los datos de entrenamiento que han sido utilizados.

En cuanto a la monitorización de los datos para prevenir su modificación, así como para verificar su integridad, AWS no dispone de tales funcionalidades, si bien sí posee una herramienta llamada AWS CloudTrail que permite la monitorización y el seguimiento de los eventos de la cuenta de AWS en cuestión [19]. Además, Amazon Lex no ofrece funcionalidades que permitan la verificación de integridad de los mensajes enviados y recibidos.

5.2.1.4 Autenticidad

Si bien Amazon Lex no dispone de métodos específicos para autenticar al usuario, sí dispone de un servicio de AWS llamado Amazon Cognito que permite agregar autenticación, autorización y gestión de usuarios [20].

Por otro lado, Amazon Lex brinda la opción de garantizar la seguridad y autenticidad de las solicitudes y respuestas a través de medidas de seguridad estándar en el entorno de Amazon Web Services, aunque no dispone de funcionalidades propias de la herramienta.

5.2.1.5 Responsabilidad

Así como con la disponibilidad, es posible utilizar Amazon CloudWatch para supervisar los registros pertenecientes a eventos y conversaciones. En estos registros se hallan tanto las solicitudes de los usuarios como las respuestas generadas por el *bot*, así como otros eventos de relevancia [18].

Amazon Lex no proporciona opciones específicas con respecto a la gestión de la privacidad y el consentimiento de los usuarios, por lo que es desarrollador debe ser el encargado de implementarlo teniendo en cuenta la legislación vigente.

5.2.2 Rasa

5.2.2.1 Completitud funcional

Rasa dispone de una gran variedad de herramientas y funcionalidades que le permiten la creación de *chatbots* de distinta índole y complejidad. Estas funcionalidades van desde el uso del procesamiento del lenguaje natural, hasta el uso de directivas para guiar el flujo de la conversación, así como también permite la integración con otras herramientas externas para expandir sus capacidades. Es por ello por lo que Rasa es una herramienta potente, válida para cualquier tipo de proyecto.

5.2.2.2 Disponibilidad

Como bien se ha comentado en el punto perteneciente a la subcaracterística Capacidad, Rasa no ofrece mecanismos de escalado automático en función de su necesidad, así como tampoco puede garantizar la disponibilidad total del bot, si no que esto recae únicamente de los servidores en los que se encuentre alojada la instalación de Rasa.

5.2.2.3 Integridad

Si bien Rasa tiene la capacidad para garantizar que las respuestas emitidas por el *chatbots* tengan cierta confiabilidad, esta confiabilidad recae mayormente en los datos de entrenamiento que el usuario ha proporcionado. Esto significa que Rasa no tiene la capacidad de corregir información errónea que haya sido aportada en su fase de entrenamiento.

A su vez, esta herramienta no dispone de mecanismos que permitan prevenir la modificación de datos, si no que deberá ser una herramienta externa la que se encargue de estas comprobaciones, así como tampoco es capaz de garantizar la integridad de los mensajes enviados y recibidos.

5.2.2.4 Autenticidad

Rasa se centra exclusivamente en la construcción de *chatbots* y no es una herramienta diseñada para la autenticación de los usuarios, por lo que no posee métodos internos para ello. Es por ello por lo que la única forma de realizar estas comprobaciones en la versión gratuita es mediante la integración de métodos externos. Sin embargo, la versión X/Enterprise sí que posee la posibilidad de creación de usuarios, así como su inicio de sesión.

5.2.2.5 Responsabilidad

Rasa dispone de una clase llamada Tracker que permite mostrar un seguimiento de los mensajes entre el *chatbot* y el usuario, así como los *slots* que han sido rellenados durante la conversación o la información referida a un mensaje en concreto, como por ejemplo su *intent* y *entities* [21].

Sobre la privacidad de los usuarios, Rasa no recopila información que pueda resultar sensible, como puede ser información personal que permita identificar al usuario, datos de entrenamiento o los mensajes enviados y recibidos [22].

5.2.3 Dialogflow

5.2.3.1 Completitud funcional

Dialogflow dispone de todas las capacidades y funcionalidades necesarias para la creación de *chatbots* competentes y de gran potencia. Bajo la misma premisa y como ocurre en Amazon Lex con las tarjetas de respuesta, Dialogflow dispone de respuestas enriquecidas con tal de disponer de una conversación más interactiva con el usuario [23].

En cuanto a la integración con servicios externos y sistemas *backend*, Dialogflow dispone de traducción automática, ofrecida por Google, a numerosos idiomas. Además, dispone del módulo Fulfillment, el cual permite interactuar con *webhooks* de manera sencilla y eficaz para comunicarse con servicios externos y sistemas *backend* [24].

5.2.3.2 Disponibilidad

Dialogflow no dispone de mecanismos que permitan una disponibilidad total de servicio, pero, la infraestructura global de Google sumado a los acuerdos de nivel de servicio (*SLA*) ofrecen una alta disponibilidad y unos tiempos de servicio bien definidos [25].

Si bien no dispone de un escalado automático en función de la demanda, es posible configurar Dialogflow junto con Compute Engine para realizar su escalado de manera automática por medio de un proceso detallado en su documentación [26]. Además, es posible la utilización de Google Kubernetes Engine, herramienta la cual permite implementar Dialogflow en clústeres de Kubernetes con el fin de aprovechar el escalado automático que proporciona esta [27].

Con respecto a la monitorización de la disponibilidad del servicio, mediante Google Cloud Monitoring es posible supervisar tanto la disponibilidad de este como la utilización de recursos con tal de identificar posibles errores que surjan [28].

5.2.3.3 Integridad

Como ocurre con las diferentes herramientas de creación de *chatbots*, estas ofrecen la información con la que han sido entrenadas, por tanto, la confiabilidad de las respuestas recae en la calidad de los datos con los que ha sido entrenada.

De igual manera, Dialogflow no tiene la capacidad ni herramientas propias necesarias para evitar que los datos que posee se vean modificados por terceros, así como tampoco puede verificar la integridad de los mensajes recibidos, si no que esta responsabilidad recae sobre los servicios que ofrece Google Cloud Platform, los cuales permiten dotar de cierta seguridad tanto el *bot* como sus datos. Algunas de estas herramientas son Google Cloud Security Command Center, Google Cloud Key Management Service y Google Cloud Identity-Aware Proxy [29].

5.2.3.4 Autenticidad

Dialogflow dispone de una amplia gama de servicios propios y externos para identificar al usuario, entre los cuales se encuentran Google Sign-In, OAuth 2 y Firebase Authentication.

No dispone de opciones de seguridad propias para garantizar la autenticidad de las solicitudes, pero sí es posible utilizar servicios externos, como los mencionados anteriormente, para garantizar esta funcionalidad.

5.2.3.5 Responsabilidad

Mediante Google Cloud Logging es posible acceder a los registros referentes a las interacciones entre el usuario y el *chatbot*, pudiendo realizar un análisis detallado de las interacciones con el fin de mejorar la interacción con este [30].

Con respecto a la regulación de los datos de carácter personal, Dialogflow permite, mediante el uso de entidades, seleccionar los datos que van a ser recopilados. Sin embargo, es importante que el desarrollador implemente avisos sobre la concesión de datos, así como la no recopilación de datos de carácter sensible.

5.2.4 Chatterbot

5.2.4.1 Completitud funcional

Chatterbot es una librería creada con el fin de satisfacer las necesidades mínimas para la creación de un *chatbot* de baja complejidad. Es por ello por lo que, si bien tiene las funcionalidades necesarias para ello, es posible que pueda ser insuficiente en casos en los que se precise una mayor complejidad.

Por otro lado, *Chatterbot* tiene la funcionalidad intrínseca de integrarse con Django's ORM, siendo Django un *framework* de alto nivel y de código abierto diseñado para simplificar el proceso de desarrollo de aplicaciones web [31]. Sin embargo, *Chatterbot* no tiene la capacidad de integrarse con otros servicios externos, si no que esto recae directamente en la decisión del desarrollador siendo posible agregar esta funcionalidad debido a que *Chatterbot* es una librería de Python y es posible utilizar las múltiples librerías que existen en este lenguaje de programación para este fin.

5.2.4.2 Disponibilidad

Al igual que en las herramientas anteriormente vistas, *Chatterbot* no es capaz de asegurar la disponibilidad total de su servicio, si no que esta responsabilidad recae en los servidores en los que se encuentre alojada, así como tampoco proporciona herramientas de supervisión.

Al ser una librería y no una herramienta, al igual que ocurre con Rasa Open Source, *Chatterbot* no es capaz de escalar su tamaño automáticamente en función de su necesidad.

5.2.4.3 Integridad

Como ha podido comprobarse en herramientas anteriores, garantizar que la información y las respuestas proporcionadas por el *chatbot* sean confiables depende de los datos de entrenamiento con los que ha sido entrenado, así como tampoco es posible comprobar la integridad de los mensajes enviados y recibidos ya que *Chatterbot* no dispone de tales funcionalidades al no ser una herramienta de monitoreo, si no una librería.

5.2.4.4 Autenticidad

Con respecto a la autenticación de usuarios, *Chatterbot* no dispone de funcionalidades referentes a la autenticación y gestión de usuarios, si no que esta funcionalidad debe ser proporcionada por servicios externos como OAuth o la autenticación multifactor.

5.2.4.5 Responsabilidad

Chatterbot dispone de funcionalidades para leer las conversaciones entre el usuario y el *chatbot*, así como es posible diseñar una estructura para almacenar este registro en base de datos o en archivos de registro, por lo tanto, esta capacidad no se encuentra incluida automáticamente en *Chatterbot*, sino que es necesaria su implementación.

Y, con respecto a la gestión de la privacidad y el consentimiento de los usuarios, *Chatterbot* no dispone de funcionalidades que cumplan con el Reglamento General de Protección de Datos, si no que será el desarrollador el encargado de implementar las funcionalidades pertinentes.

5.3 Tabla de calificaciones

Tras llevar a cabo una evaluación exhaustiva de las herramientas mencionadas, se ha elaborado una tabla de calificaciones sobre la cual se ha puntuado cada subcaracterística en base a las fortalezas y debilidades de cada herramienta haciendo uso de la puntuación definida en la Tabla 1.

Tabla 3. Tabla de calificación

Subcaracterísticas	Consideraciones de evaluación	Amazon Lex	Rasa	Dialogflow	Chatterbot
Completitud funcional	Ofrece todas las funcionalidades necesarias para satisfacer los requisitos establecidos.	5	5	5	3
	Permite la creación de flujos de conversación complejos con múltiples opciones.	5	5	5	2
	Ofrece soporte para la integración de servicios externos y sistemas backend.	4	5	4	3
Disponibilidad	Ofrece mecanismos para garantizar una disponibilidad total del chatbot.	5	1	5	1
	Proporciona opciones de escalado automático para adaptarse a aumentos de carga repentinos y evitar interrupciones en el servicio.	5	1	4	1
	Incluye herramientas de supervisión y alerta para identificar y abordar problemas de disponibilidad.	5	1	5	1
Integridad	Garantiza que la información y las respuestas proporcionadas por el chatbot sean confiables.	3	3	3	3
	Ofrece mecanismos de detección y prevención de modificación de datos.	4	1	3	1
	Permite la verificación de integridad de los mensajes enviados y recibidos.	1	1	1	1
Autenticidad	Proporciona métodos para autenticar y verificar la identidad de los usuarios.	4	4	5	3
	Permite la integración con servicios de autenticación externos.	4	4	5	3
	Incluye opciones para la implementación de protocolos de seguridad para garantizar la autenticidad de las solicitudes y respuestas.	5	4	5	3
Responsabilidad	Ofrece herramientas y funciones para el seguimiento y registro de las interacciones del chatbot.	5	4	5	2
	Permite la revisión de acciones realizadas tanto por el chatbot y como por los usuarios.	5	4	5	2
	Proporciona opciones para gestionar la privacidad y el consentimiento de los usuarios en cumplimiento con las regulaciones de protección de datos.	3	5	3	2

6 Herramienta escogida para el caso de estudio

Tras reconsiderar las distintas opciones y ventajas que tiene el uso de cada una de las anteriores herramientas, se ha seleccionada Rasa como la herramienta idónea para llevar a cabo un caso de estudio relacionado con la temática del turismo. Para dicho caso de estudio, se busca la implementación de dos casos de uso concretos.

El primer caso de uso está relacionado con la búsqueda de vuelos. El usuario pedirá al chatbot una lista de vuelos a partir de un origen, un destino y una fecha, por lo que este debe ser capaz de extraer todas las entidades necesarias para realizar la búsqueda mediante una API de viajes.

El segundo caso de uso se relaciona con la búsqueda de rutas. El usuario proporcionará la provincia donde quiere buscar rutas de senderismo que realizar y será el chatbot el encargado de realizar esta búsqueda en bases de datos o *datasets*.

De esta manera, se evaluará la capacidad del chatbot de trabajar con múltiples entradas y repositorios de información, buscando facilitar al usuario la búsqueda que desee extrayendo toda la información que esté disponible y sea de utilidad.

6.1 Elección

El hecho de haber escogido Rasa como la herramienta a utilizar se debe a una serie de factores expuestos a continuación.

6.1.1 Flexibilidad y personalización

Rasa Open Source proporciona un alto grado de flexibilidad y personalización para crear *chatbots*, siendo esto fundamental para lograr un enfoque altamente específico como lo es la provisión de información relativa a desplazamientos y rutas.

6.1.2 Control Total del Flujo de Conversación

La capacidad que posee Rasa de gestionar el flujo de conversación de manera contextual y orientada a objetivos permite que al *chatbot* interactuar de manera natural y fluida con los usuarios, siendo esencial para proporcionar respuestas coherentes y útiles, y viéndose más limitada en *Chatterbot*.

6.1.3 Integración de Acciones Personalizadas

La funcionalidad de Acciones Personalizadas (*Custom Actions*) en Rasa permite la integración con fuentes externas, como APIs de viaje y conjuntos de datos de rutas [32]. Esto facilita la obtención y entrega de información actualizada y precisa a los usuarios.

6.1.4 Open Source y Comunidad Activa

Rasa cuenta con una gran comunidad activa de desarrolladores, una documentación completa y una gran variedad de tutoriales, lo cual permite que el desarrollo de una aplicación con Rasa no se vea pausado gravemente debido a problemas que puedan surgir durante su desarrollo.

6.1.5 Costo Efectivo

A diferencia de herramientas como Amazon Lex y Dialogflow, Rasa Open Source ofrece una solución gratuita y de código abierto, lo que es importante en proyectos que no puedan permitirse el pago de estos servicios.

6.2 Componentes

Entrando más en detalle en los componentes que posee Rasa y que van a ser utilizados para el desarrollo del caso de estudio, se encuentran: Rasa NLU (*Natural Language Understanding*) y Rasa Core.

6.2.1 Rasa NLU

NLU (*Natural Language Understanding*) es la parte de Rasa que realiza la clasificación de intenciones, la extracción de entidades y la recuperación de respuestas [33].

La cadena de procesos de NLU se define dentro del archivo “config.yml” y define los pasos que seguirá Rasa para detectar los *entities* e *intents*. Existen una serie de componentes dentro de esta cadena de procesos, o *pipeline*, que permiten extraer los *intents* del texto.

En primer lugar, del texto que se pretende analizar se extraen todas las palabras, pasando a tener una lista de *tokens*, por medio del *tokenizer*. Algunos de estos *tokenizers* aplican también lematización, concluyendo en una lista de *tokens* similar a la anterior, pero con la diferencia de que los *tokens* resultantes resultan en las formas canónicas, o lemas, de los *tokens* originales (ver Ilustración 16).

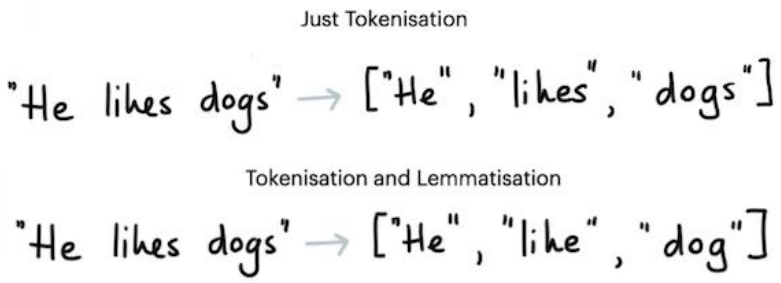

Ilustración 16. Ejemplos de tokenización y lematización [34]

Nota: La imagen corresponde al resultado que surge de aplicar la tokenización y la lematización.

En segundo lugar, los *featurizers* generan características numéricas para los modelos de aprendizaje automático (ver Ilustración 17).

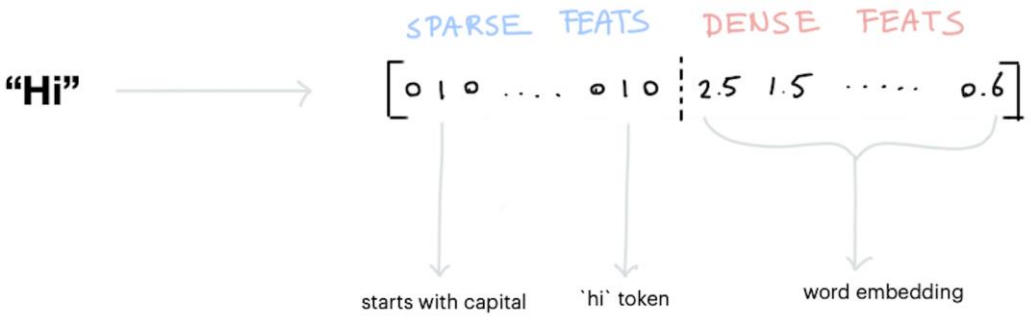

Ilustración 17. Ejemplo de featurizer [34]

Nota: La imagen corresponde al resultado de aplicar un *featurizer* a una palabra concreta.

Dentro de Rasa, los *featurizers* más utilizados son los siguientes: RegexFeaturizer, CountVectorFeaturizer, SpacyFeaturizer y LangageModelFeaturizer.

En tercer lugar, los clasificadores de intenciones utilizan las características conseguidas de los *tokens* anteriores para clasificar los *tokens* en función de su intención. Desde Rasa, se recomienda el uso de su modelo DIET.

DIET es una arquitectura de transformadores multitarea que gestiona conjuntamente la clasificación de intenciones y el reconocimiento de entidades. Ofrece la posibilidad de conectar y reproducir varias incrustaciones preformadas como BERT, GloVe, ConveRT, etc [35]. El hecho de que se recomiende el uso de DIET, si bien los grandes modelos ya entrenados no se recomiendan para construir aplicaciones conversacionales, es porque es una arquitectura modular que se adapta al *workflow*, es paralela a los modelos lingüísticos ya entrenados y es seis veces más rápido de entrenar que un modelo normal.

Si bien antes Rasa utilizaba en su *pipeline* un *bag-of-words*, ahora con DIET el orden de las palabras importa, por lo que es capaz de ofrecer un mayor rendimiento (ver Ilustración 18).

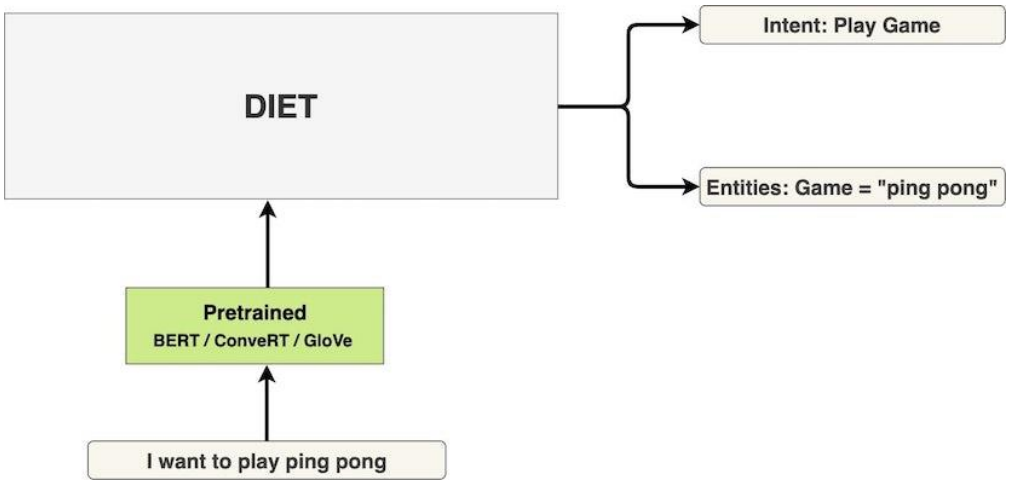

Ilustración 18. Ejemplo de uso de DIET [36]

Nota: La imagen corresponde al resultado de aplicar DIET a una frase, pudiendo extraer información relevante como la intención de esta o sus entidades.

En cuarto y último lugar, el extractor de entidades se encarga de extraer las entidades del texto que corresponden con las entidades predefinidas en el modelo: un ejemplo de esto podría ser método de pago, teléfono o nombre. Cierto tipo de entidades, como pueden ser las mencionadas anteriormente, siguen un patrón estructurado, por lo que no es necesario tener un algoritmo que se encargue de detectarlas, haciendo innecesario el uso de DIET en estos casos. Es por eso por lo que se recomienda utilizar RegexEntityExtractor para estas situaciones, el cual se encarga de extraer las entidades usando las tablas de búsqueda y *regexes* definidos en los datos de entrenamiento [36].

6.2.2 Rasa Core

Rasa Core es el componente de Rasa encargado de seleccionar qué acción se debe realizar al recibir un *intent* en función de las historias definidas y la manera en la que seleccionará el *intent* dependerá del porcentaje de coincidencia que tenga el *token* en cuestión con el *intent*.

Hace uso de una serie de ficheros: policies.yml y domain.yml. En el archivo policies.yml se deberán especificar los componentes que se quieran añadir al *pipeline* y en el archivo domain.yml se deben añadir todos los *intents*, *entities*, *slots*, *forms* y *responses*, ya que Core se encargará de seguir las historias y reglas haciendo uso de los elementos anteriormente mencionados.

7 Implementación del caso de estudio

Una vez explorada en detalle la herramienta Rasa, así como los elementos principales que posee y que permiten la creación de *chatbots* competentes, va a mostrarse el proceso completo referente al uso de esta herramienta, lo cual supone tanto la preparación del entorno como su implementación a varios niveles.

7.1 Alcance del caso de estudio

En este apartado, se delimitará el alcance del caso de estudio sobre la implementación de un *chatbot* utilizando la herramienta Rasa, estableciendo los objetivos del desarrollo del proyecto. El alcance, pues, se divide en los siguientes puntos.

7.1.1 Objetivo principal

El objetivo principal de este caso de estudio es la creación de un *chatbot* funcional y eficiente, capaz de interactuar con el usuario y que proporcione información sobre vuelos y rutas de parques nacionales.

Cada uno de estos puntos se enfoca en aspectos específicos de la funcionalidad del *chatbot*, desde la interacción con el usuario hasta la precisión en la entrega de información:

- Implementación de la capacidad para recopilar información sobre vuelos, incluyendo destino, origen y fecha del viaje.
- Desarrollo de un módulo referente a parques nacionales que permita mostrar al usuario información relevante sobre las distintas rutas que existen en estos.

Para este caso de estudio, el enfoque se ha visto limitado a las rutas pertenecientes a los parques nacionales ubicados en España.

7.1.2 Plataforma de implementación

Este proyecto ha sido desarrollado en el entorno de desarrollo Visual Studio Code, en el sistema operativo Windows 11 y haciendo uso de Python como lenguaje de programación.

Para el acceso a la información relacionada con vuelos, ha sido necesaria la integración de la API de Amadeus, la cual proporciona datos actualizados y en tiempo real de los distintos vuelos de manera confiable.

En el caso de las rutas de parques nacionales, han sido utilizados archivos recogidos del Centro de Descargas del CNIG (Centro Nacional de Información Geográfica), los cuales incluían información relevante sobre los distintos parques nacionales de la península e islas del territorio español.

7.2 Instalación de la herramienta

Para comenzar con la instalación, deben seguirse una serie de pasos que incluyen la configuración del entorno de desarrollo, la creación de un entorno virtual opcional y, por último, la instalación de Rasa y sus dependencias.

En primer lugar, Rasa precisa de Python, en concreto y a partir de la versión 3, es necesario disponer de una versión de Python comprendida entre la 3.7 y la 3.10. Para este caso de estudio, la versión escogida para Rasa ha sido la 3.5.0 y la versión de Python 3.10.0, la cual puede descargarse desde la página oficial de Python.

En segundo lugar, y de manera opcional, es recomendado la creación de un entorno virtual en el cual se ejecutará Rasa con el fin de evitar conflictos con otras bibliotecas de Python. Esto puede realizarse con herramientas como *virtualenv* o *conda*. Para instalar virtualenv basta con abrir la aplicación Símbolo del sistema (*CMD* o *Command Prompt*) y usar el siguiente comando:

```
pip install virtualenv
```

Una vez realizada la instalación de virtualenv, se puede proceder a la creación del entorno virtual posicionándose en la carpeta seleccionada para la instalación y utilizar, de nuevo, el siguiente comando en CMD:

```
python3 -m venv mi_entorno
```

En tercer lugar, es recomendable comprobar que pip esté actualizado a la última versión disponible, siendo posible esto por medio del siguiente comando:

```
pip install --upgrade pip
```

En cuarto lugar, para instalar Rasa se utilizará el siguiente comando:

```
pip install rasa
```

Y, en último lugar y con tal de comprobar que la instalación ha sido correcta, se ejecutará el siguiente comando que devolverá la versión actual de Rasa que ha sido instalada en el paso anterior:

```
rasa --version
```

7.3 Preparación

Una vez realizada la instalación de Rasa, se procederá a la creación de un directorio en el cual se creará el proyecto. Para ello, es necesario utilizar el siguiente comando:

```
rasa init
```

Este comando crea la estructura del proyecto, la cual incluye los directorios *actions*, *data* y *tests*, el fichero Python actions.py, así como los ficheros YAML nlu.yml, rules.yml, stories.yml, config.yml, credentials.yml, domain.yml y endpoints.yml. En la Ilustración 19 puede observarse la estructura mencionada.

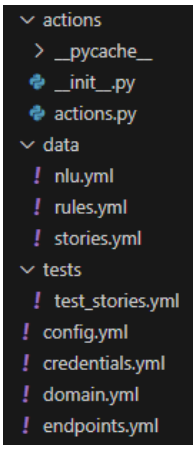

Ilustración 19. Estructura de un proyecto en Rasa [Elaboración propia]

Nota: La imagen corresponde a la estructura de un proyecto en Rasa. En esta se pueden observar los ficheros YAML que contienen diversas funciones para la operabilidad del *chatbot*, así como el fichero actions.py en el cual se programan las diversas funcionalidades de este.

Una vez creado el proyecto, se habrán generado una serie de ficheros que contienen información y que forman parte del ejemplo de bienvenida de Rasa. Esto permite al desarrollador comprender como funcionan los diversos ficheros y la responsabilidad que tiene cada uno de ellos.

Además, para la creación de este proyecto han sido utilizadas tanto una fuente de datos externa, como una API:

- Rutas Parques Nacionales de España: Desde el Centro de Descargas del CNIG [37] fueron descargados múltiples archivos en formato KML, los cuales contenían la información relativa a las distintas rutas que existen dentro de los parques nacionales de España. Fue necesario formatear la totalidad de estos archivos, mediante un programa creado en Python, para su posterior inclusión en un archivo Excel información_rutas.xlsx.

- API de Amadeus: Desde la página web de Amadeus [38] fue necesaria la creación de una cuenta gratuita y, posteriormente, la creación de una aplicación desde el *workspace* que se nos brinda. Esta aplicación ofrece unas claves necesarias para la conexión con el servicio de Amadeus y ha servido para realizar la búsqueda de los vuelos, así como toda la información referente a estos. Cabe destacar que existe un límite de 500 peticiones para cada cuenta gratuita.

7.4 Implementación

Una vez inicializado el proyecto, se inicia con la puesta en marcha de los componentes del *chatbot*. Esto implica la programación de las interacciones del *bot*

con el usuario, la configuración de las políticas de manejo de diálogos y la integración de funcionalidades que permitan alcanzar el objetivo propuesto en el alcance del proyecto.

7.4.1 config.yml

El archivo config.yml es utilizado para definir la configuración del modelo de lenguaje, así como otros aspectos importantes del funcionamiento del *chatbot*. En este se puede encontrar los componentes *pipeline* y *policies*.

El componente *pipeline* es una lista ordenada de componentes que se utilizan para procesar y comprender los mensajes del usuario, así como su intención.

El componente *policies* es una lista ordenada de reglas y algoritmos que determinan como debe tomar las decisiones el *chatbot* durante la conversación con el usuario.

```yaml
assistant_id: default_config_bot
language: "es"

pipeline:
  - name: WhitespaceTokenizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: CRFEntityExtractor
  - name: EntitySynonymMapper
  - name: DIETClassifier
    epochs: 100
  - name: ResponseSelector
    epochs: 100
policies:
  - name: "MemoizationPolicy"
    max_history: 5
  - name: "TEDPolicy"
    max_history: 5
    epochs: 100
  - name: "RulePolicy"
    core_fallback_threshold: 0.3
    core_fallback_action_name: "action_default_fallback"
```

Ilustración 20. Estructura del archivo config.yml [Elaboración propia]

Nota: La imagen corresponde a la estructura del archivo config.yml del proyecto actual.

La selección de los componentes, tanto en el *pipeline* como en *policies*, es muy importante puesto que permite que el *chatbot* responda de manera adecuada, por lo que es necesario realizar una buena selección de componentes.

En el *pipeline*, la selección de componentes que ha sido escogida (ver Ilustración 20), en base a su definición [39], es la siguiente:

- WhitespaceTokenizer: Facilita la identificación de palabras clave en la solitud del usuario ya que divide el texto en *tokens* que el modelo de lenguaje puede procesar de manera óptima.

- RegexFeaturizer: Permite la detección de patrones en el texto, como direcciones de correo electrónico o números de teléfono, lo cual facilita la tarea de seleccionar las entidades que proporciona el usuario.

- LexicalSyntacticFeaturizer: Permite analizar la estructura léxica y sintáctica de la solicitud del usuario, permitiendo identificar la intencionalidad de este.

- CountVectorsFeaturizer: Transforma el texto en vectores, contando cuántas veces aparece cada palabra en un contexto determinado, mejorando la comprensión del *chatbot*. La segunda instancia de este sirve para transformar los caracteres en vectores, pudiendo extraer información ya no solo de las palabras, si no de estos.

- CRFEntityExtractor: Es un extractor de entidades basado en CRF (*Conditional Random Fields*) que permite identificar y extraer entidades de los mensajes del usuario.

- DIETClassifier: Es un clasificador de intenciones y extractor de entidades de alto rendimiento que, junto a CRFEntityExtractor, cumplirán con el objetivo de extraer tantos los *intents* como las entidades correctamente.

- EntitySynonymMapper: Permite al *chatbot* reconocer diferentes variaciones de una misma entidad como una sola, pudiendo flexibilizar la entrada que proporciona el usuario.

- ResponseSelector: Permite predefinir respuestas a entradas concretas del usuario, siendo útil para manejar respuestas y preguntas frecuentes.

De igual manera, la selección de componentes para *policies* ha sido la siguiente:

- MemoizationPolicy: Se utiliza para memorizar las acciones que el *chatbot* ha tomado durante la conversación con el usuario antes entradas concretas, pudiendo hacer uso de esta respuesta memorizada y dotando de una mayor consistencia al *chatbot*.

- TEDPolicy: Permite comprender el contexto de la conversación, siendo útil en conversaciones complejas y extensas, pudiendo así anticiparse a la respuesta que precisa el usuario.

- RulePolicy: Permite definir reglas conversacionales para controlar las acciones que realiza el *chatbot* antes una entrada concreta

7.4.2 nlu.yml

El archivo nlu.yml es utilizado para definir ejemplos de entrenamiento para la comprensión del lenguaje natural del *chatbot*. Este contiene ejemplos de frases que los usuarios pueden enviar al *chatbot*, por lo que un buen entrenamiento, incluyendo una gran cantidad de ejemplos, supondrá que el *chatbot* tenga una mayor capacidad de comprender aquello que el usuario le está requiriendo y, por tanto, acertará de mejor manera la intención de este (ver Ilustración 21).

El formato correspondiente a este archivo es el siguiente:

- *intent*: indica el nombre del *intent*.
- *examples*: permite definir una serie de ejemplos con el fin de entrenar el *chatbot* y, así, aumentar su efectividad. La inclusión de entidades en los ejemplos permite entrenar el *chatbot* en el reconocimiento de estas.

En este archivo han sido definidos los *intents* que permiten buscar un vuelo específico en función del destino, origen y fecha, así como también el *intent* que permite extraer la diferentes rutas que poseen los parques nacionales del destino seleccionado. Además, se ha definido un *intent* referente al saludo entre el usuario y el *chatbot*.

```
- intent: buscar_vuelo_completo
  examples: |
    - Quiero encontrar un vuelo desde [Nueva York](origen) a [Los Ángeles](destino) el [15 de octubre](fecha)
    - Busca vuelos de [Miami](origen) a [Chicago](destino) para el [próximo viernes](fecha)
    - ¿Puedes ayudarme a encontrar un vuelo de [Londres](origen) a [París](destino) el [10 de septiembre](fecha)?
    - Necesito un vuelo de [Madrid](origen) a [Barcelona](destino) el día [5 de noviembre](fecha)
    - Dame opciones de vuelo desde [San Francisco](origen) a [Las Vegas](destino) el [próximo lunes](fecha)
    - ¿Cuáles son los vuelos disponibles de [Chicago](origen) a [Miami](destino) para el día [20 de diciembre](fecha)?
    - Busco un vuelo de [Roma](origen) a [Atenas](destino) para el [próximo viernes](fecha)
    - ¿Tienes información sobre vuelos de [Los Ángeles](origen) a [Nueva York](destino) el [25 de septiembre](fecha)?
    - Necesito reservar un vuelo desde [París](origen) a [Berlín](destino) para el [30 de octubre](fecha)
    - Quiero encontrar un vuelo de [Londres](origen) a [Nueva York](destino) el [miercoles próximo](fecha)
    - Busco opciones de vuelo de [San Francisco](origen) a [Miami](destino) el dia [15 de noviembre](fecha)
    - ¿Cuáles son los vuelos disponibles de [Barcelona](origen) a [Madrid](destino) para el [3 de diciembre](fecha)?
    - Dame información sobre vuelos de [Chicago](origen) a [Los Ángeles](destino) para el [8 de octubre](fecha)
    - Necesito un vuelo de [Nueva York](origen) a [Miami](destino) el [5 de noviembre](fecha)
    - Busca opciones de vuelo desde [Berlín](origen) a [París](destino) para el [proximo sábado](fecha)
```

Ilustración 21. Ejemplo de estructura del archivo nlu.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo nlu.yml, diferenciándose el *intent* en cuestión, así como los ejemplos ofrecidos para entrenar al *chatbot*, dentro de los cuales puede observarse las entidades origen, destino y fecha.

7.4.3 stories.yml

El archivo stories.yml corresponde al archivo en el que se definen historias de conversación entre el usuario y el *chatbot* con el fin de entrenar a este, consistiendo en una secuencia de eventos que muestran la interacción entre estos (ver Ilustración 22).

El formato correspondiente a este archivo es el siguiente:

- *story*: indica el nombre de la historia.
- *intent*: nombre del *intent* proporcionado por el usuario y detectado por el *chatbot*.
- *action*: nombre de la acción que se ejecutará posteriormente al *intent*.

En este archivo han sido definidas las historias referentes a la búsqueda de vuelos y rutas.

```yaml
stories:
- story: Saludo y reserva de vuelo
  steps:
  - intent: saludar
  - action: utter_saludo
  - intent: reservar_vuelo
  - action: utter_pregunta_destino
  - intent: indicar_destino
  - action: action_confirmar_destino
  - intent: indicar_origen
  - action: action_confirmar_origen
  - intent: indicar_fecha
  - action: action_confirmar_fecha
  - action: action_buscar_vuelos
```

Ilustración 22. Ejemplo de estructura del archivo stories.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo stories.yml, en la cual puede observarse una secuencia de *intents* y sus respectivos actions.

7.4.4 rules.yml

El archivo *rules*.yml permite la definición de reglas conversacionales que guían el comportamiento del *chatbot* en diferentes situaciones, permitiendo controlar cómo debe actuar el *chatbot* ante un *intent* específico (ver Ilustración 23).

El formato correspondiente a este archivo es el siguiente:

- *rule*: indica el nombre de la regla.
- *intent*: nombre del *intent* que sirve como disparador de la regla.
- *action*: nombre de la acción que ejecutará el *chatbot* al detectar el *intent* anterior.
- *slot_was_set*: elemento opcional que permite verificar si un *slot* específico ha sido establecido.
- *active_loop*: elemento opcional que permite verificar si un *loop* activo está en curso.

```yaml
- rule: Buscar ruta
  steps:
  - intent: buscar_rutas
  - action: action_confirmar_destino_ruta
```

Ilustración 23. Ejemplo de estructura del archivo rules.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo *rules*.yml. En esta puede observarse tanto el *intent* y como el *action* correspondientes a la regla “Buscar ruta”.

En este archivo han sido definidas las distintas reglas que permiten el correcto funcionamiento del *chatbot*. Estas reglas hacen referencia a la secuencia de *intents* y actions que permiten extraer los diferentes *entities* desde la respuesta del usuario.

7.4.5 domain.yml

El archivo domain.yml permite definir y organizar todos los componentes que han sido definidos en los distintos archivos YAML. Es necesario la inclusión de todos estos elementos en dicho archivo para la correcta funcionalidad del *chatbot* (ver Ilustración 24).

Los diferentes elementos que pueden encontrarse en este archivo son los siguientes:

- *intents*: define el nombre de los distintos *intents*.
- *entities*: define el nombre de las entidades que permiten la extracción de valores a partir de una respuesta del usuario.
- *responses*: define las respuestas predefinidas del *chatbot*.
- *actions*: define las acciones que realiza el *chatbot* antes una intención determinada.
- *slots*: define el nombre de las variables *slot* que almacenan un valor específico.

```yaml
version: "3.1"

intents:
- saludar
- buscar_vuelo
- buscar_vuelo_completo
- indicar_destino
- indicar_origen
- indicar_fecha
- buscar_rutas

entities:
  - destino
  - origen
  - fecha

slots:
  destino:
    type: text
    mappings:
      - type: from_entity
        entity: destino
  origen:
    type: text
    mappings:
      - type: from_entity
        entity: origen
  fecha:
    type: text
    mappings:
      - type: from_entity
        entity: fecha

responses:
  utter_saludo:
    - text: ¡Hola! ¿En qué puedo ayudarte hoy?
  utter_pregunta_destino:
    - text: Por supuesto, ¿a dónde te gustaría viajar?

actions:
- action_buscar_vuelos
- action_buscar_vuelo_completo
- action_confirmar_destino
- action_confirmar_origen
- action_confirmar_fecha
- action_confirmar_destino_ruta
- action_buscar_en_amadeus

session_config:
  session_expiration_time: 60
  carry_over_slots_to_new_session: true
```

Ilustración 24. Ejemplo de estructura del archivo domain.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo domain.yml. Pueden observarse las distintas definiciones para cada uno de los elementos principales del *chatbot*.

En este archivo han sido incluidos todos los componentes creados en los nlu.yml, stories.yml y rules.yml, así como la definición de los *slots*.

7.4.6 endpoints.yml

El archivo endpoints.yml es un archivo de configuración utilizado para especificar como se comunica el *chatbot* con servicios externos. En este archivo es únicamente necesaria la inclusión del código de la Ilustración 25 para que escuche en un puerto específico.

```
action_endpoint:
  url: "http://localhost:5055/webhook"
```

Ilustración 25. Ejemplo de estructura del archivo endpoints.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo endpoints.yml, en el cual se define la URL en la cual escucha el programa.

En este archivo ha sido definido el *endpoint*, véase la imagen anterior, necesario para realizar la conexión con el servicio de Amadeus.

7.4.7 test_stories.yml

El archivo test_stories.yml es un archivo que se utiliza para la definición de historias de prueba y cuyo propósito es el de validar el rendimiento del *chatbot*, así como la detección de problemas varios (ver Ilustración 26).

La estructura que debe seguir es la siguiente:

- *story*: nombre de la historia.
- *user*: frase de ejemplo que utiliza el usuario para transmitir la intención deseada. Puede incluir entidades que seguirán el formato [“valor”]{“*entity*”: “nombre_*entity*”}.
- *intent*: nombre del *intent* que debe recibir el *chatbot*.
- *action*: nombre de la acción que va a ejecutarse.

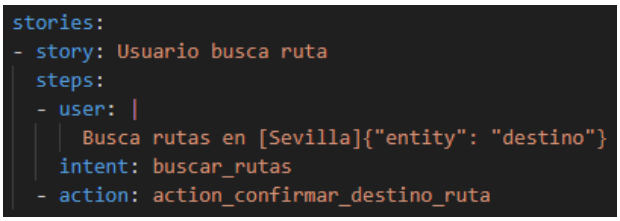

Ilustración 26. Ejemplo de estructura del archivo test_stories.yml [Elaboración propia]

Nota: La imagen corresponde a un ejemplo de la estructura del archivo test_stories.yml, en el cual se define una historia de ejemplo para comprobar la validez de la lógica del *chatbot*.

Para comprobar la validez de este archivo test_stories, es necesario la utilización del siguiente comando:

rasa test

7.4.8 credentials.yml

El archivo credentials.yml sirve como el almacén de credenciales y claves de acceso necesarias para realizar la conexión con servicios externos.

En el caso de este proyecto, las credenciales necesarias para la conexión del servicio Amadeus no han sido incluidas en este fichero, puesto que este fichero es exclusivo de la versión de pago Rasa X.

7.4.9 actions.py y acciones personalizadas

El archivo actions.py es un componente de gran importancia dentro del entorno de Rasa. Contiene las definiciones de las distintas acciones personalizadas, programadas por el usuario, que permiten al *chatbot* responder al usuario de manera altamente personalizada. La utilización de acciones personalizadas no se restringe únicamente al archivo actions.py, sino que es posible crear tantos archivos como sean necesarios.

En el caso de este proyecto, han sido creados también los archivos utilidades_amadeus.py, utilidades_excel.py, utilidades_fecha.py y amadeus_config.py. Este último archivo ha sido utilizado como almacén de las claves de la API de Amadeus. Sobre el resto de los archivos, cada uno de ellos ha permitido organizar las distintas acciones por bloques, estando divididas en:

- Acciones propias de Amadeus tales como la extracción de vuelos.
- Acciones referentes a la validación de fechas, como la conversión del formato de fecha.
- Acciones referentes a la extracción de los distintos campos del fichero información_rutas.xlsx.

Una vez implementadas las distintas acciones personalizas, así como el resto de los archivos, es necesario el lanzamiento de la siguiente instrucción, en un Símbolo del sistema aparte, con tal de poder hacer uso de estas acciones:

```
rasa run actions
```

7.5 Resultados

Para comprobar la confianza del sistema, tanto de *intents* como de *entities*, se ha hecho uso del comando:

```
rasa test nlu --nlu NLU
```

Este comando se encarga de la evaluación del rendimiento del modelo de procesamiento de lenguaje natural en la clasificación de *intents* y la extracción de *entities*, utilizando los conjuntos de prueba predefinidos en el archivo a especificar NLU. En este caso, el archivo NLU poseía una cantidad superior de ejemplos por cada intención, lo cual ha permitido evaluar con una mayor veracidad el modelo programado y extraer información valiosa.

Ilustración 27. Resultados de las matrices de confusión de entities de CRFEntityExtractor [Elaboración propia]

Nota: La imagen corresponde a las matrices de confusión de entidades del componente CRFEntityExtractor, usando tanto el archivo nlu.yml utilizado para entrenar el modelo, como el archivo NLU.

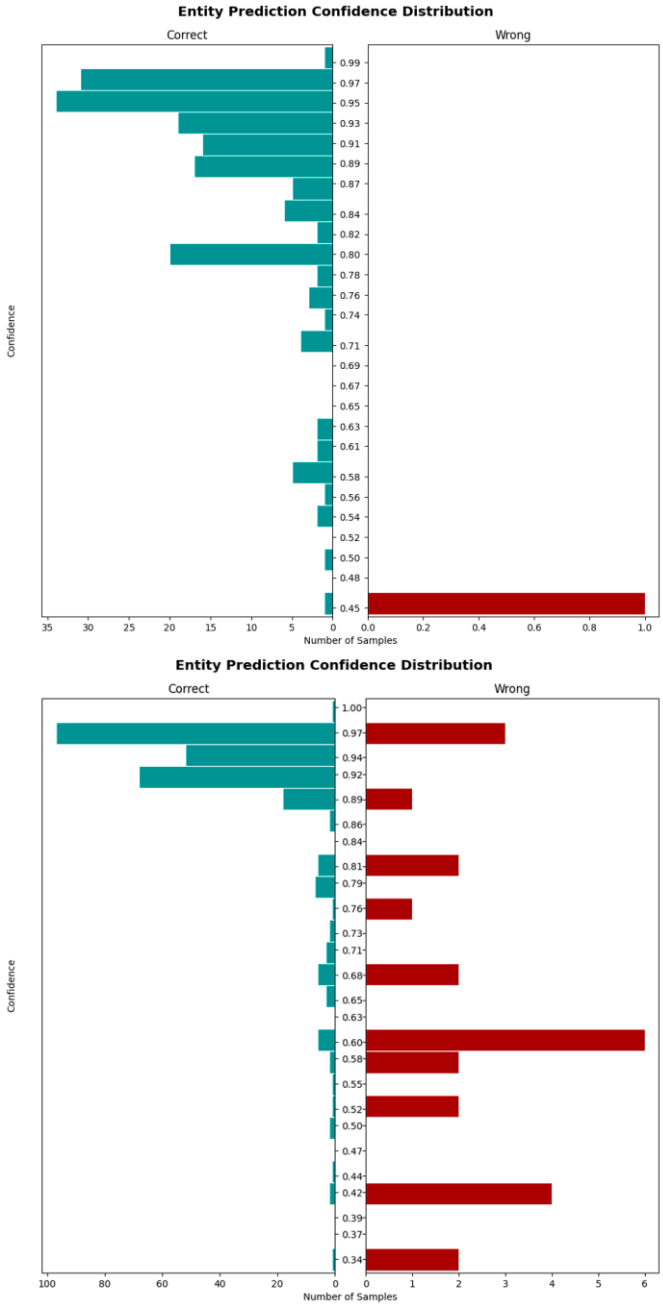

Ilustración 28. Histogramas de predicción de entities de CRFEntityExtractor [Elaboración propia]

Nota: La imagen corresponde a los histogramas de predicción de entidades del componente CRFEntityExtractor, usando tanto el archivo nlu.yml utilizado para entrenar el modelo, como el archivo NLU.

Como puede observarse en las ilustraciones 27 y 28, la clasificación de *intents* y la extracción de *entities* por parte del componente CRFEntityExtractor se realiza con normalidad, si bien es cierto que el uso de un archivo con un mayor número de ejemplos más diversos ha supuesto una mayor cantidad de errores, lo cual puede indicar que el modelo necesita entrenarse con una mayor cantidad y variedad de ejemplos como ocurre en el archivo NLU especificado.

Ilustración 29. Resultados de las matrices de confusión de entities de DIETClassifier [Elaboración propia]

Nota: La imagen corresponde a las matrices de confusión de *entities* del componente DIETClassifier, usando tanto el archivo nlu.yml utilizado para entrenar el modelo, como el archivo NLU.

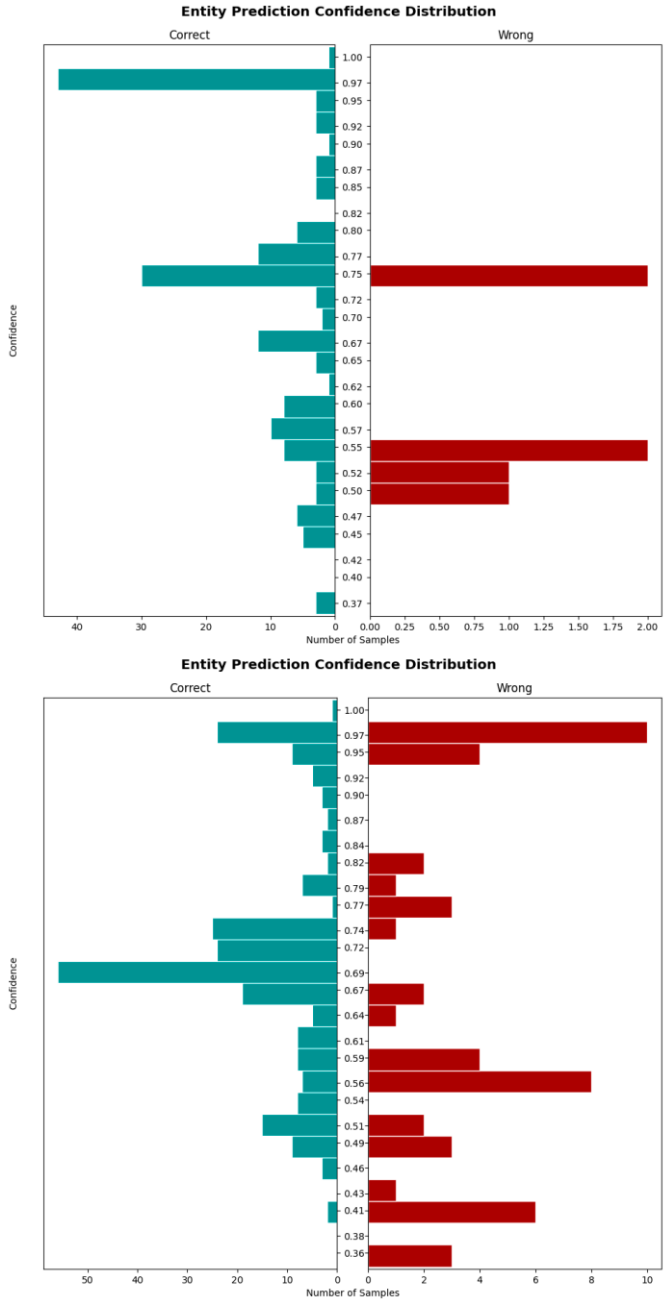

Ilustración 30. Histogramas de predicción de entities de DIETClassifier [Elaboración propia]

Nota: La imagen corresponde al histograma de predicción de *entities* del componente DIETClassifier, usando tanto el archivo "nlu.yml" utilizado para entrenar el modelo, como el archivo NLU.

De igual manera, puede observarse en estas dos últimas ilustraciones (29 y 30) lo mencionado anteriormente con el componente CRFEntityExtractor, con una peculiaridad, y es que se puede observar como el componente DIETClassifier tiene un rendimiento inferior al componente CRFEntityExtractor en cuanto a la extracción de *entities*.

Es por ello por lo que se ha decidido utilizar únicamente el componente CRFEntityExtractor como extractor de *entities*, relegando a DIETClassifier a la extracción de *intents*.

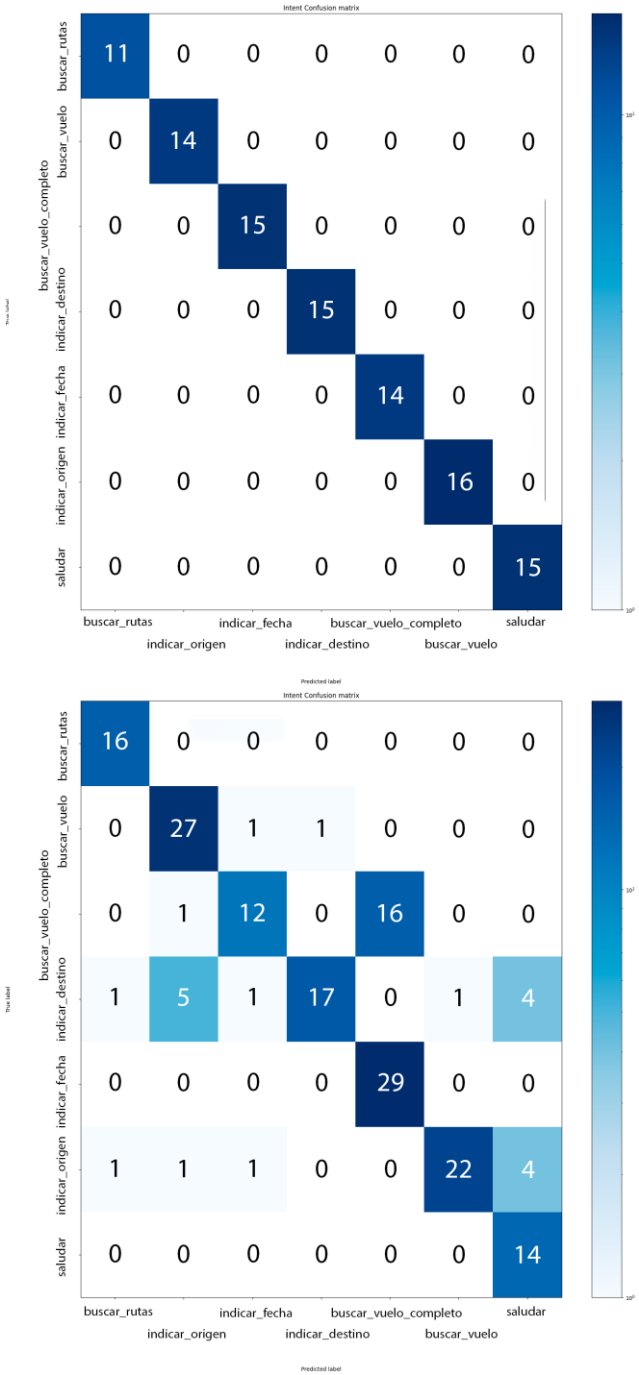

Ilustración 31. Matrices de confusión de intents de DIETClassifier [Elaboración propia]

Nota: La imagen corresponde a las matrices de confusión de *intents* del componente DIETClassifier, usando tanto el archivo nlu.yml utilizado para entrenar el modelo, como el archivo NLU.

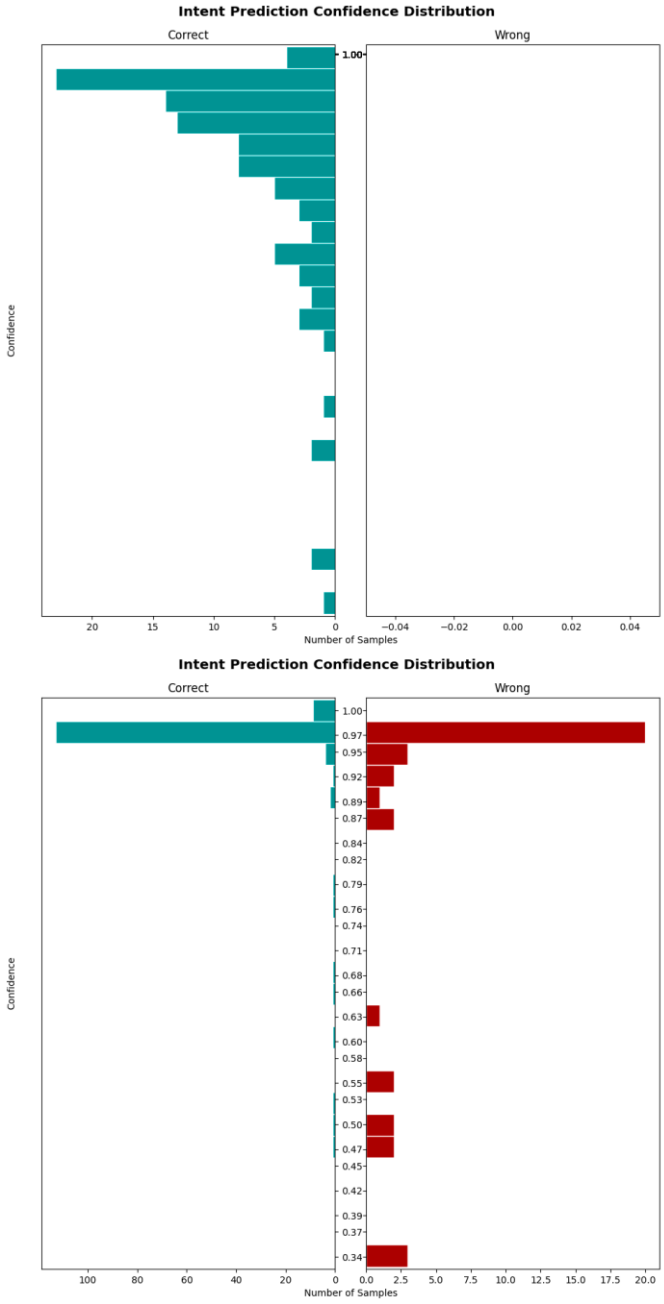

Ilustración 32. Histogramas de predicción de intents de DIETClassifier [Elaboración propia]

Nota: La imagen corresponde al histograma de predicción de *intents* del componente DIETClassifier, usando tanto el archivo nlu.yml utilizado para entrenar el modelo, como el archivo NLU.

Con respecto a la extracción de *intents*, puede observarse en las ilustraciones 31 y 32 que su clasificación resulta más exitosa que la clasificación de *entities*, percibiéndose un claro aumento de errores en su clasificación una vez utilizado el archivo NLU, en su mayoría focalizados en los *intents* buscar_vuelo_completo e indicar_destino. El hecho de que los errores se encuentren, mayormente, en estos *intents*, es debido a que la estructura que siguió el entrenamiento del modelo contiene la misma entidad, destino, así como los ejemplos ofrecidos para su entrenamiento suelen poseer la preposición *a* en ambas intenciones.

Una vez analizados los resultados de la extracción de *entities* e *intents*, es momento de observar los resultados obtenidos a nivel de programación.

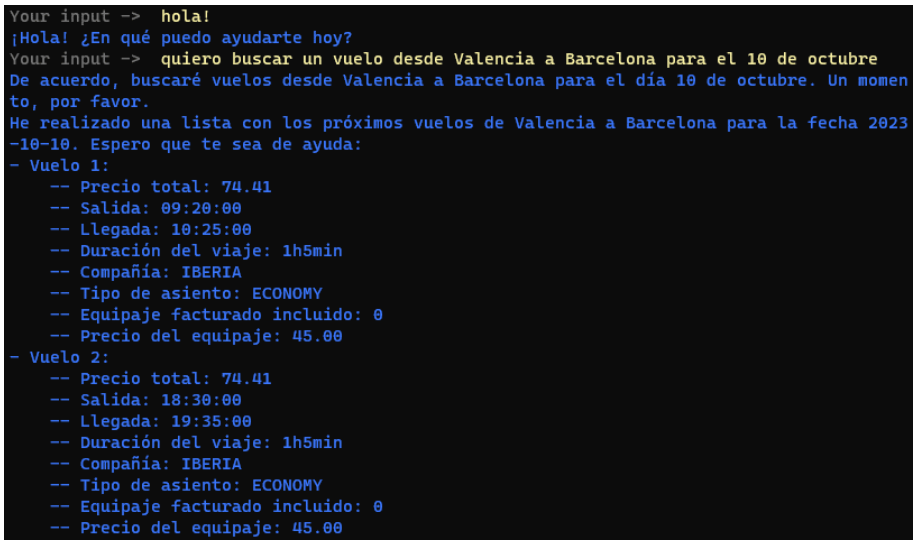

Ilustración 33. Búsqueda completa de vuelos [Elaboración propia]

Nota: En la imagen puede observarse el resultado del *intent* buscar_vuelo_completo, en el cual puede observarse como el *chatbot* extrae las diferentes *entities* que están presentes en la respuesta del usuario y, en base a su valor, realiza una búsqueda mediante la API de Amadeus, mostrando finalmente una lista de vuelos en la cual se encuentra el número de vuelo, precio total, hora de salida y llegada, duración del viaje, compañía, tipo de asiento, equipaje facturado incluido y precio del equipaje.

En la Ilustración 33 puede observarse la interacción con el usuario y el *chatbot* haciendo uso de los *intents* saludar y buscar_vuelo_completo. Tras proporcionar la entrada correspondiente a buscar_vuelo_completo, el *chatbot* procesa dicha entrada y extrae la totalidad de las *entities* presentes, siendo estas origen, destino y fecha. Tras esto, llama a la acción *action*_buscar_vuelo_completo, la cual se ubica en el archivo actions.py y ejecuta el código referente a la extracción de vuelos por medio de la API de Amadeus. Por último, y como puede observarse, los resultados obtenidos son transmitidos al usuario de manera ordenada, tras ser procesados, indicando una lista con los vuelos obtenidos para dicha fecha.

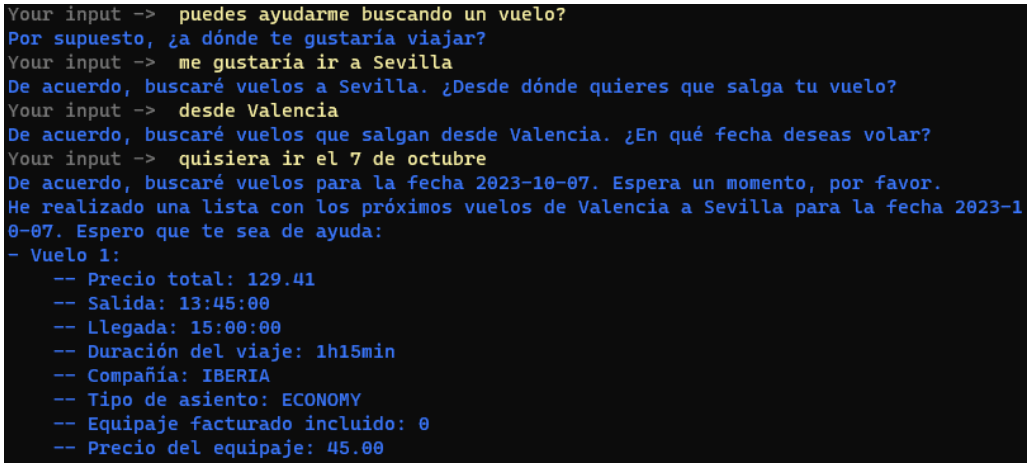

Ilustración 34. Búsqueda de vuelos [Elaboración propia]

Nota: En la imagen puede observarse el resultado del *intent* buscar_vuelo, en el cual puede observarse la conversación completa en la cual el *chatbot* extrae los valores de destino, origen y fecha mensaje por mensaje. Tras ello, muestra una lista de los vuelos disponibles de igual manera que en la anterior imagen.

Pasando a la siguiente funcionalidad, se puede observar en la ilustración 34 el diálogo referente a los *intents* buscar_vuelo, indicar_destino, indicar_origen e indicar_fecha.

Una vez el *chatbot* detecta la intención buscar_vuelo, precisa del usuario las *entities* destino, origen y fecha, siendo ofrecidas por este en cada una de las acciones a las que refieren. Tras esto, y de igual manera que en la intención buscar_vuelo_completo, se llama a la acción correspondiente y se ejecuta el código que permite la extracción de vuelos.

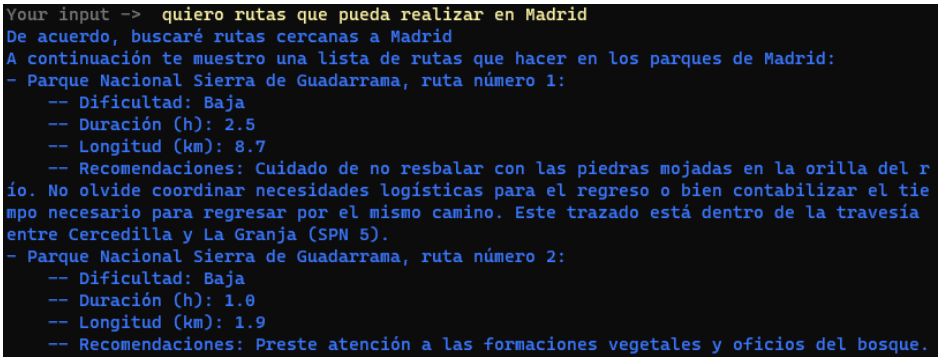

Ilustración 35. Búsqueda de rutas [Elaboración propia]

Nota: En la imagen puede observarse el resultado del *intent* buscar_rutas, en el cual pueden observarse múltiples elementos como son el número de ruta, dificultad, duración, longitud y recomendaciones.

Por último, el diálogo representado en la ilustración 35 muestra la intención buscar_rutas, la cual precisa de la entidad destino.

Tras recibir la intención en cuestión, incluyendo el nombre de la ciudad donde se quieren buscar las rutas, se llama a la acción *action*_confirmar_destino_ruta, la cual realiza una búsqueda en el archivo Excel información_rutas.xlsx, compuesto por múltiples columnas de información, y extrae la información de estas columnas en base a la ciudad destino escogida. Una vez escogidas las filas que refieren a dicha ciudad, se extrae el nombre del parque nacional, al cual refieren, y se muestra la información de las rutas que este posee, devolviendo, una vez más, una lista ordenada de resultados.

7.6 Complicaciones y desafíos técnicos

Durante el desarrollo de este proyecto, me he enfrentado a varios desafíos técnicos que requirieron una gran dedicación para solventarlos, teniendo algunos de ellos una fácil solución, así como otros una más compleja. A continuación, se detallan algunos de los desafíos y como han sido abordados.

7.6.1 *Pipeline* de Rasa

Como se ha mencionado anteriormente, el *pipeline* es un elemento esencial dentro del entorno de Rasa, pues permite dotar al *chatbot* de ciertas funcionalidades que le permiten entender el lenguaje de una manera más profunda.

Uno de los problemas con los que se ha tenido que lidiar es con errores a la hora de importar elementos al *pipeline*. Para este proyecto se planteó el uso de *spaCy*, una librería de procesamiento de lenguaje natural popular y de gran potencia que ofrece también modelos previamente entrenados. Sin embargo, tuvo que obviarse su uso por problemas de compatibilidad entre Python y Rasa.

Otro problema que se encontró a la hora de realizar pruebas fue que el extractor de *entities* DIETClassifier no estaba trabajando correctamente, ya que había algunas *entities* que no clasificaba correctamente, por lo que hubo que complementarlo con CRFEntityExtractor para conseguir una eficacia mayor. En el *pipeline* se puede observar cómo CRFEntityExtractor se encuentra por encima de DIETClassifier, lo cual supone que las *entities* serán extraídas únicamente por el componente CRFEntityExtractor. La clasificación errónea de *entities* por parte de DIETClassifier puede observarse mediante los archivos DIETClassifier_histogram.png y DIETClassifier_errors.json (ver ilustraciones 36 y 37).

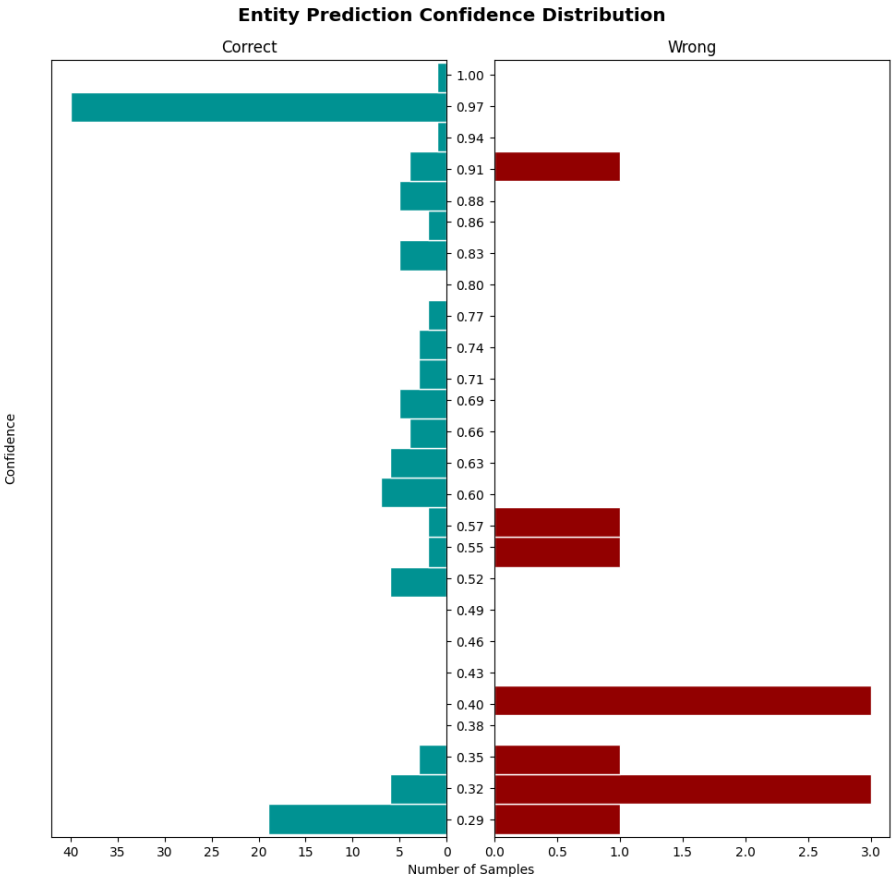

Ilustración 36. Histograma de predicción de entities [Elaboración propia]

Nota: La imagen corresponde al histograma de predicción de *entities* del componente DIETClassifier. Como puede observarse, en lado derecho se muestra la cantidad de *entities* predichas erróneamente.

```
"text": "Quiero reservar para el 17 de junio",
"entities": [
  {
    "start": 24,
    "end": 35,
    "value": "17 de junio",
    "entity": "fecha"
  }
],
"predicted_entities": [
  {
    "entity": "fecha",
    "start": 21,
    "end": 23,
    "confidence_entity": 0.5984535813331604,
    "value": "el",
    "extractor": "DIETClassifier"
  },
  {
    "entity": "fecha",
    "start": 24,
    "end": 35,
    "confidence_entity": 0.3280777037143707,
    "value": "17 de junio",
    "extractor": "DIETClassifier"
  }
]
```


Ilustración 37. Archivo DIETClassifier_errors.json [Elaboración propia]

Nota: En la imagen puede observarse el archivo DIETClassifier_errors.json, en el cual se puede observar cómo DIETClassifier cataloga de forma incorrecta la entidad fecha “17 de junio”.

7.6.2 Integración de la API de Amadeus

Uno de los mayores desafíos fue la integración de la API de Amadeus para recibir información sobre vuelos, aun siendo que la documentación de esta se encuentra completa y dispone de una gran cantidad de métodos con múltiples funcionalidades.

Una vez recuperada la lista de vuelos dado un origen, un destino y una fecha, el archivo JSON que devolvía contenía una gran cantidad de información, tanto relevante para el caso de estudio como no. Fue necesario realizar un análisis detallado de la estructura del JSON e implementar un sistema de filtrado y procesamiento de datos para garantizar que solo se obtuvieran los datos requeridos. Además, ciertos elementos no se encontraban anidados de manera lógica, lo cual puede llegar a dificultar el procesamiento de estos datos.

Por último, un problema recurrente que ha surgido en varias ocasiones es que el servicio de la API de Amadeus se encontraba fuera de línea, lo que ocasionaba una incapacidad a la hora de buscar vuelos.

8 Conclusiones

Mediante la realización de este trabajo, ha sido posible extraer una serie de conclusiones, tanto en la fase de evaluación de las distintas herramientas disponibles para la creación de *chatbots*, como en la fase de implementación y experimentación con la herramienta escogida, en este caso, Rasa. También ha podido extraerse conclusiones acerca de la metodología presentada, las cuales se presentarán tras analizar las conclusiones de las fases mencionadas anteriormente.

Sobre la primera fase, la elección de la herramienta Rasa por delante del resto de herramientas fue una decisión tomada en base a la complejidad del caso de estudio que se planteaba, siendo esta una complejidad media. Se consideró, en base a las subcaracterísticas extraídas, que las herramientas Amazon Lex y Dialogflow, si bien podrían lidiar con la implementación de ciertas características que ha sido necesario investigar, como bien puede ser el *pipeline* y todos sus complementos, no ofrecían la misma flexibilidad y personalización que ofrece Rasa. Por otra parte, para desarrolladores principiantes y usuarios sin experiencia previa, el hecho de que ambas herramientas dispongan de una interfaz gráfica puede resultar en una mayor facilidad de uso, permitiéndoles la creación de *chatbots* de una manera más dinámica y fluida. Cabe destacar que Rasa también posee una herramienta con interfaz gráfica, como se ha mencionado anteriormente, pero su uso no está permitido salvo pago, pudiendo tener una facilidad de uso similar a estas dos herramientas mencionadas anteriormente.

Por último, y a cerca de la segunda fase, ha podido observarse en las diferentes matrices e histogramas, es necesaria la utilización de una gran cantidad de ejemplos para conseguir entrenar el modelo de manera correcta. Este problema puede solventarse mediante el uso de modelos previamente entrenados, pero puede suponer una capa adicional de complejidad, puesto que Rasa es compatible con una breve lista de modelos de lenguaje. Cabe destacar, además, que es posible la implementación de una interfaz gráfica de cara al usuario que va a utilizar el *chatbot* haciendo uso de HTML, CSS o JavaScript, siendo esta una manera más sencilla de acercar al usuario el uso del *chatbot*.

Acerca de la metodología que se ha utilizado para este caso de estudio, ha podido concluirse que hace falta una validación más exhaustiva de esta mediante su inclusión en múltiples casos de estudio con tal de comprobar la fiabilidad que ofrecen tanto las etapas como los *rollback* que han sido definidos. Además de su inclusión en casos de estudio diversos, también sería interesante comprobar su efectividad antes frameworks de otra índole, como los mencionados en este proyecto y que fueron desechados en favor de Rasa.

Para finalizar, cabe añadir que las capacidades de este *chatbot* pueden ser ampliadas mediante nuevas y más complejas funcionalidades, como bien puede ser la inclusión de nuevas API de extracción de datos referentes a múltiples compañías y servicios tales como trenes o barcos.

9 Bibliografía

[1] S. Hernández Cotón y J. Sánchez Gutiérrez, «Las consecuencias de la tercera Revolución Industrial», *Mercados y Negocios*, n.º 8, pp. 11-20, feb. 2003, doi: 10.32870/myn.v0i8.4954.

[2] Amazon Web Services, «¿Qué es un bot? - Explicación sobre los tipos de bots - AWS», 2023. https://aws.amazon.com/es/what-is/bot/ (accedido 21 de junio de 2023).

[3] Nahdatul Akma Ahmad, Mohamad Hafiz Che Hamid, Azaliza Zainal, Muhammad Fairuz Abd Rauf, y Zuraidy Adnan, «Review of chatbots design techniques», *Int J Comput Appl*, vol. 181, n.º 8, pp. 7-10, ago. 2018.

[4] I. El Naqa y M. J. Murphy, «What Is Machine Learning?», en *Machine Learning in Radiation Oncology*, Cham: Springer International Publishing, 2015, pp. 3-11. doi: 10.1007/978-3-319-18305-3_1.

[5] A. Følstad, M. Skjuve, y P. B. Brandtzaeg, «Different Chatbots for Different Purposes: Towards a Typology of Chatbots to Understand Interaction Design», 2019, pp. 145-156. doi: 10.1007/978-3-030-17705-8_13.

[6] A. Følstad, M. Skjuve, y P. B. Brandtzaeg, «Different Chatbots for Different Purposes: Towards a Typology of Chatbots to Understand Interaction Design», 2019, pp. 145-156. doi: 10.1007/978-3-030-17705-8_13.

[7] «Chatbot | Deep learning | Amazon Lex», *Amazon Web Services, Inc.*, 2023.

[8] «¿Qué es AWS Lambda?», *AWS Lambda*. https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html (accedido 6 de septiembre de 2023).

[9] GitHub CI, «Rasa Architecture Overview», *Rasa Technologies GmbH*, 2023.

[10] Dialogflow, «Dialogflow libraries and samples», *GitHub*, 2018. https://github.com/dialogflow/resources (accedido 28 de agosto de 2023).

[11] «Documentación de Dialogflow», *Google Cloud*. https://cloud.google.com/dialogflow/docs?hl=es-419 (accedido 6 de septiembre de 2023).

[12] gunthercox, «About ChatterBot — ChatterBot 1.0.8 documentation», *Chatterbot*, 2021. https://chatterbot.readthedocs.io/en/stable/ (accedido 28 de agosto de 2023).

[13] ISO, «ISO/IEC 25010», *ISO*, 2022.

[14] «Uso de una tarjeta de respuesta - Amazon Lex V1», *Amazon Web Services, Inc.* https://docs.aws.amazon.com/es_es/lex/latest/dg/ex-resp-card.html (accedido 24 de agosto de 2023).

[15] «Receiving events using AWS Lambda function URLs - Amazon EventBridge», *Amazon Web Services, Inc.* https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-saas-furls.html (accedido 24 de agosto de 2023).

[16] «Resiliencia en Amazon Lex - Amazon Lex V1», *Amazon Web Services, Inc.* https://docs.aws.amazon.com/es_es/lex/latest/dg/disaster-recovery-resiliency.html (accedido 24 de agosto de 2023).

[17] «Preguntas frecuentes sobre Amazon LEX - Amazon Web Services», *Amazon Web Services, Inc.* https://aws.amazon.com/es/lex/faqs/ (accedido 24 de agosto de 2023).

[18] «Monitoreo de infraestructuras y aplicaciones – Amazon CloudWatch – Amazon Web Services», *Amazon Web Services, Inc.* https://aws.amazon.com/es/cloudwatch/ (accedido 24 de agosto de 2023).

[19] «Registros de API - Servicio de registro estandarizado de Seguridad - AWS CloudTrail - Amazon Web Services», *Amazon Web Services, Inc.* https://aws.amazon.com/es/cloudtrail/ (accedido 25 de agosto de 2023).

[20] «Características | Amazon Cognito | Amazon Web Services (AWS)», *Amazon Web Services, Inc.* https://aws.amazon.com/es/cognito/details/ (accedido 25 de agosto de 2023).

[21] GitHub CI, «Tracker», *Rasa Technologies GmbH*, 2022. https://rasa.com/docs/action-server/sdk-tracker/ (accedido 19 de agosto de 2023).

[22] GitHub CI, «Rasa Telemetry», *Rasa Technologies GmbH*, 18 de agosto de 2023. https://rasa.com/docs/rasa/telemetry/telemetry/ (accedido 19 de agosto de 2023).

[23] «Mensajes de respuesta enriquecida», *Google Cloud*, 2023. https://cloud.google.com/dialogflow/es/docs/intents-rich-messages?hl=es-419 (accedido 26 de agosto de 2023).

[24] «Fulfillment», *Google Cloud*, 2022. https://cloud.google.com/dialogflow/es/docs/fulfillment-overview?hl=es-419 (accedido 26 de agosto de 2023).

[25] «Dialogflow Service Level Agreement SLA | Google Cloud», *Google Cloud*, 2020. https://cloud.google.com/dialogflow/sla (accedido 26 de agosto de 2023).

[26] «Usar el ajuste de escala automático para aplicaciones altamente escalables», *Google Cloud*, 2023, Accedido: 26 de agosto de 2023. [En línea]. Disponible en: https://cloud.google.com/compute/docs/tutorials/high-scalability-autoscaling?hl=es-419

[27] «Acerca de la escalabilidad de GKE», *Google Cloud*, 2023. https://cloud.google.com/kubernetes-engine/docs/best-practices/scalability?hl=es-419 (accedido 26 de agosto de 2023).

[28] «Cloud Monitoring | Google Cloud», *Google Cloud*. https://cloud.google.com/monitoring?hl=es-419 (accedido 26 de agosto de 2023).

[29] «Seguridad, privacidad y cumplimiento en la nube | Google Cloud | Google Cloud», *Google Cloud*. https://cloud.google.com/security?hl=es-419 (accedido 26 de agosto de 2023).

[30] «Cloud Logging | Google Cloud | Cloud Logging», *Google Cloud*. https://cloud.google.com/logging?hl=es (accedido 26 de agosto de 2023).

[31] gunthercox, «Django Integration — ChatterBot 1.0.8 Documentation», *GitHub*, 2021. https://chatterbot.readthedocs.io/en/stable/django/index.html (accedido 28 de agosto de 2023).

[32] «Custom action», *Rasa & Rasa Pro Documentation*, 2023. https://rasa.com/docs/rasa/custom-actions/ (accedido 28 de agosto de 2023).

[33] «Generating NLU Data», *Rasa & Rasa Pro Documentation*. https://rasa.com/docs/rasa/generating-nlu-data/ (accedido 6 de septiembre de 2023).

[34] «Intents & Entities: Understanding the RASA NLU Pipeline», *Rasa*, 2 de diciembre de 2021. https://rasa.com/blog/intents-entities-understanding-the-rasa-nlu-pipeline/ (accedido 8 de septiembre de 2023).

[35] «Introducing DIET: state-of-the-art architecture that outperforms fine-tuning BERT and is 6X faster to train», *Rasa*. https://rasa.com/blog/introducing-dual-intent-and-entity-transformer-diet-state-of-the-art-performance-on-a-lightweight-architecture/ (accedido 6 de septiembre de 2023).

[36] «Introducing DIET: state-of-the-art architecture that outperforms fine-tuning BERT and is 6X faster to train», *Rasa*, 11 de enero de 2023. https://rasa.com/blog/introducing-dual-intent-and-entity-transformer-diet-state-of-the-art-performance-on-a-lightweight-architecture/ (accedido 8 de septiembre de 2023).

[37] «Rutas Parques Nacionales de España - Dataset», *datos.gob.es*. https://datos.gob.es/en/catalogo/e00125901-spaignrutasppnn (accedido 5 de septiembre de 2023).

[38] «Connect to Amadeus Travel APIs | Amadeus for Developers», *Amadeus IT Group SA*. https://developers.amadeus.com (accedido 5 de septiembre de 2023).

[39] «Components», *Rasa & Rasa Pro Documentation*, 2023. https://rasa.com/docs/rasa/components/ (accedido 7 de septiembre de 2023).