



UNIVERSIDAD DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Servicios cognitivos para el diseño de agentes
conversacionales: Un enfoque teórico-práctico

Cognitive services for the design of conversational agents: A
theoretical-practical approach

Realizado por
Manuel López Reviriego

Tutorizado por
Rafaela Benítez Rochel

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2021

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

**Servicios cognitivos para el diseño de agentes
conversacionales: Un enfoque teórico-práctico**

**Cognitive services for the design of conversational agents:
A theoretical-practical approach**

Realizado por
Manuel López Reviriego
Tutorizado por
Rafaela Benítez Rochel
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE 2021

Fecha defensa:
OCTUBRE 2021

Fdo.: El Secretario del Tribunal

Abstract

The development of artificial agents capable of emulating the cognitive capacities of human beings has always been one of the most ambitious goals of Computer Science researchers, more specifically, of the Artificial Intelligence subfield. The acquisition, understanding, and generation of the syntactic and semantic structures of a natural language are considered some of the cognitive capacities that most identify us as human beings.

The subfield of natural language processing has had a great boom in recent years, with the introduction into the market of conversational assistants or chatbots, which are designed and implemented following different architectures, models, techniques, etc. The main goal of this project is the contribution of a theory of conversational agents that allows describing, using the same theoretical framework, the composition, and interaction of cognitive processes in agents that can hold conversations in natural language. To empirically test the completeness and correctness of the theory, we have developed from it a *framework* in Python for the development of *chatbots*.

Following a theoretical-practical approach, we will introduce the services and cognitive processes of natural language understanding, dialogue management, and natural language generation. By way of conclusion, we will present a practical case of a conversational agent for COVID-19 that has been designed and implemented based on the theoretical concepts introduced and their corresponding implementation in our *framework*.

Keywords: Conversational assistant, chatbot, NLP, NLU, NLG

Resumen

El desarrollo de agentes artificiales capaces de emular las capacidades cognitivas de los seres humanos ha sido siempre uno de los objetivos más ambiciosos de los investigadores de las Ciencias de la Computación, más concretamente, del subcampo de la Inteligencia Artificial. La adquisición, comprensión y generación de las estructuras sintácticas y semánticas de un lenguaje natural son consideradas algunas de las capacidades cognitivas que más nos identifican como seres humanos.

El subcampo del procesamiento del lenguaje natural ha tenido un gran auge en los últimos años, con la introducción en el mercado de asistentes conversacionales o chatbots, que son diseñados e implementados siguiendo distintas arquitecturas, modelos, técnicas, etc. Este trabajo de fin de grado tiene por objeto la aportación de una teoría de agentes conversacionales que permita describir, utilizando un mismo marco teórico, la composición e interacción de procesos cognitivos en agentes que pueden mantener conversaciones en lenguaje natural. En aras de probar empíricamente la completitud y corrección de la teoría, hemos desarrollado a partir de ella un *framework* en Python para el desarrollo de *chatbots*.

Siguiendo un enfoque teórico-práctico introduciremos los servicios y procesos cognitivos de comprensión de lenguaje natural, gestión de diálogo y generación de lenguaje natural. A modo de conclusión, expondremos un caso práctico de agente conversacional para el COVID-19 que ha sido diseñado e implementado a partir de los conceptos teóricos introducidos y su correspondiente implementación en nuestro *framework*.

Palabras clave: Asistente conversacional, chatbot, NLP, NLU, NLG

Índice

1. Introducción	11
1.1. Motivación	11
1.2. Objetivos	12
1.3. Estructura del documento	12
1.4. Metodología de trabajo	13
1.5. Tecnologías utilizadas	15
2. Estado del Arte	17
2.1. Evolución histórica de los agentes conversacionales	17
2.1.1. Test de Turing (1950)	17
2.1.2. STUDENT (1964)	17
2.1.3. ELIZA (1964-1966)	18
2.1.4. SHRDLU (1968-1970)	19
2.1.5. PARRY (1971-1972)	20
2.1.6. Era de los Asistentes de voz (2009-Actualidad)	20
2.1.7. Amazon Alexa (2014-Actualidad)	21
2.2. Motor de comprensión de lenguaje natural: Wit.ai	22
2.3. Bot Framework Composer	23
2.3.1. Servicio de comprensión de lenguaje natural LUIS	24
2.3.2. Arquitectura	25
2.3.3. Diálogos	25
2.3.4. Puntos positivos y negativos de Bot Framework Composer	26
2.4. Framework Rasa para la creación de <i>chatbots</i>	28
2.4.1. Dominio del agente	28
2.4.2. Pipeline NLU	30
2.4.3. Políticas	31
2.4.4. Forms	32
2.4.5. Formato de datos de entrenamiento	32

2.4.6.	Datos de entrenamiento NLU	32
2.4.7.	Datos de entrenamiento de conversaciones	33
2.4.8.	Puntos positivos y negativos de Rasa	34
3.	Teoría de agentes conversacionales	37
3.1.	Agente conversacional	37
3.1.1.	Modelo de comunicación	37
3.1.2.	Taxonomía de agentes conversacionales	38
3.1.3.	Arquitectura de un agente conversacional	40
3.2.	Expresividad y restricciones sobre los canales de comunicación	44
3.3.	Clasificación PENS de los lenguajes naturales	45
3.3.1.	Propiedades fundamentales	45
3.4.	Proceso de comprensión del lenguaje	46
3.4.1.	Las representaciones mentales de las entidades textuales	47
3.4.2.	Estimación de la intencionalidad	48
3.4.3.	Complejidad del proceso de comprensión	49
3.5.	Proceso <i>core</i>	50
3.6.	Proceso de generación del lenguaje	52
4.	Diseño del proceso de comprensión	53
4.1.	Preprocesamiento de lenguaje natural	53
4.2.	Segmentación	54
4.2.1.	A nivel de enunciado	54
4.2.2.	Problemas de la segmentación a nivel de palabra	54
4.3.	Normalización	55
4.3.1.	Mayúsculas y minúsculas	56
4.3.2.	Signos diacríticos	56
4.3.3.	Emoticonos y emojis	56
4.3.4.	Stemming	56
4.3.5.	Lematización	57
4.4.	Filtrado	57
4.4.1.	Palabras vacías (Stop words)	57

4.5.	Etiquetado gramatical	58
4.6.	Codificación	59
4.6.1.	Bag-of-words	60
4.6.2.	TF-IDF	61
4.6.3.	Word2vec	62
4.6.4.	Representación distribuida contextual: ELMo y BERT	63
4.7.	Preprocesamiento de mensajes usando nuestro <i>framework</i>	64
4.7.1.	Transformaciones a nivel de caracter	64
4.7.2.	Modelo de <i>token</i>	65
4.7.3.	Segmentación	66
4.7.4.	Características de los <i>tokens</i>	69
4.7.5.	Servicios de codificación: modelos de representación clásicos	71
4.7.6.	Servicios de codificación: modelos de representación distribuidos	72
4.8.	Clasificación de intenciones	75
4.8.1.	Introducción	75
4.8.2.	Clasificación con máquinas de soporte vectorial	76
4.8.3.	Servicio de clasificación de intenciones	77
4.9.	Clasificación del sentimiento	79
4.9.1.	Introducción	79
4.9.2.	Clasificación con Naive Bayes	81
4.9.3.	Clasificación con regresión logística	81
4.9.4.	Servicios de clasificación del sentimiento	82
4.10.	Reconocimiento de entidades nombradas	84
4.10.1.	Introducción	84
4.10.2.	Problema de etiquetado	85
4.10.3.	Normalización de menciones	86
4.10.4.	Clasificación de roles y agrupamiento de entidades	87
4.10.5.	Uso de léxicos y expresiones regulares para la detección de entidades	87
4.10.6.	Modelos ocultos de Markov	88
4.10.7.	Modelos de máxima entropía de Markov	89
4.10.8.	Campos aleatorios condicionales (CRF)	90

4.11. Servicio de etiquetado de secuencias	91
4.12. Servicio de clasificación de roles y agrupamiento de entidades	92
4.13. Servicio de extracción de entidades nombradas	92
4.14. Servicios de normalización de entidades	93
4.14.1. Corrección de posibles erratas	93
4.14.2. Sinónimos	95
5. Diseño del proceso <i>core</i>	97
5.1. Autómatas finitos	97
5.2. Diálogo guiado por <i>frames</i>	98
5.3. Gestión de diálogo basado en transformadores	99
5.3.1. Tópicos de una conversación	99
5.3.2. Política de diálogo TED	99
5.4. Lógica del agente conversacional en nuestro <i>framework</i>	100
5.4.1. Estímulos	100
5.4.2. Estado interno	102
5.4.3. Preprocesamiento en <i>core</i>	103
5.4.4. <i>Slot filling</i>	104
5.4.5. Servicio de lógica del agente	108
5.4.6. Modelo de comportamiento basado en recuerdos	110
5.4.7. Modelo de comportamiento basado en reglas	113
5.4.8. Modelo de comportamiento basado en flujos	116
5.4.9. Modelo de comportamiento basado en modelos estadísticos	117
5.4.10. Servicio de ejecución de acciones	121
5.5. Generación de lenguaje natural	123
5.5.1. Introducción	123
5.5.2. Enfoque de nuestro <i>framework</i> para la generación de texto	125
5.5.3. Definición de una plantilla desde código	125
5.5.4. Definición YAML de una plantilla	129
5.5.5. Servicio de selección de plantillas	131
5.6. Otros servicios genéricos	132

6. Diseño del agente conversacional	133
6.1. Arquitectura de los procesos	133
6.2. Configuración del proceso de comprensión	138
6.3. Configuración del proceso <i>core</i>	138
6.4. Creación de un agente conversacional	139
6.4.1. Integración con Telegram	140
6.5. Caso práctico: Bot COVID-19	141
6.6. Preprocesamiento	141
6.7. Clasificación de intenciones	142
6.7.1. Entidades	145
6.8. Lógica del agente	147
6.8.1. Modelo de comportamiento basado en reglas	147
6.9. Modelo de comportamiento basado en flujos	149
6.10. Modelos de comportamiento basado en modelos estadísticos y recuerdos	152
6.11. Acciones	154
6.12. Construcción del <i>bot</i>	155
6.13. Ejemplo de conversación completa	157
7. Conclusiones y Líneas Futuras	163
7.1. Conclusiones	163
7.2. Líneas Futuras	164
7.2.1. Intenciones múltiples	164
7.2.2. Diseño antropomórfico y vínculo con el usuario	164
7.2.3. Abstracción de estímulos	166
7.2.4. Estímulos complejos e internos	166
Referencias	169

1

Introducción

1.1. Motivación

Desde la segunda mitad del siglo XIII, la humanidad ha presenciado un proceso continuo y sistemático de automatización de tareas que, en un principio, eran desempeñadas por personas, generalmente, cualificadas. Invenciones como la máquina de vapor, el motor de combustión interna, las TICs, en general, por citar algunas de ellas, tuvieron, tienen y seguirán teniendo un gran impacto positivo en nuestra sociedad y, especialmente, en nuestra economía (Fremdling, 1996, pp. 1 - 4).

En los últimos años, se está observando una tendencia a la automatización de tareas que requieren un grado de inteligencia propio de un ser humano. Esta tendencia es posible gracias a que el campo de la inteligencia artificial nos ofrece las herramientas necesarias para acometer lo que algunos expertos consideran la Cuarta Revolución Industrial (Ross y Maynard, 2021).

Si bien es cierto que determinados sectores y grupos profesionales pueden verse afectados por el uso de la inteligencia artificial para la automatización de ciertas tareas, hemos de tener una visión general y percatarnos de que este tipo de procesos está asociado a un crecimiento de la productividad, salarios más elevados para las nuevas profesiones que aparecen y un crecimiento del producto interior bruto (Acemoglu y Restrepo, 2018, pp. 6-9).

El presente trabajo de fin de grado nace del interés en realizar una aportación a este proceso de evolución tecnológica industrial mediante la elaboración de una teoría de agentes conver-sacionales que proporcione un marco teórico común para el estudio de sus propiedades, y la implementación de un *framework* en Python para demostrar la completitud, corrección y utilidad en la práctica de la teoría.

1.2. Objetivos

- Proponer una teoría de agentes conversacionales que permita estudiar sus propiedades y la complejidad inherente de los problemas que se intentan abordar durante su desarrollo.
- Describir técnicas, algoritmos y modelos utilizados para el preprocesamiento, comprensión y generación de lenguaje natural.
- Describir técnicas, algoritmos y modelos utilizados para la gestión del diálogo y el estado interno de los agentes.
- Proporcionar y describir la implementación de servicios cognitivos basados en las técnicas, algoritmos y modelos expuestos.
- Proporcionar la implementación de arquitecturas de agentes conversacionales.
- Exponer un caso práctico en el que se diseñe e implemente un agente conversacional utilizando los conceptos introducidos en la teoría y su correspondiente implementación en nuestro *framework* en Python.

1.3. Estructura del documento

En este capítulo introductorio se ha introducido la motivación del trabajo de fin de grado en cuestión y los objetivos que se definieron a la hora de plantearlo.

En el capítulo 2 describiremos algunos de los agentes conversacionales clásicos más conocidos, analizaremos el funcionamiento general del servicio de Alexa de Amazon, y describiremos algunos de los motores de procesamiento de lenguaje natural y *frameworks* disponibles para el desarrollo de agentes conversacionales.

Tras haber introducido el estado del arte de los *chatbots*, procederemos en el capítulo 3 a definir nuestra teoría de agentes conversacionales.

Las técnicas, algoritmos y modelos utilizados para el preprocesamiento y comprensión del lenguaje natural serán descritos en el capítulo 4 de diseño del proceso de comprensión del agente. Asimismo, expondremos los servicios cognitivos de preprocesamiento, clasificación de intenciones, de sentimiento, extracción de entidades, entre otros.

En el capítulo 5 se describirá en detalle los algoritmos y técnicas utilizados para la lógica del agente conversacional. Introduciremos los conceptos de estado interno y cuatro modelos de comportamiento: basado en reglas, basado en recuerdos, basado en flujos y en modelos estadísticos de Keras.

Dedicaremos el capítulo 6 a la creación de un agente a partir de sus procesos cognitivos. Además de lo anterior, expondremos un caso práctico de agente conversacional para el COVID-19 para el cual hemos utilizado los conceptos teóricos introducidos en el capítulo 3 y los servicios cognitivos de los capítulos 4 y 5.

1.4. Metodología de trabajo

Los hechos expuestos en la sección 1.1 nos motivaron a decidir que era necesario realizar una aportación teórico-práctica al campo del diseño e implementación de agentes conversacionales.

La modelización matemática de cualquier fenómeno es un proceso cíclico en el que a partir de la información del mundo real se crea un modelo matemático, se valida con el fenómeno real y se propone uno nuevo que se ajuste mejor que el modelo matemático anterior. En el caso que nos ocupa, los fenómenos a modelar son la comprensión del lenguaje natural, la planificación de una reacción ante un estímulo, y la generación del lenguaje natural. Nuestras fuentes teóricas sobre estos procesos cognitivos y su posible simulación pertenecen a los siguientes campos:

- **Lingüística:** La morfosintaxis y semántica son ramas de la lingüística fundamentales para abordar el estudio de los fenómenos de comprensión y generación del lenguaje natural. Al tratarse de un modelo agnóstico del lenguaje, estudiaremos las propiedades universales de los lenguajes. Además de esto, investigaremos sobre cómo las estructuras morfosintácticas codifican las ideas que transmitimos.
- **Neurociencia, psicología y filosofía:** Los fenómenos a modelar se pueden describir siguiendo un enfoque de alto o bajo nivel. La neurociencia nos permitirá demostrar que para ciertos elementos de la teoría propuesta como los clasificadores de intenciones se han descubierto tipos de neuronas o redes neuronales naturales que pueden ser la base de sistemas naturales equivalentes a los propuestos en la teoría. La psicología y filoso-

fia nos permitirá definir fenómenos complejos con conceptos abstractos de alto nivel como procesos, modelos de comportamiento o estados internos, facilitando la posterior implementación práctica de la teoría en forma de *framework* en Python.

- **Inteligencia artificial:** La implementación de los procesos y servicios cognitivos requerirá el uso de aprendizaje automático. Investigaremos los modelos y técnicas disponibles, sus propiedades y su idoneidad para problemas relacionados con el procesamiento de lenguaje natural.

Además de llevar a cabo una revisión bibliográfica sobre los fenómenos a modelar y estudiar las conclusiones a las que llegan los investigadores de los campos mencionados anteriormente, investigaremos, siempre que sea posible, los detalles de implementación de algunos de los agentes conversacionales más conocidos a lo largo de la historia. Asimismo, utilizaremos varios *frameworks* y motores de comprensión de lenguaje natural para descubrir los elementos, técnicas y enfoques comunes utilizados por las distintas implementaciones.

Tal y como explicamos anteriormente, el modelo matemático será resultado de varias iteraciones en las que verificaremos si los elementos introducidos en la teoría son suficientes para describir, definir e implementar los procesos cognitivos necesarios para crear un agente conversacional. La etapa de validación se llevará a cabo de la siguiente forma:

1. Compararemos los elementos teóricos de la lingüística, neurociencia y psicología con los introducidos por nuestra teoría. La primera cuestión que nos plantearemos durante la validación será si el modelo obtenido hasta ahora es completo y compatible con otras teorías verificadas empíricamente.
2. En la segunda fase, intentaremos describir los agentes conversacionales del estado del arte con elementos de nuestra teoría. Comprobaremos nuevamente si el modelo es completo desde un punto de vista práctico. También evaluaremos la complejidad del modelo, esto es, estudiaremos si es necesario introducir suposiciones para facilitar la posterior implementación y entrenamiento de agentes.
3. En la tercera y última fase verificaremos empíricamente que nuestra teoría es correcta, implementando agentes conversacionales con nuestro *framework* en Python y evaluando su rendimiento.

1.5. Tecnologías utilizadas

■ Memoria:

- LaTeX para la redacción de la memoria.
- BibTeX para la gestión de las referencias bibliográficas.
- Inkscape para las imágenes de elaboración propia.

■ Código fuente:

- El lenguaje de programación escogido fue Python en su versión 3.7.
- El entorno de desarrollo o IDE fue PyCharm.
- Git para el control de versiones.
- PyTest para la ejecución de pruebas.
- Jupyter Notebook para la creación de cuadernos que muestren ejemplos de inicialización, entrenamiento y consumo de servicios cognitivos.
- Librerías:
 - SpaCy 2.3.5 para los modelos pre-entrenados de lenguajes.
 - Scikit-learn 0.21.3 para los modelos de máquina de soporte vectorial, regresión logística, Naive Bayes y algunas técnicas de preprocesamiento.
 - El módulo sklearn-crfsuite en su versión 0.3.6 para el entrenamiento de campos aleatorios condicionales.
 - Tensorflow 2.1.0 para definir, compilar, entrenar y utilizar la arquitectura pre-determinada de red neuronal para el modelo de comportamiento basado en modelos estadísticos.
 - La librería Gensim 4.0.1 para la creación de un servicio de codificación basado en Doc2Vec.
 - La librería simple-elmo en su versión 0.8.0 para el consumo de modelos pre-entrenados de ELMO.
 - Para el despliegue en Telegram hemos recurrido a la librería python-telegram-bot en su versión 13.7.

2

Estado del Arte

2.1. Evolución histórica de los agentes conversacionales

2.1.1. Test de Turing (1950)

Turing (1950, p. 433) plantea en su ensayo 'Computing Machinery and Intelligence' una formalización alternativa al problema de si una máquina puede llegar a pensar. Se introduce un problema denominado *Imitation game*, por el cual una persona X tiene que comunicarse con dos entidades A y B , y decidir a cual asignarle una etiqueta T .

En una instancia del problema del juego de imitación, una de estas dos entidades puede ser un *bot* C y la otra un ser humano H . La persona X debe asignar una etiqueta 'Es un humano' a una de estas dos entidades. Dado que la comunicación se produce a través de un canal que solo permite transmitir información de forma textual, X no puede recurrir a información visual o sonora para tomar una decisión (Turing, 1950, p. 434).

El test de Turing puede ser interpretado como una prueba para verificar que un agente conversacional de propósito general tiene el grado de inteligencia necesario para poder comunicarse con los usuarios de forma efectiva en un dominio general y amplio. Dado que los agentes conversacionales desarrollados por las empresas suelen tener un dominio reducido y bien definido, las pruebas para verificar estos agentes serán equivalentes a un test de Turing limitado a un dominio dado.

2.1.2. STUDENT (1964)

En 1964, el científico informático estadounidense Daniel Gureasko Bobrow, desarrolla un programa denominado STUDENT que es capaz de resolver problemas de álgebra expresados en lenguaje natural (Bobrow, 1964).

Al introducir el usuario el enunciado de un problema de álgebra en lenguaje natural, se lle-

va a cabo una serie de transformaciones sobre la cadena de entrada para expresarla haciendo uso del lenguaje relacional. Posteriormente, se aplican técnicas de deducción para descubrir relaciones implícitas (resolución de sistemas de ecuaciones) y resolver así el problema (Bobrow, 1964, p. 8).

A continuación, exponemos brevemente el algoritmo de preprocesamiento utilizado por STUDENT para transformar un mensaje en lenguaje natural en un sistema de ecuaciones:

- **Sustituciones obligatorias:** Se aplican sustituciones en la cadena original en lenguaje natural (Bobrow, 1964, p. 12). Algunos ejemplos de sustituciones obligatorias son twice → 2 times y per cent → percent.
- **Etiquetado:** Se intenta etiquetar cada palabra haciendo uso de un diccionario, que dada una palabra devuelve una etiqueta y si se trata de un operador, también el nivel del mismo, esto es, el número de parámetros (Bobrow, 1964, p. 12). Por ejemplo, square → (square / OP 1) y percent → (percent / OP 2).
- **Sentencias simples:** STUDENT aplica recursivamente una serie de reglas para obtener una lista de oraciones simples (Bobrow, 1964, pp. 13-14).
- **Mapeo al lenguaje relacional:** Se realiza una búsqueda de izquierda a derecha para encontrar el operador que admite el mayor número de parámetros. Si no se encuentra ninguno, se considera toda la oración una entidad indivisible, una variable. En otro caso, se construye la ecuación dada por una tabla en función del operador y de su contexto en la cadena (Bobrow, 1964, pp. 15-16).

2.1.3. ELIZA (1964-1966)

Entre 1964 y 1966, Joseph Weizenbaum, profesor de informática del Instituto Tecnológico de Massachusetts (MIT), desarrolla el programa ELIZA, que permite al usuario mantener una conversación por escrito con un agente conversacional que simula a un psicoterapeuta Rogeriano, desviando la atención al usuario haciéndole preguntas abiertas (Weizenbaum, 1966).

Cuando un usuario introduce un mensaje en lenguaje natural, ELIZA intenta obtener una *keyword* o palabra clave del mismo. Mientras hace una lectura de izquierda a derecha del mensaje, se tiene en cuenta el RANK o precedencia de las palabras claves para determinar cual es

la más importante. Si ninguna de las subcadenas delimitadas por delimitadores tiene asociada una palabra clave, responde con un mensaje genérico o repite uno reciente (Weizenbaum, 1966).

Una vez que se ha detectado la palabra clave de un mensaje, se procede a comprobar si la estructura del mensaje es capturada por alguna de las reglas de descomposición asociadas a la palabra clave detectada. Estas reglas son expresiones regulares restringidas a dos tipos de símbolos:

- **Palabras.** Por ejemplo, el símbolo YOU captura exclusivamente la cadena YOU.
- **Números naturales.** Un número natural $n \in \mathbb{N}$ mayor que cero captura cadenas de n palabras consecutivas. El símbolo 0 captura cadenas de 0 o más palabras consecutivas.

La aplicación de estas reglas genera una lista de tokens que son introducidos en plantillas conocidas como reglas de reensamblado, que en ocasiones requieren la aplicación de una transformación previa a ciertos tipos de tokens: pronombres, determinantes posesivos, etc. La respuesta del agente es el resultado de rellenar los *slots* de estas plantillas.

Ejemplo 2.1

Supongamos que la regla de descomposición (YOU 0 ME) está asociada a la plantilla (WHAT MAKES YOU THINK I 1 YOU).

Si el usuario introduce el mensaje 'YOU LOVE ME' y ELIZA selecciona la regla (YOU 0 ME), la lista de tokens resultante será [YOU, LOVE, ME] y el resultado de aplicar la regla de descomposición asociada será 'WHAT MAKES YOU THINK I LOVE YOU'. Nótese que el *slot* 1 de la plantilla ha sido instanciado con el segundo *token* del mensaje del usuario.

2.1.4. SHRDLU (1968-1970)

A finales de los 70, Terry Winograd desarrolla un programa denominado SHRDLU que es capaz de mantener un diálogo con el usuario sobre un mundo bastante pequeño y simple constituido por figuras geométricas (McTear, 2004). El usuario podía hacer consultas al programa en lenguaje natural sobre el estado actual del mundo. También era posible solicitarle a SHRDLU que ejecutara ciertas acciones como colocar una figura sobre otra. A la hora de especificar las entidades sobre las que se debía ejecutar la acción, SHRDLU era capaz de resolver

referencias a entidades ya mencionadas anteriormente y, en el caso de que la petición fuera ambigua, solicitaba al usuario que concretara un poco más a qué entidad o entidades se refiere. SHRDLU aplicaba dos tipos de análisis a la entrada del usuario:

- **Análisis sintáctico:** Se etiquetan las palabras del mensaje del usuario y se construye un árbol sintáctico.
- **Análisis semántico:** Una vez construido el árbol semántico se pueden hacer comprobaciones semánticas para determinar si ciertas preguntas o peticiones de acciones tienen sentido.

2.1.5. PARRY (1971-1972)

Kenneth Colby, psiquiatra americano, desarrolla PARRY entre 1971 y 1972 en la Universidad de Stanford. PARRY es un *bot* conversacional con una estructura patrón-respuesta similar a la de ELIZA, pero que además incluye características más avanzadas como un modelo mental (Jurafsky y Martin, 2020, p. 500). PARRY simula a un paciente con esquizofrenia paranoide.

Cuando el usuario introduce un mensaje, se le asigna una conceptualización, esto es, se clasifica el mensaje como pregunta, comentario sobre un tópico, etc. Se define un sistema de reglas que clasifica un mensaje como benevolente, neutral o malévolo partiendo de la conceptualización inicial y aplicando una serie de reglas de detección (Jurafsky y Martin, 2020, p. 500).

Tras llevar a cabo esta clasificación, se procede a actualizar el estado emocional de PARRY, que está constituido por tres variables: miedo, ira y desconfianza. Posteriormente, se selecciona una estrategia para generar un mensaje.

2.1.6. Era de los Asistentes de voz (2009-Actualidad)

Un asistente de voz es un software capaz de comprender mensajes en lenguaje natural transmitidos por voz y de responder a través de un sintetizador de voz (Hoy, 2018).

Las tareas básicas que suelen desempeñar estos asistentes son la planificación de eventos en el calendario del usuario, establecimiento de alarmas, respuestas a preguntas básicas como '¿Va a llover hoy?', control de otros dispositivos como Smart TVs, etc.

Uno de los primeros asistentes de voz en salir al mercado fue Siri de la compañía Apple que fue lanzado en 2010 como *app* y en 2011 se integró en el sistema operativo iOS. Dos años más tarde, Microsoft introduce su asistente Cortana. En 2014 y 2016 se introducen en el mercado Alexa (Amazon) y el asistente de Google, respectivamente (Hoy, 2018). En un estudio de 2019 se comparó el rendimiento de estos agentes conversacionales y se llegó a la conclusión de que la calidad de las respuestas de Alexa y Google era superior a las de Siri y Cortana a finales de ese año (López, Díaz, Quesada, y Guerrero, 2019).

Normalmente, todos estos tipos de asistentes llevan a cabo un procesamiento de los mensajes en sus servidores, lo cual ha planteado ciertas cuestiones sobre la privacidad de los usuarios (Bolton, Dargahi, Belguith, Al-Rakhami, y Sodhro, 2021).

2.1.7. Amazon Alexa (2014-Actualidad)

Alexa es un servicio en la nube proporcionado por Amazon para la creación de experiencias de usuario en las que la interacción con la tecnología tiene lugar a través del lenguaje natural (*Amazon Alexa Voice AI, s.f.*).

A través del *framework Alexa Skills Kit (ASK)* podemos crear *skills* para Alexa. Una *skill* es una funcionalidad que un usuario puede invocar a través de un mensaje en lenguaje natural a Alexa. Los usuarios pueden interactuar con Alexa enviando un mensaje de voz que empiece por la *wake word* 'Alexa'. El servicio de Alexa se encarga de transformar el mensaje de voz a un mensaje textual al que se le aplica un procesamiento NLU. Finalmente, se ejecuta la *skill* correspondiente enviándose una petición al servicio correspondiente (*Amazon Skills Kit, s.f.*).

Cada *skill* tiene un modelo de interacción que define qué palabras y frases activan la *skill* (*Alexa Voice Interaction Models, s.f.*). A la hora de crear un modelo de interacción propio hay que definir los siguientes parámetros:

- El nombre de la *skill*. El usuario la debe mencionar antes de hacer la petición. Por ejemplo, en 'Alexa, ask Plan My Trip to plan a trip to Madrid on Saturday' el usuario primero utiliza la *wake word*, después nombra la *skill* 'Plan my trip', y finalmente hace una petición.
- La intención del usuario. Cada funcionalidad específica de la *skill* puede tener asociada una intención.

Train Your App
Add a sample utterance and specify an intent. You can also highlight words or phrases in the utterance to annotate.

[See how it works](#)

Utterance ⓘ
“ Buenas, me llamo **Manuel** y soy de **España** 241

Intent ⓘ greeting Out of Scope ⓘ

Entity	Role	Resolved value	Confidence
person	user_name	Manuel	N/A
country	birthplace	España	N/A

[+ Add Trait](#)

[Train and Validate](#)

Figura 1: Creación de una instancia de entrenamiento NLU en Wit.ai.

- Los *slots* o argumentos del mensaje.
- Mensajes de ejemplos que tienen la intención anterior.

2.2. Motor de comprensión de lenguaje natural: Wit.ai

Wit.ai es un SaaS propiedad de Facebook para el desarrollo de aplicaciones de procesamiento de lenguaje natural. Su motor de procesamiento NLU soporta más de 130 idiomas para los cuales existen modelos pre-entrenados capaces de detectar entidades o intenciones *built-in* (Wit.ai, s.f.).

En Wit.ai podemos crear aplicaciones públicas o privadas para cada uno de los idiomas soportados. La página principal de la aplicación es un formulario en el que podemos introducir instancias de entrenamiento como se puede apreciar en la imagen 1. El campo `utterance` es el contenido textual del mensaje, el cual tiene un límite de 280 caracteres. En un desplegable podemos seleccionar una intención para el mensaje del espacio de intenciones definido hasta el momento. Las intenciones se pueden definir a demanda o bien desde la sección 'Intents'. (Wit.ai Recipes, s.f.).

Para mencionar las entidades presentes en el mensaje basta con seleccionar con el cursor una secuencia de caracteres del mensaje y seleccionar un tipo de entidad de los disponibles

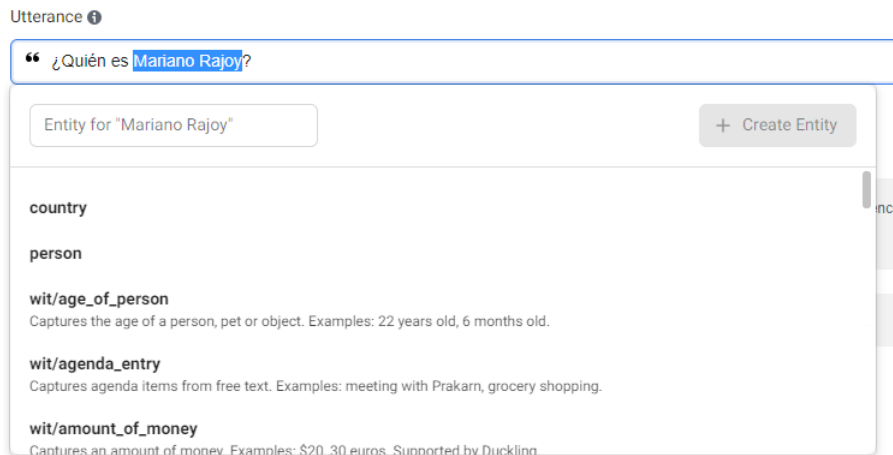


Figura 2: Selección del tipo de entidad en Wit.ai.

en una lista que aparecerá tras seleccionar una subcadena. Tal y como podemos apreciar en la figura 2, Wit.ai nos permite seleccionar un tipo de entidad *built-in* o uno definido por nosotros en una instancia anterior. Al igual que las intenciones, podemos crear un nuevo tipo de entidad desde ese mismo formulario (*Wit.ai Recipes*, s.f.).

Las entidades pueden tener roles asociados como en Rasa. No encontramos en la documentación y formulario ninguna referencia al agrupamiento de entidades en grupos.

Otra característica que incluye Wit.ai son los denominados *traits*. Un *trait* es un valor asociado al mensaje como puede ser el sentimiento. Wit.ai ofrece *built-in traits* para algunos idiomas como `wit/sentiment` y `wit/on_off` para el inglés. La tarea de entrenar los clasificadores de intenciones y *traits*, y los extractores de entidades y roles son gestionados por Wit.ai y no disponemos de un control casi absoluto de los parámetros del entrenamiento como en Rasa y otros *frameworks*.

2.3. Bot Framework Composer

Microsoft Bot Framework es un SDK para el desarrollo, *testing* y administración de *bots* inteligentes (*Microsoft Docs*, s.f.). Si bien no se exige que nuestro asistente se ejecute en el ecosistema de computación en la nube Azure de Microsoft, la integración de agentes de Bot Framework con otros servicios de Microsoft sería un proceso sencillo.

2.3.1. Servicio de comprensión de lenguaje natural LUIS

Language Understanding (LUIS) es un servicio basado en la nube que puede llevar a cabo un procesamiento de lenguaje natural de un mensaje (*Microsoft LUIS Docs, s.f.*). Al igual que otras tecnologías NLU, el servicio LUIS extrae intenciones y entidades de un mensaje. Por defecto, la intención devuelta por el servicio es la más probable, pero es posible obtener todas las puntuaciones de las entidades a través del parámetro `show-all-intents` de la petición que se envía al servicio. Se reserva la intención None para mensajes que no sean relevantes para la conversación.

Hemos de mencionar que LUIS tiene un límite de 500 intenciones por aplicación, por lo que a la hora de elegir la tecnología o *framework* a utilizar para diseñar un agente, debemos tener en cuenta la complejidad del mismo y los límites de cualquiera de los servicios de IA en la nube.

En cuanto a las entidades, podemos definir varios tipos:

- **Entidad de lista:** Es una entidad en su forma canónica que tiene asociada una lista de alternativas posibles. Por ejemplo, una ocurrencia 'mini' podría extraerse como entidad por formar parte de una lista, pero se normalizaría a 'pequeña', al ser esta la forma canónica de los elementos de esa lista.
- **Entidad de expresión regular:** Es una entidad que ha sido extraída por coincidir con una expresión regular que hemos definido.
- **Entidad pre-generada:** Son entidades comunes como fechas o nombres que LUIS ya aporta para que no tengamos que añadirlas al dominio del agente.
- **Entidad Pattern.any:** Supongamos que tenemos un patrón 'Hola, me llamo Name'. La entidad 'Manuel' de tipo Name sería extraída del mensaje 'Hola, me llamo Manuel' por la aplicación del patrón anterior. Estas entidades son el resultado de aplicar una plantilla en sentido inverso, esto es, se extrae como entidad la subcadena que al ser introducida en el patrón daría lugar al mensaje recibido.
- **Entidad extraída mediante etiquetado y aprendizaje automático.**

2.3.2. Arquitectura

Bot Framework Composer es un IDE desarrollado sobre Bot Framework que ofrece herramientas gráficas para definir las experiencias conversacionales (*Bot Framework Composer Concepts, s.f.*). La arquitectura de un diálogo en Bot Framework Composer es la siguiente:

- **Recognizer:** Es un proceso de extracción de intenciones o entidades. Podríamos recurrir a LUIS como recognizer.
- **Triggers:** Son reglas que se ejecutan al darse ciertas condiciones y dan lugar a la ejecución de una o más acciones.
- **Actions:** Una acción puede ser almacenar un valor en memoria, responder al usuario haciendo una pregunta, hacer una llamada a una API, registrar en el *log* cierta información, etc.
- **Language generator:** A partir de plantillas crea mensajes en lenguaje natural.

Además de lo anterior, Bot Framework Composer dispone de un sistema de memoria para almacenar el estado actual de la conversación. Aunque el estado tiene propiedades *built-in* que son mantenidas por el *framework*, es posible almacenar nuestras propias propiedades. Una propiedad está constituida por un *scope* y nombre. El *scope* determina cuándo estará disponible su valor y cuánto tiempo se preservará en la memoria. Por ejemplo, el *scope* `user` no expira, mientras que `conversation` expira al terminar la conversación (*Bot Framework Composer Concepts, s.f.*).

Se pone a disposición del desarrollador varias acciones para manipular la memoria del agente que son accesibles desde los gráficos en los que podemos definir conversaciones de forma gráfica. También es posible utilizar los valores de las propiedades de la memoria en estructuras de control IF-ELSE, SWITCH o bucles. Otro aspecto a destacar es que podemos crear acciones que envíen una respuesta parametrizada donde los *slots* son los identificadores de las propiedades de la memoria.

2.3.3. Diálogos

Un *bot* está constituido por uno o más diálogos que a su vez están constituidos de uno o más flujos que pueden ejecutarse a través de un *trigger* (*Bot Framework Composer Dialogs, s.f.*).

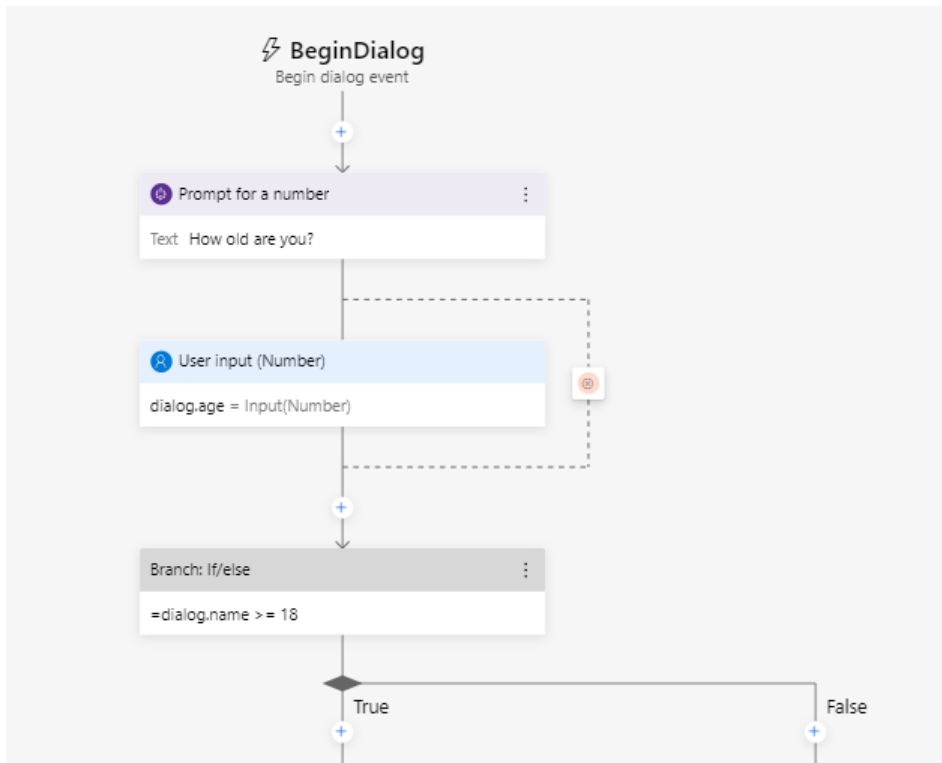


Figura 3: Ejemplo de solicitud de información con Bot Framework Composer.

Un flujo es un grafo en el que los nodos son acciones del agente o eventos. Por ejemplo, en la imagen 3 podemos ver que se utilizan dos nodos para representar una pregunta del asistente y una respuesta del usuario. El segundo nodo azul es dependiente del primero que es un nodo de tipo Prompt numérico, es decir, una acción de petición de un dato de tipo numérico.

Tal y como podemos apreciar en la figura 4, al seleccionar un nodo podemos configurar sus propiedades en el menú lateral derecho. En el caso de un nodo Prompt, es posible especificar una lista de preguntas para obtener el dato, reglas de validación del dato introducido, posibles mensajes para informar de un dato inválido, listas de textos de ejemplo donde se especifica la entidad a extraer por el *recognizer* utilizado, etc.

A través de nodos de gestión de diálogo podemos ir de un flujo a otro en función del comportamiento diseñado para el asistente.

2.3.4. Puntos positivos y negativos de Bot Framework Composer

La principal ventaja de utilizar Bot Framework Composer es que podemos consumir un servicio de comprensión de lenguaje natural proporcionado por Microsoft y centrarnos en la

Prompt for text

Text Input

Collection information - Ask for a word or sentence.

[Learn more](#)

Add a note

Bot response

User input

Other

Ask a question - text ⓘ

[Show code](#)

Text +

Responses ⓘ

⋮

- What's your name?

- Can you tell your name?

- So how should I call you?

[Add alternative](#)

Figura 4: Los nodos en Bot Framework Composer pueden ser configurados a través de un menú lateral. En la imagen se muestra las opciones disponibles para configurar la respuesta del agente en un nodo Prompt de tipo textual.

lógica del agente conversacional. Si el desarrollador no tiene conocimientos avanzados de inteligencia artificial, puede ser una opción para desarrollar soluciones empresariales o personales sencillas.

Las herramientas gráficas permiten visualizar y diseñar la lógica del agente de forma completamente visual y sin programar una sola línea de código. Si bien es cierto que esto reduce la pendiente de la curva de aprendizaje, debemos decir que la multitud de opciones, propiedades, nodos de acción y sobre todo, aspectos de configuración hacen que el desarrollador necesite más tiempo que con otras herramientas para crear la lógica, puesto que hay muchas alternativas y opciones.

Un aspecto negativo que deberíamos mencionar es que aunque se provee de mecanismos para lidiar con respuestas inesperadas a preguntas del agente, no se proporcionan mecanismos avanzados de gestión de diálogo basados en aprendizaje automático. Por lo anterior, Composer puede ser una herramienta adecuada para crear agentes conversacionales sencillos, pero para crear agentes más complejos y próximos a los de propósito general se necesitará recurrir a *frameworks* como el que introducimos en la siguiente sección.

2.4. Framework Rasa para la creación de *chatbots*

Rasa es un *framework* de código abierto en Python que recurre al aprendizaje automático para la automatización de las conversaciones ([Rasa Docs, s.f.-a](#)).

La arquitectura de los agentes conversacionales de Rasa está constituida por dos componentes: un módulo de comprensión de lenguaje natural y un gestor de diálogo. El primer módulo lleva a cabo tareas de clasificación de intenciones y extracción de entidades, entre otras tareas NLU. El segundo módulo decide la siguiente acción en la conversación en base al contexto ([Rasa Architecture, s.f.](#)).

2.4.1. Dominio del agente

En Rasa el espacio de intenciones es finito y forma parte de lo que se denomina el dominio del asistente. Se puede configurar una intención para ignorar todas las entidades, solo algunos tipos de entidades o capturar todas. Esto es de utilidad para aplicar un filtrado de entidades previo al *slot filling* ([Rasa Domain, s.f.](#)).

Además de las intenciones ha de definirse una lista de tipos de entidades para la tarea de extracción de entidades nombradas. Se puede especificar que ciertos tipos soportan roles o grupos. Los conceptos de roles y grupos se han introducido en nuestro *framework* y se explicarán en la sección [4.10.4](#).

El *slot filling*, esto es, la vinculación de las entidades extraídas en su mensaje con *slots* predefinidos también es posible en Rasa, pero requiere incluir en el dominio del agente una lista de *slots* con sus respectivos tipos. Dado que el hecho de que se ha rellenado un *slot* dado puede ser relevante para la gestión de diálogo, es necesario especificar si el *slot* se tendrá en cuenta o no para la decisión de la siguiente acción.

Otro elemento fundamental para el dominio es la definición de los identificadores de las acciones. Entre las acciones destacamos las respuestas que son mensajes que el asistente envía al usuario (*Rasa Actions, s.f.*). Las respuestas pueden ser incluidas directamente en el fichero YAML que define el dominio completo del agente. En el fragmento de código [1](#) podemos ver un ejemplo de definición del dominio de un agente en Rasa.

```
1 intents:
2   - greet:
3   - farewell
4
5 entities:
6   - PERSON
7   - city:
8     roles:
9       - from
10      - to
11 slots:
12   location:
13     type: text
14
15 responses:
16   utter_greet:
17     - text: "Hello!"
18   utter_goodbye:
19     - text: "Bye!"
20
21 actions:
```

```
22 - action_search_restaurants
23 - action_search_hotels
```

Código 1: Ejemplo de definición del dominio de un agente en Rasa.

2.4.2. Pipeline NLU

Rasa ofrece varios componentes NLU para la construcción de una *pipeline* personalizada. A continuación describimos brevemente los distintos tipos de componentes (*Rasa Componentes*, s.f.):

- **Modelo de lenguaje:** Componentes que cargan un modelo pre-entrenado necesario para utilizar vectores pre-entrenados para las palabras. Se puede importar un modelo de SpaCy o MITIE.
- **Tokenizers:** Segmentan los textos en *tokens*. Para los lenguajes segmentados se proporcionan componentes de segmentación por espacio y segmentación usando uno de los modelos de lenguaje importados.
- **Featurizers:** Crean una representación vectorial de un mensaje.
- **Clasificadores de intenciones:** Clasifican la intención del mensaje. Los componentes relacionados suelen recurrir a una máquina de soporte lineal.
- **Extractores de entidades:** Se pueden combinar extractores de entidades especializados en un subconjunto de entidades. El extractor Duckling, por ejemplo, nos permite extraer fechas, distancias, etc., mientras que podemos recurrir a un extractor basado en campos aleatorios condicionales para extraer los nombres. Es posible recurrir a los extractores de entidades proporcionados por los modelos de lenguaje.
- **Otros:** existen otros componentes como EntitySynonymMapper que permite mapear los sinónimos de una entidad a una forma canónica, DIETClassifier para efectuar la clasificación de intenciones y extracción de entidades de forma simultánea con una misma arquitectura de transformador con una capa de campo aleatorio condicional.

Esta *pipeline* se define en un fichero YAML `config.yml` como una lista de componentes NLU asociados a la clave `pipeline`. El fragmento de código 2 es un ejemplo de definición de *pipeline* con dos etapas.

```
1 pipeline:
2   - name: "SpacyNLP"
3     model: "en_core_web_md"
4     case_sensitive: False
5   - name: "SklearnIntentClassifier"
6     C: [1, 2, 5, 10, 20, 100]
7     kernels: ["linear"]
8     "gamma": [0.1]
9     "max_cross_validation_folds": 5
10    "scoring_function": "f1_weighted"
```

Código 2: Ejemplo de definición de *pipeline* NLU en Rasa.

2.4.3. Políticas

Las políticas son utilizadas en Rasa para predecir la siguiente acción a tomar en el contexto de la conversación (*Rasa Policies, s.f.*). En el caso de que más de una política haga una predicción con una misma confianza, se elige la acción generada por aquella que tenga más prioridad. Las políticas *built-in* de este *framework* son las siguientes:

- **RulePolicy:** Para definir comportamientos que siempre deberían darse si se dan una serie de condiciones.
- **MemoizationPolicy:** El agente inspecciona si el contexto actual coincide con el de alguna historia del fichero `stories.yml`. En caso afirmativo, ejecuta con una confianza absoluta de 1 la siguiente acción especificada por la historia.
- **TEDPolicy:** Mientras que las anteriores políticas son deterministas, esta política recurre al aprendizaje automático para que el agente pueda actuar en situaciones desconocidas en las que ni las reglas ni las historias son de ayuda. Entramos más en detalle sobre esta política en la sección 5.3.2.

2.4.4. Forms

Rasa introduce el patrón de conversación de recolección de datos para un formulario a través de Forms. A través de una regla o historia se puede activar o desactivar un formulario que, por definición, es un conjunto de *slots* que son rellenados bajo ciertas condiciones y métodos (*Rasa Forms, s.f.*).

Un *slot* puede tener asociado más de un *slot mapping*. Un *slot mapping* define una serie de precondiciones que de cumplirse supondrían la asociación de un valor al *slot*. Un parámetro importante de un *mapping* es el tipo. Si el tipo es `from_entity`, se rellenará el *slot* con una entidad nombrada que tenga unas ciertas características; un *mapping* con el tipo `from_text` rellenará el *slot* con el texto introducido por el usuario, y `from_intent` permite rellenarlo con un valor dado si se da una cierta intención.

Existe un *slot* especial denominado `requested_slot` para almacenar el valor solicitado al usuario. Esta información permite al agente saber si una petición ha sido ignorada.

Rasa introduce mecanismos de validación para poder determinar si un *slot* ha sido rellenado con un valor válido o no. Es posible crear subclases de `FormValidationAction` para definir un método de validación.

2.4.5. Formato de datos de entrenamiento

Rasa recurre al formato YAML para la definición de los datos de entrenamiento NLU e historias. Un *dataset* puede ser creado con uno o varios ficheros YAML que pueden contener las claves `nlu`, `stories` y `rules` (*Rasa Training Data Format, s.f.*).

2.4.6. Datos de entrenamiento NLU

En el objeto con la clave `nlu` podemos añadir instancias de entrenamiento para la *pipeline* NLU agrupadas por intención. Es posible incluir en el texto de la instancia símbolos especiales que delimiten las entidades a extraer y que especifiquen su rol, grupo, etc. Asimismo, es posible introducir campos adicionales a través de una propiedad de la instancia denominada `metadata`.

- 1 nlu:
- 2 - intent: check_balance

```

3 examples: |
4 - how much do I have on my [savings]("account") account
5 - how much money is in my [checking>{"entity": "account"} account
6 - What's the balance on my [credit card account>{"entity": "account", "value": "credit"}

```

Código 3: Instancias de entrenamiento para una intención en Rasa.

Bajo la misma clave `nlu` es posible definir sinónimos, expresiones regulares y tablas de búsqueda para proporcionar información adicional para la extracción de entidades.

```

1 nlu:
2 - synonym: covid
3 examples: |
4 - coronavirus
5 - covid-19
6
7 - regex: account_number
8 examples: |
9 - \d{10,12}
10
11 - lookup: banks
12 examples: |
13 - JPMC
14 - Bank of America

```

Código 4: Definición de sinónimos, expresiones regulares y entidades de la tabla de búsqueda en Rasa.

2.4.7. Datos de entrenamiento de conversaciones

Las historias son ejemplos de conversaciones entre un usuario y un asistente. Los mensajes del usuario se representan como intenciones y entidades, y las respuestas del asistente son secuencias de acciones (*Rasa Docs, s.f.-b*). Con las historias podemos entrenar modelos de aprendizaje automático para descubrir los patrones que hay detrás de las conversaciones esperadas para ser capaz de generalizar (*Rasa Training Data Format, s.f.*).

Las historias están constituidas de pasos que pueden representar mensajes del usuario, acciones del asistente, activación o finalización de un formulario, eventos como el que tiene

lugar cuando un *slot* se ha rellenado, etc. Existe un tipo de paso denominado OR que permite especificar en un mismo paso varias posibilidades para el mensaje del usuario.

```
1 stories:
2 - story: Ejemplo de historia con OR
3   steps:
4     - intent: signup # Mensaje del usuario
5     - slot_was_set: # Evento de slot filling (se relleno 'user_name')
6       - user_name
7     - action: say_ask_confirmation # Accion del agente
8   - or:
9     - intent: yes
10    - intent: thanks
11  - action: action_signup
12  - active_loop: signup_form # se inicia el formulario
13  - active_loop: null # termina
```

Código 5: Ejemplo de historia de Rasa.

Además de historias se pueden definir reglas con condiciones para la política de reglas:

```
1 - rule: Only say `hey` when the user provided a name
2   condition:
3     - slot_was_set:
4       - user_provided_name: true
5   steps:
6     - intent: greet
7     - action: utter_greet
```

Código 6: Ejemplo de historia de Rasa con condiciones.

2.4.8. Puntos positivos y negativos de Rasa

Rasa proporciona una gran multitud de componentes que hacen de su módulo NLU un sistema lo suficientemente sofisticado y potente para ser utilizado incluso en aquellos casos en los que solo se quiere aplicar técnicas de comprensión de lenguaje natural. Además de estos componentes, debemos destacar lo sencillo que es de utilizar su formato YAML para configurar el asistente, definir la *pipeline* y los *datasets*.

En cuanto al sistema de diálogo, hemos de mencionar que se aporta una serie de políticas suficientes para construir un agente conversacional de una complejidad media. En general, Rasa es muy escalable al ser posible extender el código del *framework* fácilmente con nuestras propias políticas, formularios, etc. Además de esto, el hecho de que sea *open source* y gratuito lo convierten en una buena opción para el desarrollo de agentes conversacionales.

Rasa ofrece un conjunto de herramientas denominado Rasa X que nos permite mejorar un asistente construido con Rasa a través de la interacción con usuarios de prueba y usuarios reales (*Rasa Docs, s.f.-b*).

La principal desventaja de Rasa es que la curva de aprendizaje puede ser pronunciada al principio, aunque suele ser un problema inherente de aquellos sistemas o librerías que proporcionan muchos componentes, independientemente de si el uso de los mismos es sencillo o no. Otra desventaja frente a otras opciones disponibles como Wit.ai es la complejidad derivada del grado de personalización. Con otras tecnologías se pueden crear agentes conversacionales con interfaces gráficas de forma más sencilla que en Rasa. No obstante, la facilidad de uso sacrifica la capacidad de crear comportamientos complejos.

3

Teoría de agentes conversacionales

3.1. Agente conversacional

En nuestro marco teórico, un agente conversacional es un agente capaz de entablar una conversación con otros agentes conversacionales. Dos palabras claves de esta definición son *agente* y *conversación*. Un agente es una entidad humana o artificial que percibe su entorno a través de unos sensores y ejecuta acciones que lo pueden modificar (Burgin y Dodig-Crnkovic, 2009), y una conversación es un intercambio de información entre dos o más agentes (*Conversation*, s.f.).

La información transmitida en la conversación es codificada y segmentada en unidades discretas denominadas mensajes. Este intercambio tiene lugar en el medio o entorno en el que se encuentran presentes los interlocutores, por lo que los agentes perciben un mensaje destinado a ellos como un estímulo procedente del entorno y actúan sobre este al enviar un mensaje al receptor.

3.1.1. Modelo de comunicación

Una primera restricción sobre los agentes conversacionales que desarrollaremos es el establecimiento de un límite al número de agentes que intervienen en una misma conversación. En nuestro *framework* las conversaciones tendrán lugar entre dos agentes conversacionales, por lo que estaremos ante un tipo específico de conversación denominado diálogo. Esta decisión de diseño ha llevado a la elección de un modelo de comunicación inspirado en el modelo de sistema de comunicación propuesto por Shannon y Weaver (1964, pp. 6-7), cuya representación esquemática se puede apreciar en la figura 6.

Sea $m \in \mathcal{M}$ un mensaje seleccionado de un espacio de mensajes \mathcal{M} . Decimos que en la intervención en la que un agente A envía el mensaje m a un agente B , denotada $A \xrightarrow[m]{}$ B , A adopta el rol de emisor y B el de receptor. El modelo de [Shannon y Weaver \(1964, pp. 6-7\)](#) nos permite representar esquemáticamente una intervención en la comunicación, por lo que se trata de un modelo unidireccional en el que los roles de emisor y receptor son estáticos.

Si bien un acto comunicativo es de naturaleza unidireccional, un diálogo entre dos entidades es bidireccional. Formalmente, un diálogo entre dos agentes A y B en un lenguaje \mathcal{M} es una secuencia de intervenciones $e_t \xrightarrow[m]{}$ r_t donde $e_t, r_t \in \{A, B\}$, $e_t \neq r_t$, y $m \in \mathcal{M}$. Aunque dos agentes pueden enviarse información de forma continua en el tiempo, los agentes que desarrollaremos interactuarán con usuarios humanos y por consiguiente, es más adecuado suponer que el tiempo es discreto.

3.1.2. Taxonomía de agentes conversacionales

Podemos dividir los agentes conversacionales en dos grupos: basados en tareas y de dominio abierto. Los que pertenecen al primer grupo tienen por objeto ayudar al otro agente en la compleción de tareas propias de uno o múltiples dominios ([Zhang, Takanobu, Huang, y Zhu, 2020, p. 1](#)). Por ejemplo, la reserva o cancelación de una reserva en un restaurante son tareas propias de un dominio específico y no general. Los agentes de dominio abierto, al contrario que los basados en tareas y como su nombre indica, no están limitados a un número finito de dominios y pretenden maximizar la participación del usuario ([Huang, Zhu, y Gao, 2019](#)).

Nuestro *framework* estará orientado al desarrollo de agentes conversacionales basados en tareas, aunque es posible extender la implementación de esta teoría para soportar cualquier tipo de agente conversacional. Una restricción adicional será la limitación del número de interlocutores a dos entidades. Dada la definición de agente basado en tareas, podemos suponer que en la mayoría de los casos un agente artificial estará asistiendo a un solo usuario en una misma conversación. Si bien es posible la integración de este tipo de agentes en grupos o foros constituidos por más de un usuario, la interacción con este tipo de *chatbots* suele tener lugar entre un solo usuario y el *bot*. En otras palabras, no suele ser necesario tener en cuenta de forma explícita la información transmitida en una conversación entre A y B en una conversación entre B y C .

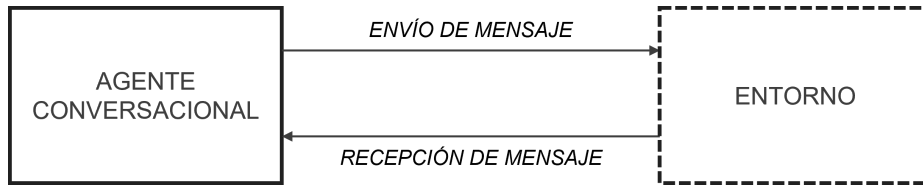


Figura 5: Representación esquemática de la interacción entre un agente conversacional y su entorno.

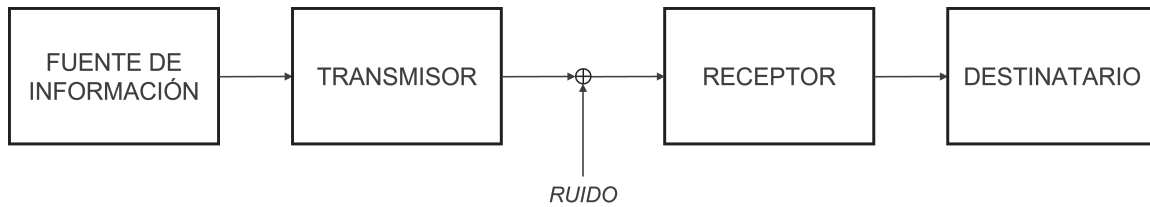


Figura 6: Representación esquemática del modelo de sistema de comunicación de Shannon y Weaver.



Figura 7: Representación esquemática de la aplicación del modelo de sistema de comunicación de Shannon y Weaver a una intervención en un diálogo entre dos agentes conversacionales.

3.1.3. Arquitectura de un agente conversacional

El comportamiento de un agente es el resultado de la interacción de varios procesos que procesan la información procedente de los sensores y determinan las acciones a ejecutar. Cada proceso está constituido a su vez de servicios que pueden llevar a cabo tareas cognitivas. Por ejemplo, un agente podría estar dotado de un proceso para analizar las personas que aparecen en una imagen. Este proceso podría estar constituido por un servicio para reconocer caras, uno que asocie identidades a las caras extraídas y otro servicio que determine si la persona parece feliz, triste, enfadada, etc.

En el caso que nos ocupa, el agente debe ser capaz al menos de comprender un mensaje, realizar una serie de acciones u operaciones internas y generar una respuesta. Distinguimos, pues, al menos tres capacidades cognitivas avanzadas: comprensión de lenguaje, toma de decisiones y generación de lenguaje. Cada una de estas capacidades debe estar asociada a un único proceso del agente. Los servicios que constituyen estos procesos pueden ser cognitivos o no. Un servicio cognitivo resuelve una subtarea intelectual más simple que la que resuelve el proceso del que forma parte. Por citar un ejemplo, la extracción de entidades textuales es un servicio cognitivo que forma parte de un proceso de comprensión de lenguaje natural.

Uno de los aspectos fundamentales de un proceso es el mecanismo u orquestador que determina cuándo se llama un servicio, qué entrada se le proporciona y qué tratamiento se da a su salida. Proponemos dos tipos de arquitectura para un proceso:

- **Secuencial:** Se establece un orden total sobre los servicios que constituye el proceso y se define una región de memoria común para todos ellos. La entrada con la que se ejecuta el proceso se introduce en esta memoria y en cada paso se invoca un servicio proporcionándole un segmento de la memoria compartida que contiene la información que necesita. El resultado generado por cada servicio se almacena en esta memoria, permitiendo que un servicio pueda consumir los resultados generados por otros en etapas anteriores.
- **Grafo dirigido acíclico:** La arquitectura anterior puede fijar restricciones demasiado fuertes sobre el orden de ejecución de los servicios. Una alternativa es una arquitectura en forma de grafo dirigido en el que los nodos son servicios y las aristas determinan el sentido de los distintos flujos de cómputo. Si dos o más nodos que forman parte de

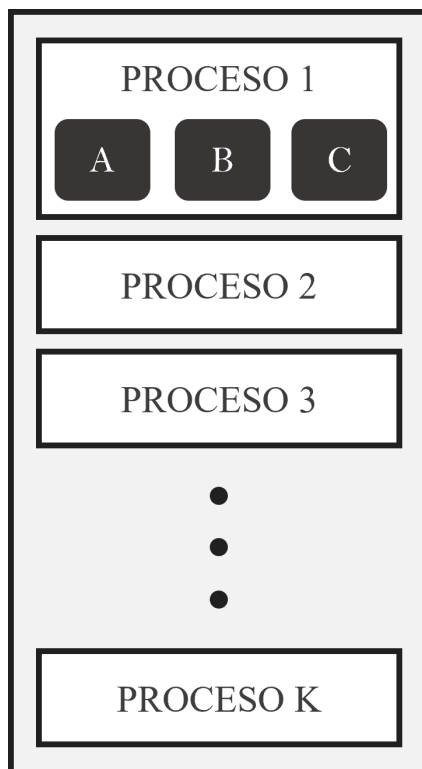


Figura 8: Un agente está constituido por k procesos. Cada proceso contiene una serie de servicios. En este ejemplo, el proceso 1 tiene tres servicios: A, B y C.

distintos flujos de cómputo apuntan a un mismo nodo s_i , se produce una convergencia de estos flujos en un mismo punto, y la ejecución de todos ellos deberá ser completada antes de ejecutar s_i .

Las dos arquitecturas anteriores hacen uso de una memoria compartida durante el entrenamiento y la ejecución del proceso. Los servicios consumen un subconjunto de la información presente en esta memoria y su salida es almacenada junto con un identificador en esa misma memoria. La interfaz de un servicio especifica los identificadores de los símbolos obligatorios u opcionales que deberían estar presentes en la memoria compartida en el momento de ejecutar el servicio. Asimismo, debe especificarse un identificador para cada símbolo generado por el servicio.

Uno de los problemas del enfoque expuesto es que puede ser necesario modificar la interfaz de un servicio para adaptarlo a nuevos cambios en el proceso del que forma parte. Por ejemplo, supongamos que un servicio R_1 genera una representación vectorial de un estímulo y la almacena en la memoria compartida con un identificador *repr*. Un servicio C_1 de clasificación puede definir en su interfaz que espera consumir un símbolo denominado *repr*. Si introducimos otro servicio R_2 que genere una representación alternativa a la de R_1 , debemos modificar las interfaces de R_1 , R_2 y C_1 . Los dos primeros no pueden almacenar sus salidas con el mismo identificador, porque se perdería información que podría ser necesaria más adelante. Si el identificador de R_1 y R_2 pasa a ser *repr1* y *repr2*, respectivamente, la definición de la interfaz de C_1 también ha de actualizarse para que consuma un símbolo *repr1*.

La solución a la problemática expuesta anteriormente es el aislamiento de los servicios. La interfaz de un servicio no debe depender del proceso del que forma parte, por lo que aislamos los servicios en contenedores denominados nodos computacionales. Un nodo computacional está constituido por un único servicio y una tabla de traducción dependiente del proceso que especifica una asignación de identificadores esperados en la memoria compartida e identificadores de la interfaz del servicio. Asimismo, se especifica un identificador para cada una de las salidas del servicio.

Al igual que un proceso está constituido por servicios, decimos que un agente conversacional está constituido por procesos. Desde un punto de vista computacional, podemos contemplar un proceso como un servicio más complejo y el agente sería un proceso en sí. Los mismos principios de arquitectura expuestos para los servicios en esta sección se aplican a los proce-

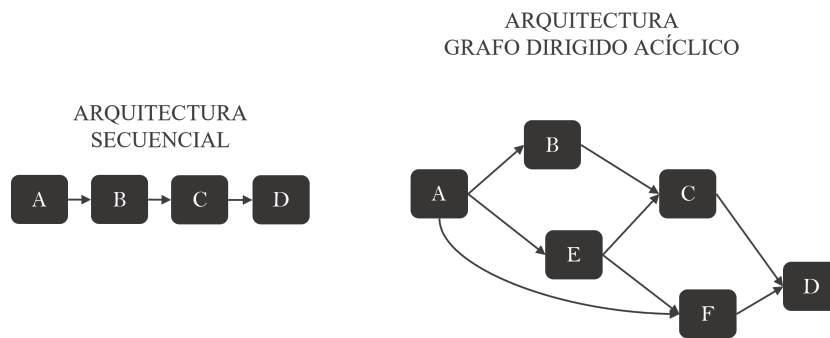


Figura 9: Dos topologías o arquitecturas posibles para establecer un orden parcial o total sobre la invocación de servicios.

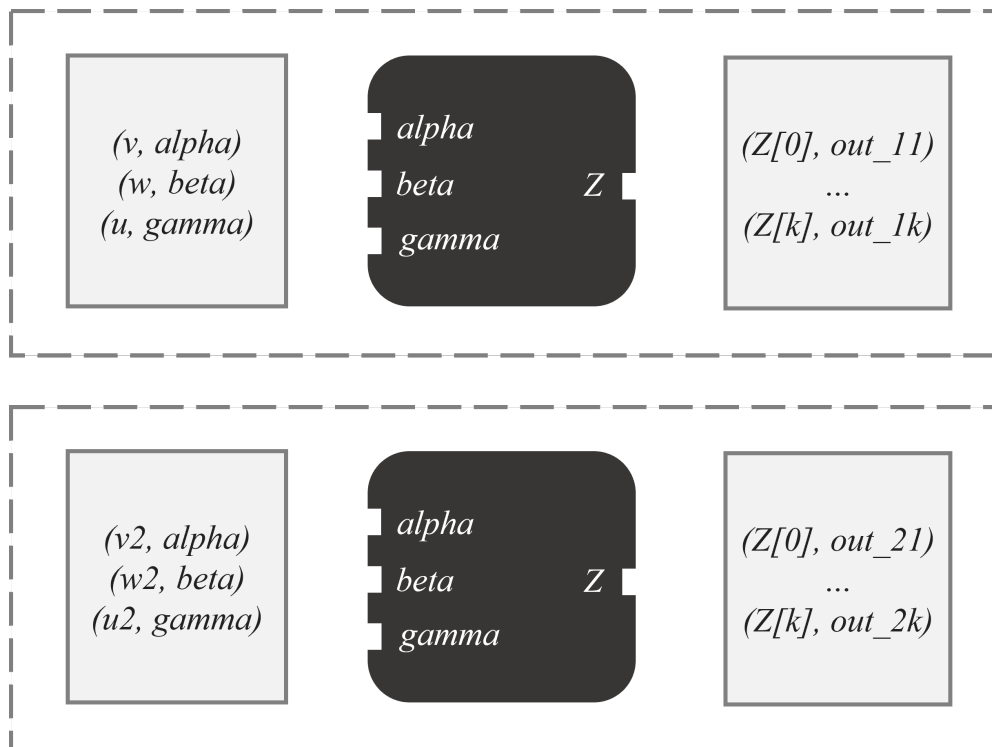


Figura 10: Dos instancias de un mismo servicio pueden convivir en un mismo proceso al encapsularse como nodos computacionales. Ambas instancias comparten una misma interfaz con tres entradas y una salida Z . Al inyectarse los valores de la memoria compartida en cada entrada, no se utiliza el nombre de los parámetros de la interfaz sino los identificadores asociados al parámetro que queremos inyectar. Por ejemplo, el campo v se inyecta como argumento asociado a 'alpha' en la primera instancia. Las salidas también tiene asociadas identificadores.

tos. La interacción y cooperación de todos ellos permite la generación del comportamiento racional y complejo esperado para este tipo de agentes.

3.2. Expresividad y restricciones sobre los canales de comunicación

La naturaleza del canal o medio por el cual tiene lugar la transmisión de mensajes puede establecer una serie de restricciones sobre los símbolos o señales disponibles para transmitir la información. Por ejemplo, si un *chatbot* y un usuario entablan una conversación en un chat, los símbolos empleados para codificar la información se pueden limitar a texto y emojis. Si un usuario se comunica con un asistente por comandos de voz, el tono, ritmo o intensidad de la voz puede ser procesados como información adicional al mensaje verbal.

La riqueza de un medio de comunicación se define como la capacidad que tiene la información transmitida por dicho medio de actualizar la comprensión del receptor en un intervalo de tiempo determinado (Daft y Lengel, 1986, p. 560). En otras palabras, un medio o canal será rico o expresivo si los tipos de símbolos que pueden emitirse permiten evitar o resolver ambigüedades rápidamente. Por ejemplo, la información no verbal disponible en una comunicación cara a cara puede proporcionar la información adicional necesaria para completar o aclarar el mensaje verbal. En el caso de una comunicación estrictamente textual, el usuario o *chatbot* podría tener que enviar varios mensajes para confirmar o aclarar el significado de un mensaje anterior. A menor capacidad expresiva, mayor será el número de intervenciones necesarias para resolver ambigüedades y, por ende, será necesario un mayor periodo de tiempo para pasar de un estado de confusión a comprensión.

Nuestros agentes conversacionales podrían recibir información adicional al contenido textual del mensaje, pero en este trabajo de fin de grado nos centraremos en el procesamiento del mensaje verbal. A la hora de evaluar el rendimiento de un agente conversacional artificial en un medio de baja capacidad expresiva, debemos tener en cuenta que tanto para un agente humano como uno artificial el proceso de comprensión dispondrá de menos información que la disponible a través de otros canales y, por lo tanto la incertidumbre será mayor.

Además de las propiedades inherentes de un canal de comunicación, podemos poner a disposición del agente canales con ciertas restricciones impuestas por los requisitos del proyecto. Por ejemplo, un asistente por voz puede utilizar un canal textual privado para notificar información confidencial y uno de voz para transmitir información pública y no perteneciente a la

esfera íntima del usuario.

3.3. Clasificación PENS de los lenguajes naturales

La transmisión de información entre dos entidades no sería posible sin la existencia de un lenguaje común que permitiera expresar mensajes siguiendo un conjunto de reglas y haciendo uso de un mismo vocabulario. Además de los lenguajes naturales, que son fruto de una evolución natural de la comunicación entre los seres humanos, existen lenguajes artificiales y lenguajes naturales controlados. Estos últimos tienen como base un lenguaje natural al que se le aplican restricciones para simplificar su gramática o vocabulario con el objeto de reducir la complejidad o ambigüedad intrínseca del lenguaje utilizado como base (Kuhn, 2014, pp. 123-124).

En esta sección, expondremos algunas de las propiedades de los lenguajes naturales, y no pondremos énfasis en los lenguajes artificiales como los lenguajes de programación. Las características que mencionaremos no pertenecen al nivel superficial de los lenguajes, sino a un nivel abstracto, de tal forma que, en lugar de exponer las propiedades de lenguajes específicos como el chino o el inglés, mencionaremos exclusivamente aquellas propiedades universales comunes a todos.

3.3.1. Propiedades fundamentales

El esquema de clasificación de propiedades fundamentales de los lenguajes introducido por Kuhn (2014, pp. 126-132) consta de solo cuatro factores o dimensiones denominadas precisión, expresividad, naturalidad y simplicidad. Cada uno de estos factores puede tener un valor entero entre 1 y 5, ambos incluidos. Aquellos lenguajes que tengan una misma puntuación para la precisión, expresividad, naturalidad y simplicidad, pertenecerán a una misma clase de lenguajes. Desafortunadamente, no existe ningún método formal para asignar un valor a cada una de estas variables, aunque se toma como referencia la clase a la que pertenece un lenguaje natural como el inglés y la clase del lenguaje de la lógica proposicional. A continuación, definimos cada uno de los factores del esquema PENS:

- **Precisión:** Cuantifica el grado de recuperación de la información original que quería transmitir el emisor en su mensaje (Kuhn, 2014, p. 128). A mayor precisión, menor será

la desviación del significado inicial del mensaje, esto es, el lenguaje será más resistente al conocido como ruido semántico (Shannon y Weaver, 1964, p. 26). Mientras que el lenguaje de la lógica proposicional tiene una precisión máxima P^5 y carece de ambigüedad, un lenguaje natural suele tener una precisión mínima o prácticamente mínima.

- **Expresividad:** A mayor expresividad, mayor es el espacio de mensajes transmitibles. El lenguaje de la lógica de primer orden es más expresivo que la lógica proposicional al introducir cuantificadores, pero es menos expresivo que cualquiera de los lenguajes naturales (Kuhn, 2014, pp. 129-127). Kuhn propone el siguiente criterio para asignar un valor de expresividad a un lenguaje determinado.
 - *Nivel 1.* Ni soporta cuantificación universal sobre individuos ni relaciones no monarias.
 - *Nivel 2.* Soporta cuantificación universal sobre individuos y relaciones n -arias.
 - *Nivel 3.* Es capaz de expresar reglas e introducir la negación.
 - *Nivel 4.* Introduce la cuantificación universal de segundo orden.
 - *Nivel 5.* Los lenguajes con este grado de expresividad pueden expresar cualquier enunciado de cualquier tipo de lógica.
- **Naturalidad:** Se define como el grado de legibilidad y comprensibilidad del lenguaje para un hablante de una lengua natural. Los lenguajes con un nivel de naturalidad bajo hacen un uso de ciertos símbolos, estructuras sintácticas o semánticas que no son frecuentes en los lenguajes naturales y requieran de formación o entrenamiento para poder ser comprendidas por parte de un ser humano (Kuhn, 2014, pp. 130-131).
- **Simplicidad:** Cuantifica la complejidad de la descripción exacta y completa de la sintaxis y gramática de un lenguaje. A menor simplicidad, mayor será la dificultad para implementar un modelo matemático del mismo (Kuhn, 2014, pp. 131-132).

3.4. Proceso de comprensión del lenguaje

En un agente conversacional la información que es extraída por el proceso de comprensión debe ser almacenada en algún tipo de estructura expresada en un lenguaje dado. Extendemos y

supondremos cierta la hipótesis del lenguaje del pensamiento (LOTH) atribuida principalmente a Fodor (Rescorla, 2019). De acuerdo con esta hipótesis, existe un lenguaje mental análogo al lenguaje hablado para expresar el pensamiento, lo cual incluye creencias, ideas e intenciones.

Hipótesis 3.1 (Lenguaje del pensamiento de un agente conversacional). *Cualquier agente conversacional, artificial o no, capaz de almacenar, analizar, transformar o generar información requiere de un lenguaje para expresar con estructuras mentales la información.*

En el capítulo 4 introduciremos técnicas para poder extraer la intención, el sentimiento y las entidades textuales presentes en mensajes en un cierto lenguaje natural, y obtener así unas estructuras mentales que almacenarán aquella información que sea de interés, lo cual nos lleva la siguiente suposición que haremos.

Hipótesis 3.2 (Independencia del lenguaje mental y natural). *El lenguaje mental en el que se expresen las estructuras mentales es independiente del lenguaje natural utilizado para la comunicación entre dos agentes conversacionales.*

Al suponer cierta la hipótesis anterior, podemos desarrollar una agente conversacional capaz de entender varios idiomas, sin modificar el *core* del agente, que sería responsable de almacenar, analizar o generar información en un formato estructurado, independientemente del lenguaje natural utilizado para su transmisión.

La comprensión es, por ende, un proceso de traducción de un lenguaje natural a un lenguaje mental, en la que se podría producir pérdida de información. El problema de la comprensión de lenguaje natural o NLU, que será introducido al principio del capítulo ??, se reduce a cómo ha de hacerse esta traducción.

3.4.1. Las representaciones mentales de las entidades textuales

Asumiendo como cierta la hipótesis de independencia entre un lenguaje natural empleado para la comunicación y el lenguaje mental de representación de la información, debe existir, asimismo, una independencia entre los términos con los que nos referimos a una entidad y la entidad en sí.

Hipótesis 3.3 (Independencia entre las representaciones de las entidades). *El significante, esto es, la representación de una entidad en un lenguaje natural es independiente del significado al que está asociado, el cual es la representación de esa misma entidad en un lenguaje mental.*

Desde un punto de vista de la implementación de agentes artificiales, suponer cierta la hipótesis anterior nos permite la creación de una lógica independiente de la representación de las entidades que intervendrán en los flujos de los agentes. Por ejemplo, en lugar de almacenar el significante 'Nueva York' en una estructura mental, almacenamos un identificador, el significado, que estará asociado a 'Nueva York' en el idioma español y a 'New York' en el idioma inglés.

Hipótesis 3.4 (Unicidad de la representación mental de una entidad). *Toda entidad tendrá una y solo una representación posible en el lenguaje mental, independientemente de si esta representación está asociada a más de una representación en el lenguaje natural. En otras palabras, el concepto de sinonimia no es necesario en el lenguaje mental.*

La hipótesis de unicidad anterior establece un requisito que simplificará la lógica de los asistentes introduciendo un mecanismo de normalización en el proceso de comprensión, el cual transformará todo significante en el representante canónico de la clase de significantes asociados a un significado.

3.4.2. Estimación de la intencionalidad

Asumiendo como cierta la hipótesis de que todo acto comunicativo es ejecutado por un agente con una intención, exponemos un modelo basado en el propuesto por [Malle y Knobe \(1997\)](#) para definir el concepto de intencionalidad. Las intenciones de un agente serán expresadas en términos de otras estructuras mentales: las creencias y deseos u objetivos. Un agente conversacional tiene una lista de objetivos que pueden variar con el paso del tiempo. Desde un punto de vista de la planificación, el agente construye un plan de acciones sustentando por unas creencias que le llevan a concluir que su ejecución tendría como consecuencia la satisfacción de uno o más objetivos.

La intención de una acción $x \in A$ se define como un símbolo del lenguaje mental que representa el compromiso de ejecutar x para cumplir unos objetivos $O \subset \Omega$, suponiendo que el sistema de creencias ha realizado una inferencia correcta. Las reglas del sistema de inferencia pueden ser representadas o no con representaciones mentales. En el segundo caso, las reglas forman parte del sistema que gobierna la aplicación de operaciones sobre las representaciones. Dado que en el proceso de comprensión se pretende reconstruir estas representaciones

en el agente receptor, parte de las reglas que determinaron la elección de un plan de acción podrían no ser recuperables. Un agente conversacional puede actuar en respuesta al mensaje de un emisor estimando la posible intención que pudo tener este al generar el mensaje. Por lo anterior, podemos considerar que la intención estimada es una representación mental que nos abstrae de las creencias y objetivos exactos del agente emisor, pero que es suficiente para determinar una respuesta adecuada.

Una vez expuesto el concepto de intención, se debe proporcionar evidencias de que es posible construir un agente conversacional capaz de estimar la intención de los mensajes de otro agente con el que se comunica. Como se mencionó anteriormente, la comprensión requiere de un conocimiento compartido por ambos agentes. Un agente humano podrá entender a un agente artificial que tenga capacidades limitadas de generación de lenguaje natural proporcionadas por algoritmos codificados por un ser humano. En el caso de los seres humanos, existe la hipótesis de que la comprensión del lenguaje es posible gracias a un sistema de neuronas espejo (Fogassi y Ferrari, 2007, pp. 137-139), que se activan en respuesta a la ejecución de acciones, independientemente de si la ejecutan otros o nosotros mismos (Di Pellegrino, Fadiga, Fogassi, Gallese, y Rizzolatti, 1992). En el caso de un agente conversacional artificial, el sistema espejo está entrenado para estimar la intención de los mensajes de los seres humanos con una capacidad expresiva, estilo, etc. similares a los presentes en el conjunto de entrenamiento. Por lo anterior, la capacidad de estimar la intención de los agentes conversacionales estará acotada a la cantidad y calidad de datos de entrenamiento disponibles.

3.4.3. Complejidad del proceso de comprensión

El esquema de clasificación PENS nos permite analizar cualitativamente la complejidad del proceso de comprensión de un agente capaz de comprender los mensajes expresados en un determinado lenguaje. Para llevar a cabo este análisis debemos introducir cuatro métricas o propiedades básicas que evalúan o caracterizan un proceso de comprensión:

- El porcentaje de información original recuperada.
- El tiempo de ejecución del proceso de comprensión.
- El grado de interpretabilidad de los modelos utilizados por el proceso.

- La cantidad de información a la que debe tener acceso el proceso para comprender mensajes en el lenguaje natural objetivo.

El porcentaje de información original recuperada por el proceso de comprensión está íntimamente relacionado con la precisión del lenguaje natural objetivo. A mayor precisión, menor será la influencia del ruido semántico y por lo tanto, mejor será la capacidad del agente para extraer información.

En cuanto al tiempo de ejecución, debemos decir que depende de dos factores: el tipo de agente y la naturalidad del lenguaje objetivo. Un agente humano requerirá más tiempo para procesar un mensaje que contenga unas estructuras lingüísticas que se aproximen a las propias de un lenguaje máquina o de programación. Por el contrario, una máquina requerirá más etapas de procesamiento para extraer información de un texto en lenguaje natural, ya que la información está expuesta en un formato no estructurado.

Las dos propiedades básicas restantes de los procesos de comprensión ponen énfasis en la implementación de los servicios que constituyen el proceso. Si estos servicios hacen uso de modelos basados en reglas que no requieren aprendizaje, el grado de interpretabilidad de la implementación será mayor que si recurrimos a modelos estadísticos. Mientras que los modelos basados en reglas codifican un conocimiento innato de hechos y reglas, los modelos estadísticos aprenden el conocimiento necesario para desempeñar una tarea. Por lo anterior, la cantidad de información a la que tienen que estar expuestos agentes artificiales que utilicen algoritmos o modelos de aprendizaje automático es superior a la que necesitan los agentes basados en reglas. La decisión de optar por una clase de modelos u otra dependerá del grado de expresividad y naturalidad. A mayor naturalidad y expresividad, mayor será la complejidad de la descripción del lenguaje. Un lenguaje simple puede ser modelado fácilmente con un modelo determinista, pero uno complejo hará necesario el uso de modelos que aprendan las características más relevantes del lenguaje mediante su exposición a mensajes expresados en ese lenguaje.

3.5. Proceso *core*

Tras obtener una representación estructurada de un mensaje entrante, el siguiente es paso es tomar una decisión sobre qué acciones ejecutar en respuesta. La responsabilidad de esta

tarea recae sobre un proceso denominado *core* que opera principalmente con estructuras mentales.

Dado que nuestros agentes son conversacionales, contemplamos al menos dos acciones: emisión de un mensaje y espera de una respuesta. Algunas de las acciones no relacionadas directamente con la comunicación pueden ser la ejecución de servicios de negocio para almacenar información en una base de datos, hacer peticiones a una API, etc. Estos servicios también forman parte del proceso *core* del agente.

Los modelos de comportamiento de nuestros agentes están basados en las teorías S-R (*Stimulus-Response*), las cuales suponen que toda respuesta se debe a un estímulo, y que el comportamiento de un agente se puede describir como una secuencia de estímulos y reacciones (Deutsch, 1966, p. 413). En nuestro marco teórico, estos modelos están constituidos por dos elementos:

- Un mecanismo que mantenga una representación actualizada del contexto en el que se encuentra el agente, introduciendo nueva información relevante y eliminando la que ya no es significativa.
- Un algoritmo que establezca una asociación entre clases de contextos similares y acciones. Si dado un contexto selecciona la siguiente acción a ejecutar, estaremos ante un modelo de paso pequeño. Si en lugar de la acción siguiente selecciona una lista de todas las acciones a ejecutar, el modelo será de paso grande.

El contexto está constituido por los nuevos estímulos que acaban de ser procesados y la información que conservamos de los estímulos recibidos en un pasado reciente. En el caso de un agente humano, las capacidades de incorporar y eliminar información son posibles gracias a la atención selectiva y la inhibición de información como resultado de un proceso competitivo entre recuerdos (Long, Kuhl, y Chun, 2018). Estas capacidades cognitivas han servido de inspiración para crear modelos estadísticos con operaciones de olvido y almacenamiento como LSTM (Zeng, 2019).

En cuanto a la asociación entre representaciones simplificadas de un contexto y las acciones, podemos distinguir entre dos métodos para codificar el conocimiento necesario para seleccionar la reacción. El primero de ellos consiste en la existencia de reglas que determinen con total certeza las acciones que deben ser ejecutadas en un contexto. El segundo método

consiste en exponer al agente a un proceso de aprendizaje que culminará con la adquisición de un conjunto de reglas que rijan su comportamiento. Aunque ambos enfoques son opuestos, no son incompatibles y de hecho, se complementan. El primer método introduce determinismo y garantiza que el agente se comportará de forma adecuada en situaciones críticas o importantes; el segundo puede generalizar el comportamiento adecuado para ciertos contextos poco comunes.

3.6. Proceso de generación del lenguaje

La transmisión de información requiere una traducción en el sentido inverso a la comprensión. Las estructuras mentales correspondientes deben ser codificadas en forma de palabras, frases hechas, enunciados, etc. La intención de transmitir una colección de datos es otra estructura mental, que está constituida por un subconjunto de las estructuras mentales presentes en el agente conversacional, así como de hiperparámetros del proceso de traducción como el grado de complejidad sintáctica o semántica, la longitud, el grado de formalidad, etc.

La complejidad de una instancia dada del problema de la generación del lenguaje natural radica en el grado de creatividad y flexibilidad necesarias en el proceso de generación. Aspectos como el uso de recursos retóricos y sarcasmo pueden hacer el problema NLG más complicado, aunque, en general, se puede asumir que el problema NLU reviste una mayor dificultad.

La generación de texto en un determinado lenguaje requiere conocer su gramática y vocabulario. Si restringimos la gramática a un conjunto de plantillas con una sintaxis correcta, reducimos la expresividad del agente, pero simplifica la generación de texto. Por otro lado, debemos disponer de una función que asocie identificadores de entidades con su correspondiente representación textual en el lenguaje objetivo. En ciertas ocasiones, puede ser necesario que esta asociación tenga en cuenta información sobre rasgos gramaticales como el género, número o caso.

Si tratamos de resolver el problema NLG a nivel de carácter, es posible que el resultado de la generación nos sea gramaticalmente correcto, o que el enunciado no sea coherente. Al recurrir a un enfoque basado en plantillas, podemos disponer de varias plantillas para una misma intención que no solo introduzcan variabilidad en las respuestas, sino que ofrezca alternativas más simples y que omitan información secundaria.

4

Diseño del proceso de comprensión

El proceso de comunicación se puede contemplar como un intercambio de conocimiento, sentimientos y pensamientos entre dos o más individuos. Los agentes que forman parte de una conversación intercambian entidades externas, sus mensajes, que permiten la reconstrucción parcial de las entidades internas de los agentes. Estas entidades son las estructuras mentales que introducimos en el segundo capítulo (Brady y Berwick, 1989, p. 27).

Mientras que el rol del emisor es construir un mensaje que transmita una intención comunicativa, el receptor debe ser capaz de reconstruir la información original. La capacidad de reconstruir las estructuras internas exige una comprensión que nace de la compartición de un mismo conjunto de reglas que rigen la validez o no de los enunciados que podemos generar (Brady y Berwick, 1989, pp. 27-28). Dado que un agente es consciente de que necesita dotar su mensaje de ciertas propiedades sintácticas, semánticas y pragmáticas para transmitir unas determinadas estructuras mentales, es capaz de invertir el proceso y dado un mensaje, deducir aproximadamente qué quería transmitir su emisor.

El problema general de la comprensión de lenguaje natural o NLU (*Natural Language Understanding*) consiste en la obtención de una representación estructurada adecuada del contenido lingüístico de un mensaje. A través de un modelo o componente NLU adaptado a sus necesidades, un agente conversacional artificial resuelve parcialmente este problema de comprensión (Clark, Fox, y Lappin, 2013, p. 43).

4.1. Preprocesamiento de lenguaje natural

La calidad de los modelos estadísticos que entrenamos está íntimamente ligada a la calidad de los datos que utilizamos para su entrenamiento. Este hecho convierte el preprocesamiento

en un aspecto fundamental del proceso de minería de datos (Mushtaq, 2019). El preprocesamiento, tal y como su nombre indica, consiste en aplicar un procesamiento a unos datos para obtener una representación de los mismos más óptima para su consumo por parte de un proceso de extracción de conocimiento (Acuña, 2011).

4.2. Segmentación

La tokenización o segmentación consiste en segmentar una secuencia de caracteres en unidades denominadas *tokens* si la segmentación es a nivel de palabra o enunciados si es a nivel de enunciado (Manning, Raghavan, y Schütze, 2008, p. 22).

En el caso que nos ocupa haremos énfasis en el tratamiento que se da a esta tarea en lenguajes segmentados, esto es, aquellos en los que las palabras están delimitadas por espacios.

4.2.1. A nivel de enunciado

En la mayoría de lenguajes, se utilizan símbolos de puntuación para delimitar enunciados, por lo que esta segmentación se reduce al problema de determinar los símbolos de puntuación que están delimitando un enunciado (D. Palmer, 2010, p. 22).

La no trivialidad de este problema radica en que un símbolo de puntuación no siempre determina el final de un enunciado (D. D. Palmer y Hearst, 1997). Para resolver este problema se suele recurrir a una lista de excepciones de la regla general de segmentación (D. Palmer, 2010, pp. 43-44).

Ejemplo 4.1

Una simple regla general de segmentación para el castellano puede ser la siguiente

El símbolo '.' denota el final de un enunciado.

Dos excepciones de la regla anterior son las abreviaturas 'Sr.' y 'Sra.' de los tratamientos 'Señor' y 'Señora', respectivamente.

4.2.2. Problemas de la segmentación a nivel de palabra

Al aplicar este tipo de segmentación nos podemos encontrar con una serie de palabras que requieren un tratamiento especial:

- **Contracciones:** Son el resultado de combinar varias palabras mediante la omisión de letras ([Merriam-Webster, s.f.](#)). O bien se trata la contracción como un solo *token* o se puede generar tantos *tokens* como palabras la constituyan ([D. Palmer, 2010, p. 17](#)).
- **Expresiones compuestas por más de una palabra separada por espacio:** La [RAE \(s.f.\)](#) define locución como "grupo de palabras que funcionan como una sola pieza léxica con un sentido unitario y cierto grado de fijación formal". Algunos ejemplos de locución son 'a pesar de' o 'in situ'. [D. Palmer \(2010, pp. 18-19\)](#) propone que los algoritmos de segmentación tengan acceso a una lista de grupos de palabras que deban tratarse como un sola unidad.
- **Palabras compuestas:** En ciertos lenguajes como el alemán, es habitual crear palabras compuestas sin establecer un delimitador entre las dos o más palabras que se componen ([D. Palmer, 2010, p. 18](#)). En otros lenguajes, es común construir una palabra compuesta haciendo uso del guión.
- **Texto entrecomillado:** Debemos configurar el algoritmo de segmentación para tratar o no las citas textuales y, en general, fragmentos de texto delimitados por comillas simples o dobles como una sola unidad, esto es, un solo *token*.

La plataforma de procesamiento de lenguaje natural SpaCy recurre a un sistema de reglas para aplicar la segmentación a nivel de palabra. En primer lugar se aplica una segmentación considerando el espacio como delimitador. Las reglas tratan casos especiales permitiendo unificar *tokens* contiguos o separar un *token* en varios. El algoritmo es iterativo, esto es, se aplica una regla de segmentación o unificación en cada etapa y el proceso termina cuando no se puede aplicar ninguna más ([Linguistic Features · spaCy Usage Documentation, s.f.](#)).

4.3. Normalización

Normalizar un *token* consiste en aplicarle una serie de transformaciones que deriven en su forma canónica ([Manning y cols., 2008, p. 28](#)).

4.3.1. Mayúsculas y minúsculas

El rendimiento de los modelos utilizados para la comprensión de lenguaje natural puede verse afectado o no por la presencia de letras en mayúsculas y minúsculas. En la tarea de extracción de entidades, por ejemplo, es posible aplicar técnicas como *truecasing* para reducir el impacto de mensajes que no respeten las reglas de capitalización (Lita, Ittycheriah, Roukos, y Kambhatla, 2003). No obstante, en otras tareas como la clasificación de intenciones puede ser útil expresar todo el mensaje en minúsculas.

4.3.2. Signos diacríticos

Se pueden suprimir los acentos diacríticos, puesto que los usuarios suelen evitarlos al escribir mensajes (Manning y cols., 2008, p. 29). Asimismo, Schwab (2015) concluye que, en el caso del castellano, el reconocimiento visual de palabras a las que se le había eliminado previamente el acento era posible en aquellos casos en los que no existiera ambigüedad. Por ejemplo, la cadena 'carcel' puede identificarse como 'cárcel' en cualquier contexto.

4.3.3. Emoticonos y emojis

En aquellas tareas en las que sea beneficioso preservar la información transmitida por la presencia de emojis y emoticonos podemos llevar a cabo una normalización de estos símbolos a representaciones textuales o palabras emocionales que transmitan una carga sentimental similar en polaridad e intensidad.

4.3.4. Stemming

Desde un punto de vista lingüístico, las palabras están constituidas por unidades de significado irreducibles denominadas morfemas. Las palabras morfológicamente complejas están constituidas por un morfema raíz y uno o más afijos dando lugar a lo que se denomina un tema o *stem* (Fromkin y cols., 2018, pp. 41-46).

Ejemplo 4.2

Las palabras 'deporte', 'deportista' y 'deportistas' son tres *stems* que comparten una misma raíz 'deport'.

Los algoritmos de *stemming* tienen por objeto reducir una palabra a su raíz, de tal forma que las distintas variantes morfológicas de una palabra se normalicen a una misma forma canónica. Algunos de los *stemmers* más conocidos como el algoritmo de Porter tratan de resolver parcialmente el problema de obtener una forma próxima a la raíz eliminando un sufijo de la palabra (Hull y Grefenstette, 1996).

El algoritmo de Porter fue desarrollado para llevar a cabo un proceso de *stemming* a textos en inglés (Willett, 2006, p. 4), por lo que más adelante el mismo Porter llegó a desarrollar el lenguaje Snowball para definir rigurosamente algoritmos de *stemming* que podrían ser utilizados para otros idiomas (*A language for stemming algorithms*, s.f.).

4.3.5. Lematización

La lematización es un proceso similar a *stemming* con la diferencia de que la palabra se reduce a su lema, esto es, la base canónica aceptada por convenio (Manning y cols., 2008, p. 33). Un ejemplo de normalización por lematización sería transformar las formas verbales al infinitivo del verbo correspondiente.

4.4. Filtrado

Aquellos símbolos o palabras que no contribuyan significativamente al significado de un texto deberían ser filtrados. Algunos ejemplos de estos símbolos son los caracteres de control o varios espacios en blanco consecutivos. En función del dominio de aplicación, ciertos símbolos como los emojis podrían ser necesarios (Ayvaz y Shiha, 2017).

Si bien la repetición de un mismo símbolo de puntuación podría simplificarse, habría que analizar si la supresión completa de este tipo de símbolos pudiera tener un impacto negativo.

4.4.1. Palabras vacías (Stop words)

Las palabras vacías (*stop words*) son aquellas cuya aportación semántica al significado o tópico del mensaje del que forman parte es mínima o nula (Savoy y Gaussier, 2010, p. 458). Se atribuye el concepto a Hans Peter Luhn (Ladani y Desai, 2020).

Las palabras funcionales como las preposiciones, conjunciones o artículos, en contraposición a las palabras de contenido, suelen presentar una frecuencia elevada en textos y un

contenido semántico prácticamente inexistente (Haspelmath, 2001, p. 16539). Por esta razón, las palabras funcionales suelen ser consideradas uno de los dos tipos de palabras vacías: las palabras vacías genéricas (Ladani y Desai, 2020).

Con respecto a las palabras de contenido como 'coche' o 'pedido', hemos de decir que en función del dominio, ciertas palabras que transmiten un significado concreto son consideradas *stopwords* por no ser de interés para el dominio en cuestión (Ladani y Desai, 2020). Por ejemplo, la palabra 'monitor' probablemente no sea relevante en un texto asociado al dominio de la medicina.

Una de las primeras técnicas utilizadas para eliminar las palabras vacías de un texto fue la construcción de listas de *stopwords* por lenguaje (Ladani y Desai, 2020; Manning y cols., 2008, p. 27). Aunque esta lista puede construirse de forma manual, existen métodos para poder generarla de forma automática. Lo, He, y Ounis (2005) proponen un método para automatizar la construcción de esta lista que cuantifica la importancia de una palabra a partir de una métrica basada en la divergencia de Kullback-Leibler.

4.5. Etiquetado gramatical

El etiquetado gramatical o *part-of-speech tagging* consiste en asignar etiquetas de categorías gramaticales o léxicas a cada palabra de un texto (Jurafsky y Martin, 2020, p. 151).

Se parte de la suposición de que palabras que pertenecen a una misma categoría comparten características gramaticales que son exclusivas de dicha categoría (Kroeger, 2015, p. 34). Por ejemplo, una propiedad de los sustantivos es que pueden ser modificados por adjetivos.

Un POS *tagger* es un algoritmo que procesa una secuencia de palabras y elige una etiqueta gramatical para cada una de ellas (Bird, Klein, y Loper, 2009, p. 179). La complejidad del problema a resolver radica en la posibilidad de que una palabra pueda pertenecer a distintas categorías en función de su posición en el enunciado. Por ejemplo, encontramos algunas palabras en inglés como *can* o *book* que podrían ser verbos o sustantivos según el contexto (Jurafsky y Martin, 2020, p. 151).

Estos *taggers* pueden implementarse mediante un sistema de reglas de etiquetado con sus respectivas excepciones (Güngör, 2010, p. 209). Aunque se pueden construir manualmente, existen métodos como el aprendizaje basado en transformaciones (TBL) para generarlas de forma automática (Güngör, 2010, p. 209). También es posible recurrir a los modelos ocultos de

Markov para obtener la secuencia de etiquetas POS (Güngör, 2010, pp. 214-215).

Ejemplo 4.3

Un ejemplo de regla de etiquetado podría ser la siguiente:

Si la palabra anterior era un determinante artículo, la palabra actual es un sustantivo

4.6. Codificación

Ciertas tareas de comprensión del lenguaje natural son llevadas a cabo por modelos estadísticos que no consumen secuencias de caracteres, sino vectores de componentes reales o estructuras numéricas similares. El uso de estos modelos exige la elección de uno o varios esquemas de codificación que transformarán los enunciados en una representación adecuada para los algoritmos de aprendizaje automático.

Naseem, Razzak, Khan, y Prasad (2020) dividen los esquemas de codificación en dos grandes grupos: los que están basados en algoritmos de aprendizaje de representaciones y las propuestas clásicas o anteriores a estos esquemas. Los modelos clásicos se dividen a su vez en dos grupos:

- **Representación categórica:** Estos esquemas codifican los enunciados a partir de la información sobre la presencia o número de ocurrencias de cada palabra del vocabulario en el texto (Naseem y cols., 2020, p. 10). El modelo *bag-of-words* se considera un esquema de este tipo.
- **Representación ponderada:** La información utilizada para codificar los enunciados es la frecuencia relativa de las palabras en el texto y, en algunos casos, la frecuencia relativa de las mismas en un *corpus* (Naseem y cols., 2020, p. 11). Los modelos de referencia son *Term-frequency* y *Term Frequency-Inverse Document Frequency* (Jurafsky y Martin, 2020, pp. 107 - 108; Jones, 1972).

Una alternativa a los modelos clásicos son los modelos de representación distribuida. Estos modelos pertenecen a la clase de esquemas que recurren a algoritmos de aprendizaje de representaciones, por lo que son capaces de capturar propiedades semánticas que son ignoradas en los esquemas de representación categórica y ponderada, y proporcionan representaciones más eficientes en espacio (Naseem y cols., 2020, pp. 12-13).

En la representación distribuida, las palabras se representan con vectores de longitud fija denominados vectores distribuidos o *embeddings* que capturan el significado de las palabras por su distribución en el uso de las mismas en el lenguaje, esto es, por los contextos en los que suelen encontrarse (Jurafsky y Martin, 2020, p. 100). Por lo anterior, decimos que estos modelos siguen la hipótesis distribucional que asume que existe una correlación entre la similitud en significado y similitud en distribución en el lenguaje (Sahlgren, 2008, p. 34). Naseem y cols. (2020, pp. 12-16) distingue entre dos clases de representación distribuida: los *embeddings* contextuales y no contextuales.

- **Representación no contextual:** El vector distribuido de una palabra es global, único e independiente del contexto en el que se encuentra (Liu, Kusner, y Blunsom, 2020). Algunos de los modelos y arquitecturas de representación distribuida no contextual a destacar son Word2vec, GloVe y FastText (Naseem y cols., 2020, pp. 13-16)
- **Representación contextual:** La representación de una palabra es una función de la secuencia de palabras de la que forma parte (Liu y cols., 2020), permitiendo así una representación dependiente del contexto que tiene en cuenta fenómenos lingüísticos como la polisemia y homonimia (Nair, Srinivasan, y Meylan, 2020). Los modelos Context2Vec, CoVe, ELMo, BERT y sus variantes pertenecen a esta categoría (Liu y cols., 2020; Naseem y cols., 2020, pp. 16-18).

4.6.1. Bag-of-words

El modelo *bag-of-words* (BoW) descarta la información sobre el orden de las palabras, pero preserva el número de ocurrencias de cada una de ellas (Jurafsky y Martin, 2020, p. 57). En el caso binario, el número de ocurrencias se reduce a la presencia o ausencia de la palabra.

Es posible representar un *bag-of-words* como un vector de longitud fija definiendo un orden total sobre el vocabulario del modelo y aplicando una codificación *One-Hot*. Si el vocabulario es de k palabras, los vectores que representan las palabras son de k componentes. Dado que los vectores que codificarían las palabras serían ortonormales, habríamos transformado el vocabulario en un espacio vectorial con una base ortonormal y canónica de \mathbb{R}^k (Lu, 2020, p. 4).

El vector que codifica un mensaje utilizando la representación vectorial de *bag-of-words* se puede construir como una combinación lineal de los vectores de la base ortonormal inducida

por la codificación *One-Hot*. El coeficiente iésimo α_i sería la multiplicidad asociada a la palabra iésima, esto es, su número de ocurrencias en el mensaje.

Uno de los problemas principales de este esquema de codificación es que el número de dimensiones crece linealmente con el número de palabras del vocabulario, dando lugar a representaciones vectoriales dispersas no eficientes en espacio (Naseem y cols., 2020, pp. 10-11; Lu, 2020).

4.6.2. TF-IDF

La versión ponderada del modelo *bag-of-words* construye una representación del mensaje a partir de la frecuencia de las palabras que lo constituyen (Naseem y cols., 2020, p. 11). Las palabras vacías son frecuentes en el *corpus* y, probablemente también en el mensaje, por lo que su relevancia es prácticamente nula. Por el contrario, las palabras no vacías frecuentes en el mensaje son más relevantes en cuanto a su contribución semántica al significado del mensaje (Jurafsky y Martin, 2020, pp. 106-107). El algoritmo TF-IDF utiliza dos métricas para cuantificar la relevancia de una palabra en un mensaje:

- **Term Frequency (TF):** La frecuencia de un término se computa como la frecuencia relativa de una palabra en un texto, siendo posible aplicar una transformación booleana o logarítmica para expresarlo en la escala deseada (Naseem y cols., 2020, p. 11; Jurafsky y Martin, 2020, p. 107).
- **Inverse document frequency (IDF):** La frecuencia inversa de documento cuantifica cuán frecuente es una palabra en un *corpus* dado. Dado que las palabras vacías pueden tener una frecuencia de término elevada, necesitamos penalizar el hecho de que sean comunes en cualquier texto (Naseem y cols., 2020, p. 11; Jones, 1972)

Sea D el conjunto de textos que constituye nuestro *corpus*. Decimos que la TF-IDF de una palabra w en un texto $t = w_1 \dots w_k$ viene dada por la expresión 1 (Naseem y cols., 2020, p. 11).

$$tf-idf(w, t, D) = tf(w, t) \cdot \log \left(\frac{|D|}{\sum_{d \in D} \chi_d(t)} \right) \quad (1)$$

donde $tf(w, t)$ es la frecuencia del término w en el texto t , y $\chi_d(t)$ es la función característica del predicado de pertenencia o no de t a la secuencia d , esto es, $\chi_d(t) = 1$ si y solo si t es una

palabra que aparece en el texto t . En otro caso, $\chi_d(t) = 0$.

Al igual que con el esquema de codificación anterior, para representar un texto debemos aplicar una codificación *One-Hot* al vocabulario. El vector que codifica a un texto se computaría como una combinación lineal de los vectores *One-Hot*, donde el coeficiente α_i se corresponde con la $tf-idf$ de la palabra i ésima del vocabulario.

Los dos esquemas de codificación expuestos hasta ahora presentan las mismas desventajas. En primer lugar, ignoran la información sobre el orden de las palabras, y su simplicidad limita su capacidad para capturar todas las propiedades semánticas de interés. Por otro lado, sus representaciones vectoriales crecen linealmente con el tamaño del vocabulario, por lo que si no aplicamos ninguna técnica de reducción, son métodos ineficientes en espacio.

4.6.3. Word2vec

Suponiendo cierta la hipótesis distribucional, palabras que aparezcan en contextos similares tienen significados semejantes (Harris, 1954). En aras de garantizar que se preserva la semántica en la codificación, un modelo de representación distribuida debe computar los vectores distribuidos de tal forma que cuanto más similares sean los contextos en los que aparecen unas palabras w_1 y w_2 , menor sea la distancia entre los vectores distribuidos de w_1 y w_2 .

Para el aprendizaje de representaciones distribuidas, Mikolov, Chen, Corrado, y Dean (2013) proponen dos arquitecturas log-lineales de redes neuronales de dos capas, que presentan una menor complejidad computacional que otros modelos basados en redes neuronales al eliminar la no linealidad de la capa oculta.

- **Continuous Skip-gram:** Este modelo tiene por objeto resolver el problema de determinar el contexto más probable en el que podemos encontrar una palabra determinada. Dado un vocabulario de k palabras, la entrada de la red es un vector *one-hot* de k componentes. La matriz de pesos \mathbf{W} de la capa oculta almacenará los vectores distribuidos que queremos aprender. Al final del entrenamiento, la fila i ésima de \mathbf{W} será el *embedding* del vector i ésimo del vocabulario. La capa de salida computa para cada palabra del vocabulario la probabilidad de que aparezca en el contexto de la palabra que hemos introducido en la red (Mikolov, Chen, y cols., 2013; Naseem y cols., 2020, p. 14).
- **Continuous Bag of words (CBOW):** Resuelve el problema opuesto al anterior, es de-

cir, dado un contexto de m palabras, determina la palabra más probable que puede estar presente en dicho contexto. La entrada \mathbf{x} de la red es la concatenación de m vectores *one-hot* de k componentes. La matriz de pesos \mathbf{W} de la capa intermedia, al igual que en el modelo anterior, almacenará los *embeddings* de las palabras. A diferencia de la arquitectura *Continuous Skip-gram*, en la capa oculta se produce una suma de los *embeddings* de las palabras que forman parte del contexto:

$$\mathbf{xW} = \mathbf{v}_1 + \dots + \mathbf{v}_m \quad (2)$$

donde $\mathbf{v}_1, \dots, \mathbf{v}_m$ son los vectores distribuidos de las palabras que forman parte del contexto según \mathbf{x} . La capa de salida computa para cada palabra del vocabulario la probabilidad de que aparezca en el contexto que hemos introducido en la red (Mikolov, Chen, y cols., 2013; Naseem y cols., 2020, p. 14).

Las capas finales de estos modelos aplican *softmax* jerárquico para calcular las probabilidades. Otra de las funciones objetivo propuesta para entrenar la red es *negative sampling* (Mikolov, Sutskever, Chen, Corrado, y Dean, 2013, pp. 3-4).

Existen varias técnicas para el cómputo del vector distribuido de un texto a partir de los vectores distribuidos de sus palabras. La plataforma SpaCy, por ejemplo, suma los vectores de cada palabra y divide el resultado por el número de palabras, o lo que es lo mismo, el vector distribuido de un texto es la media de los vectores de sus palabras (*Linguistic Features · spaCy Usage Documentation, s.f.*). Otros métodos propuestos son *SentenceToVec* y *DocumentToVec* que son extensiones de *Word2vec* a representaciones de enunciados y documentos, respectivamente (Kumari y Shashi, 2019, p. 306).

4.6.4. Representación distribuida contextual: ELMo y BERT

Uno de los problemas de los modelos pre-entrenados de representaciones de palabras como Word2vec es la unicidad de la representación de palabras polisémicas (Peters y cols., 2018). Por ejemplo, estos modelos asignarían a la palabra 'gato' en los textos 'El gato hidráulico de mi coche' y 'El gato de mi vecina', un mismo vector distribuido, a pesar de que el significado de la palabra 'gato' difiere notablemente de un texto a otro.

ELMo (*Embeddings from Language Models*) es un modelo de representación que asigna a una palabra w_i de un texto $w_1 \dots w_i \dots w_n$ una representación que tiene en cuenta el contexto

$\{w_1 \dots w_{i-1}, w_{i+1}, \dots, w_n\}$ en el que aparece la palabra a codificar. Las representaciones dependientes del contexto tienen como base la información aprendida por un modelo de lenguaje bidireccional, que es capaz de predecir la palabra anterior o siguiente dado un contexto futuro o pasado (Peters y cols., 2018). En cada capa un LSTM codifica el contexto pasado y otro LSTM que consume la entrada en un sentido inverso codifica el contexto futuro. La salida de la capa es la concatenación de la codificación de ambos contextos (Liu y cols., 2020).

Otro modelo de representación distribuida contextual es BERT (*Bidirectional Encoder Representations from Transformers*). Su arquitectura está basada en un *stack* de codificadores de un *Transformer* bidireccional (Devlin, Chang, Lee, y Toutanova, 2019), lo cual le permite tener en cuenta las interacciones entre el contexto pasado y futuro de las palabras que en el modelo ELMo no son explotadas al simplemente concatenar las codificaciones de los mismos (Liu y cols., 2020). BERT es entrenado con dos tareas de modelado de lenguaje que le permiten aprender relaciones de contexto entre palabras y enunciados:

- **Masked Language Modelling (MLM):** Dada una secuencia de *tokens* en la que algunos de ellos han sido enmascarados o corrompidos, el modelo debe ser capaz de recuperar la secuencia inicial (Naseem y cols., 2020, p. 22; Devlin y cols., 2019).
- **Next sentence prediction (NSP):** Dado un par de secuencias de *tokens* s_1 y s_2 , el modelo debe determinar si s_2 puede ser una secuencia que sigue a s_1 (Naseem y cols., 2020, p. 23; Devlin y cols., 2019).

4.7. Preprocesamiento de mensajes usando nuestro *framework*

4.7.1. Transformaciones a nivel de carácter

El servicio `TextNormalizer` ofrece varias operaciones de normalización a nivel de carácter:

- **Transformar el texto a mayúsculas o minúsculas:** A través del parámetro `letter_case` a `'lower'` o `'upper'`.
- **Simplificación o eliminación de elipsis:** El parámetro `ellipsis` admite dos valores: `'remove'` y `'normalize'`. El primer modo consiste en la eliminación completa de cual-

quier secuencia de tres o más puntos consecutivos, mientras que el segundo reduce estas secuencias a `'...'`.

- **Simplificación o eliminación de signos de exclamación:** El parámetro denominado `exclamation_marks` admite dos valores: `'remove'` y `'normalize'`. El primer modo consiste en la eliminación completa de ocurrencias de `'¡'` y `'!'`, mientras que el segundo reduce secuencias de dos o más signos de exclamación a uno.
- **Simplificación o eliminación de signos de interrogación:** Equivalente al anterior, pero en lugar de signos de exclamación aplica un tratamiento a los signos de interrogación a través del parámetro `question_marks`.
- **Eliminación de caracteres no ASCII:** Si el parámetro `to_ascii` es `True`, se eliminan o reemplazan los caracteres no ASCII. Por ejemplo, la `'á'` sería reemplazada por `'a'`.

En el siguiente ejemplo podemos comprobar cómo se inicializa y consume el servicio:

```
1 from smartbot.services.preprocessing.normalization.text_normalization import \
2     TextNormalizer
3
4 text_normalizer = TextNormalizer(
5     letter_case="lower", ellipsis="remove",
6     exclamation_marks="normalize", question_marks="normalize",
7     to_ascii=True
8 )
9
10 msg = "¡¡¡Buenos días!!! Cómo ha ido todo?? No muy bien aquí....."
11
12 normalized_text = text_normalizer.execute(text=msg)
13
14 print(normalized_text) # buenos dias! como ha ido todo? no muy bien aqui
```

Código 7: Ejemplo de inicialización y ejecución del servicio de normalización de texto.

4.7.2. Modelo de *token*

Un mensaje en lenguaje natural puede ser segmentado en unidades conocidas como *tokens*. Un *token* es una subsecuencia contigua de caracteres que puede contener información

sobre su etiquetado gramatical, dependencias sintácticas respecto a otros *tokens*, etc. No necesariamente deben coincidir con las palabras de un texto, puesto que puede ser necesario tratar expresiones de varias palabras como las locuciones como una sola entidad.

Existen ciertas plataformas como SpaCy que a partir de modelos de lenguajes pueden llevar a cabo una segmentación de un texto o documento en *tokens* que pueden contener información lingüística de interés. Un modelo de *token* puede contener información que sea suficiente o no para el consumo de ciertos servicios cognitivos en etapas posteriores. Por lo anterior, puede ser necesario recurrir a más de un modelo de *token* y para ello nuestra librería introduce una abstracción a nivel de *token*, secuencia y *span* de *tokens*. Todos los modelos de estas unidades deben heredar de una clase base `BaseToken` y deben como mínimo almacenar la representación textual del *token*.

Determinados servicios cognitivos pueden consumir una secuencia de *tokens* de cualquier modelo, por lo que introducir esta clase nos permite que servicios como los extractores de entidades sean independientes de las plataformas o librerías de preprocesamiento que utilicemos. Por el contrario, ciertos servicios pueden acotar el conjunto de modelos de *tokens* que pueden procesar. El servicio `SpacyFeaturizer`, por citar un ejemplo, extrae características de *tokens* generados por SpaCy.

Se proporcionan dos implementaciones de estos modelos `SpacyToken` y `SimpleToken`. El primero de ellos es un *wrapper* de la clase `Token` de SpaCy que ofrece información sobre dependencias sintácticas, etiquetado gramatical, entre otros datos. La implementación `SimpleToken` almacena solo la representación textual y puede ser utilizado para crear secuencias a la hora de probar un nuevo servicio. En la siguiente sección introduciremos los segmentadores y ejemplos de extracción de *tokens* de estos modelos.

4.7.3. Segmentación

Los servicios de segmentación presentan una dependencia con el modelo de *token* escogido. Esto se debe a que la segmentación es la operación que los genera.

El resultado de una segmentación de este tipo es una secuencia de *tokens*. Definimos una clase para estas secuencias denominada `BaseTokenSequence` que define una serie de operaciones sobre una secuencia de *tokens* de la clase `BaseToken`. Se requiere que los *tokens* sean accesibles de forma secuencial (i.e. la clase debe ser iterable) y de forma aleatoria a través de

un indizador. Además de lo anterior, se define una operación `get_span` que será de utilidad en tareas posteriores como la extracción de entidades. `get_span` devuelve una subsecuencia contigua de *tokens* o *span* que ha sido accedida a través de un índice a nivel de carácter y el número de *caracteres*. Un *span* almacena los índices que definen el rango de *tokens* y una referencia a la secuencia a la que pertenecen. La clase base de los *spans* es `BaseTokenSpan`.

En aquellos casos en los que sea necesario recurrir a SpaCy para efectuar la segmentación, recurriremos al servicio `SpacyTokenizer`. A través de un parámetro de configuración debemos especificar un identificador o modelo de lenguaje de SpaCy que tengamos instalado en el equipo. A modo de ejemplo, supongamos que instalamos desde consola uno de los modelos para el lenguaje español

```
python -m spacy download es_core_news_md
```

El siguiente fragmento de código instancia un segmentador basado en SpaCy que carga el modelo de lenguaje anterior, consume un texto en lenguaje natural y genera una secuencia de tipo `SpacyTokenSequence` de *tokens* de la clase `SpacyToken`.

```
1 from smartbot.services.preprocessing.segmentation import SpacyTokenizer
2
3 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
4
5 msg = "Quiero tomar dos hamburguesas con bacon y queso"
6
7 seq = tokenizer.execute(text=msg)
8
9 print(seq) # Quiero tomar dos hamburguesas con bacon y queso
10
11 print(len(seq)) # 8 tokens
12
13 print(seq[3]) # hamburguesas
```

Código 8: Ejemplo de inicialización y ejecución del servicio de segmentación de texto en secuencias de *tokens* de SpaCy.

Una alternativa a un segmentador basado en SpaCy es el servicio `WhitespaceTokenizer` que segmenta un texto en *tokens* suponiendo que el carácter espacio es el delimitador.

```

1 from smartbot.services.preprocessing.segmentation import WhitespaceTokenizer
2
3 tokenizer = WhitespaceTokenizer()
4
5 msg = "Quiero tomar dos hamburguesas con bacon y queso"
6
7 seq = tokenizer.execute(text=msg)
8
9 print(seq) # Quiero tomar dos hamburguesas con bacon y queso
10
11 print(len(seq)) # 8 tokens
12
13 print(seq[3]) # hamburguesas

```

Código 9: Ejemplo de inicialización y ejecución del servicio de segmentación de texto en secuencias de *tokens* simples.

Independientemente del modelo de *token* subyacente, un servicio puede extraer una subsecuencia de la secuencia de *tokens* a partir de la información posicional de la misma basada en índices de caracteres. Dado que en los conjuntos de entrenamiento de ciertas tareas de comprensión de lenguaje natural se especificará la presencia de una entidad a partir de la posición de su primer y último caracteres, es necesario aplicar una transformación de este tipo de índices a índices de *tokens* para poder extraer cadenas correspondientes a uno o más *tokens* consecutivos.

```

1 msg = "Quiero tomar dos hamburguesas con bacon y queso"
2
3 # Posicion de la entidad 'hamburguesas' en el texto
4 start_char = 17
5 length = 11
6
7 # Se aplica una transformacion de indices de caracteres a indices de token
8 span = seq.get_span(char_index=start_char, length=length)
9
10 print(seq[span.index_first_token:span.index_last_token]) # hamburguesas

```

Código 10: Ejemplo de extracción de *span* (i.e. una secuencia de *tokens* contiguos).

4.7.4. Características de los *tokens*

Estos servicios construyen una lista con un diccionario por cada *token* de una secuencia. Cada diccionario contiene una serie de características extraídas manualmente del *token* en cuestión. No necesariamente todos los diccionarios tienen las mismas claves, puesto que pueden existir características sobre *tokens* anteriores y posteriores que no serían aplicables al inicio o final de la secuencia, por ejemplo.

El servicio `Featurizer` es capaz de extraer características de un *token*. Al instanciar el servicio debemos seleccionar una subconjunto de las características de interés que pueden ser booleanas, numéricas o textuales. Por ejemplo, las características booleanas `isdigit` e `isupper` tienen un valor `True` si y solo si el *token* es un número o todos sus caracteres están en mayúsculas, respectivamente. La característica `lower` es de naturaleza textual y permite llevar a cabo una normalización que puede ser necesaria para el servicio que consuma esta información.

```
1 from smartbot.services.preprocessing.featurizer.featurizer import Featurizer
2 from smartbot.services.preprocessing.segmentation import WhitespaceTokenizer, \
3   SpacyTokenizer
4
5 featurizer = Featurizer(features=[
6     "bias", # Constante a 1.0
7     "isdigit", # True si el token es un numero
8     "lower", # El texto del token en minusculas
9     "istitle", # True sii el casing es XxxX
10    "firstword" # True sii es el primer token de la secuencia
11 ])
12
13 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
14
15 msg = "Precio a PAGAR: 13 euros"
16
17 seq = tokenizer.execute(text=msg)
18 print(seq)
19 features_per_token = featurizer.execute(tokens=seq)
20
21 features_precio = features_per_token[0]
22 features_pagar = features_per_token[2]
23 features_13 = features_per_token[-2]
24
25 print(features_precio) # {..., 'lower': 'precio', 'istitle': True, 'firstword': True}
```

```

26 print(features_pagar) # {..., 'lower': 'pagar', 'istitle': False, 'firstword': False}
27 print(features_13) # {..., 'isdigit': True, ...}

```

Código 11: Ejemplo de extracción de características a nivel *token*.

Si nuestros *tokens* son de SpaCy, podemos extraer más información que no es posible extraer con `Featurizer`. Este servicio ofrece características que pueden ser computadas si disponemos de la información mínima de un *token*: su contenido textual. `SpacyFeaturizer` es una subclase de `Featurizer` que introduce características como el etiquetado gramatical que pueden ser extraídas de un *token* de SpaCy.

```

1 from smartbot.services.preprocessing.featurizer.spacy_featurizer import SpacyFeaturizer
2 from smartbot.services.preprocessing.segmentation import SpacyTokenizer
3
4 featurizer = SpacyFeaturizer(features=[
5     "bias", # Constante a 1.0
6     "isdigit", # True si el token es un numero
7     "lower", # El texto del token en minusculas
8     "istitle", # True sii el casing es Xxxx
9     "pos", # etiqueta gramatical
10    "lemma" # lema
11 ])
12
13 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
14
15 msg = "4 pizzas de jamón y 2 hamburguesas"
16
17 seq = tokenizer.execute(text=msg)
18
19 features_per_token = featurizer.execute(tokens=seq)

```

Código 12: Ejemplo de extracción de características a nivel *token* de SpaCy.

Si queremos incluir una característica booleana que se evalúe como cierta si y solo si hay *match* con una expresión regular, debemos incluir en la lista de características la cadena `'regex:'` seguida de la expresión regular. Por ejemplo, `'regex:^[0-9]+[A-Za-z]$'` tendrá un valor de `True` si el *token* tiene el formato de un DNI.

También es posible especificar que queremos incluir varias características sobre un *token* a *i* posiciones del actual. Por ejemplo, el identificador `-1:istitle` identifica a la característica `istitle` del *token* anterior al que estamos procesando. La posibilidad de incluir este *offset* está soportado por todas las características de forma automática, por lo que si construimos un servicio de construcción de características que herede de `Featurizer` solo deberemos redefinir el método `define_feature(name, offset)` para devolver funciones que reciben por argumentos una secuencia de *tokens* y un índice. Estas funciones son las características.

4.7.5. Servicios de codificación: modelos de representación clásicos

Los modelos clásicos introducidos en este capítulo están disponibles a través de los servicios `BagOfWordsEncoder` y `TfidfEncoder`. El primero de ellos permite obtener la representación en forma de *bag-of-words* de un texto. Se puede configurar el servicio para que no se especifique la multiplicidad de cada palabra sino su ausencia o presencia (i.e. modelo binario). El entrenamiento del esquema de codificación tiene lugar al invocar `train` con una lista de secuencias de *tokens* `tokens`.

```
1 from smartbot.services.preprocessing.representation import BagOfWordsEncoder
2 from smartbot.services.preprocessing.segmentation import SpacyTokenizer
3
4 corpus = [
5     "este modelo de coche no me gusta",
6     "este coche rojo es barato",
7     "un coche azul no muy caro",
8     "me gusta este coche rojo"
9 ]
10
11 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
12 encoder = BagOfWordsEncoder(binary=False)
13
14 # Entrenamiento
15 corpus_tokens = tokenizer.execute(
16     text=corpus,
17     execution_mode=ServiceExecutionMode.SEVERAL_INPUTS
18 )
19 encoder.train(tokens=corpus_tokens)
20
21 # Consumir entrada
```



```

22 msg = "un coche azul"
23
24 shared_memory = {
25     "text": msg
26 }
27
28 vector = tokenizer.execute(**shared_memory)
29 shared_memory["tokens"] = vector
30
31 vector = encoder.execute(**shared_memory)
32
33 print(vector) # [1 0 0 1 0 0 0 0 0 0 0 0 0 1]

```

Código 13: Ejemplo de entrenamiento y consumo de servicio de codificación basado en *bag-of-words*.

En el ejemplo anterior utilizamos un servicio de segmentación para segmentar los mensajes y los *tokens* generados por este son consumidos por un servicio de codificación. Tal y como mencionamos en secciones anteriores, ciertos servicios consumen *tokens* de cualquier modelo y los servicios de codificación son un ejemplo de estos.

El servicio de codificación `TfidfEncoder` está basado en el esquema clásico alternativo a *bag-of-words*: TF-IDF. Su interfaz es muy similar a la del servicio `BagOfWordsEncoder` por lo que en el ejemplo anterior solo tendríamos que sustituir la inicialización del codificador. Para este *corpus* y entrada, y servicio de segmentación el vector Tfidf generado fue el siguiente:

```
[0.66338461, 0, 0, 0.34618161, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.66338461]
```

4.7.6. Servicios de codificación: modelos de representación distribuidos

Proporcionamos tres servicios de codificación basados en las arquitecturas Word2vec, Doc2vec y ELMo. El servicio `SpacyEncoder` recibe una lista de *tokens* del modelo `SpacyToken` y computa el vector distribuido de un mensaje como la media de los vectores de cada uno de los *tokens* que lo constituyen. Este servicio es una opción si ya hemos utilizado el servicio de segmentación `SpacyTokenizer` en una etapa anterior del preprocesamiento.

```

1 from smartbot.services.preprocessing.representation import SpacyEncoder
2 from smartbot.services.preprocessing.segmentation import SpacyTokenizer

```

```

3
4 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
5 encoder = SpacyEncoder()
6
7 msg = "quiero reservar mesa para cuatro personas"
8
9 shared_memory = {
10     "text": msg
11 }
12
13 vector = tokenizer.execute(**shared_memory)
14 shared_memory["tokens"] = vector
15
16 vector = encoder.execute(**shared_memory)
17
18 print(vector) # [ 0.43536964 -0.28229314 ... -0.02233332 0.21482164]
19 print(len(vector)) # 300 dimensiones

```

Código 14: Ejemplo de inicialización y consumo de servicio de codificación de SpaCy.

Al contrario que `SpacyEncoder`, el siguiente servicio que introducimos no está pre-entrenado, por lo que debemos disponer de un corpus y aplicar una segmentación a nivel de palabra a cada texto del mismo. El servicio `Doc2vecEncoder` implementa la arquitectura Doc2vec que es una extensión de Word2vec para enunciados. Dado que está diseñado expresamente para codificar secuencias de palabras, puede obtenerse un mejor rendimiento que con `SpacyEncoder` que simplemente computa la media de los vectores de las palabras.

```

1 from smartbot.services.preprocessing.representation import Doc2vecEncoder
2 from smartbot.services.preprocessing.segmentation import SpacyTokenizer
3 from smartbot.arch.service import ServiceExecutionMode
4
5 corpus = [
6     "teneis pizza en el menu",
7     "me gustaria reservar mesa",
8     "teneis libre una mesa para cuatro personas",
9     "una mesa en el exterior a ser posible"
10 ]
11
12 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
13 encoder = Doc2vecEncoder(

```

```

14     case_sensitive=True,
15     vector_length=200
16 )
17
18 # Entrenamiento
19 tokens = tokenizer.execute(
20     text=corpus,
21     execution_mode=ServiceExecutionMode.SEVERAL_INPUTS
22 )
23 encoder.train(tokens=tokens)
24
25 # Codificación de una instancia
26 msg = "quiero reservar mesa para cuatro personas en exterior"
27
28 shared_memory = {
29     "text": msg
30 }
31 vector = tokenizer.execute(**shared_memory)
32 shared_memory["tokens"] = vector
33
34 vector = encoder.execute(**shared_memory)
35
36 print(vector) # [-1.6771752e-02 -1.7826591e-02 ... 4.1204169e-03 6.7010331e-03]
37 print(len(vector)) # 200 dimensiones

```

Código 15: Ejemplo de inicialización, entrenamiento y ejecución de servicio de codificación basado en Doc2vec.

Finalmente, disponemos del servicio de codificación distribuida `PretrainedELMoEncoder` que utiliza un modelo de lenguaje pre-entrenado para la arquitectura ELMo. La principal ventaja frente a los servicios de codificación anteriores es que un modelo distribuido contextual y tal y como expusimos en la sección teórica sobre este tipo de esquemas de codificación, son capaces de capturar la polisemia de las palabras.

```

1 from smartbot.services.preprocessing.representation.elmo_encoder import \
2     PretrainedELMoEncoder
3 from smartbot.services.preprocessing.segmentation import WhitespaceTokenizer
4
5 tokenizer = WhitespaceTokenizer()
6 import time

```

```

7
8 start = time.time()
9 # modelo descargado de http://vectors.nlpl.eu/repository/
10 encoder = PretrainedELMoEncoder(model_path="194.zip")
11 end = time.time()
12
13 # La carga de un modelo de ELMo de NLPL en formato zip suele llevar unos minutos
14 # por el peso de los modelos
15 print("Elapsed time:", end - start) # 147.89 segundos
16
17 tokens = tokenizer.execute(text="I want to book a flight to Moscow")
18 vector = encoder.execute(tokens=tokens)
19
20 print(vector) # [-0.02987773 -0.0127002 ... 0.03256254 0.0170264]

```

Código 16: Ejemplo de inicialización y ejecución de servicio de codificación basado en ELMo.

4.8. Clasificación de intenciones

4.8.1. Introducción

Nuestro agente conversacional, partiendo de la hipótesis de que todo acto comunicativo es ejecutado con una intención, ejecuta un servicio de clasificación de intenciones siempre que recibe un mensaje con contenido textual.

Dado que nuestros *chatbots* no cuestionan el porqué un usuario ha enviado un cierto mensaje, los símbolos de las creencias, objetivos e inferencias exactas que dieron lugar al mensaje en el agente emisor son traducidos a otro tipo de símbolos estructuralmente más simples: las intenciones. Una intención es una clase de equivalencia de tuplas de creencias, objetivos e inferencias que se define por comprensión y, generalmente, de forma informal. Representaremos un símbolo de intención con un *string*.

Ejemplo 4.4

La clase de intención 'SALUDO' se puede definir como la clase de aquellos mensajes que son emitidos con la intención de iniciar o reiniciar una conversación, o bien con la intención de confirmar al agente que envió el primer saludo que es posible iniciarla en este momento.

Definiremos las intenciones aportando una lista de intenciones en el conjunto de entrenamiento NLU. Cada intención es definida con una lista de mensajes que pertenecen o fueron

emitidos con dicha intención.

En el fragmento de código YAML ?? mostramos un ejemplo de la definición de una intención 'greeting' en el conjunto de datos NLU.

```
1 greeting:
2   - text: Buenos dias
3   - text: Hola
4   - text: Hola!
5   - text: Buenas
6   - text: Hola, ¿qué tal?
7   - text: ¡Buenos días!
```

Código 17: Ejemplo de definición de instancias para una intención.

4.8.2. Clasificación con máquinas de soporte vectorial

El problema de clasificación de la intención de un mensaje es un problema de clasificación multinomial en el marco de trabajo del aprendizaje automático.

Dado que algunos estudios concluyen que los conjuntos de datos para clasificación de documentos o texto suelen ser linealmente separables (Joachims, 1998), un clasificador lineal podría ser una opción adecuada para abordar esta tarea de comprensión.

En el caso que nos ocupa, hemos decidido por optar por una máquina de soporte vectorial para la clasificación de intenciones. Una máquina de soporte vectorial es un clasificador lineal y binario basado en el principio de maximización del margen entre instancias de ambas clases. Este modelo divide el espacio en dos regiones, una por cada clase, que están separadas por el hiperplano que maximice el margen entre ambas (Moens, 2006, pp. 93-94; Adankon y Cheriet, 2009).

Una de las ventajas que presenta este modelo es el uso de un mecanismo de regularización que consigue mejorar su capacidad de generalización y hacer frente al *overfitting* (Xu, Caramanis, y Mannor, 2008). Dado que el fenómeno del *overfitting* puede darse en aquellos casos en los que el número de características sea superior al número de instancias (Guyon y Elisseeff, 2003, p. 1665) y los servicios de codificación pueden generar vectores de aridad elevada, las máquinas de soporte vectorial pueden ser un modelo con propiedades beneficiosas para resolver tareas de clasificación de textos.

Aunque estos modelos son binarios, existen métodos para extenderlos a clasificadores multinomiales entrenando tantos modelos como intenciones (p. ej. *one-against-all*) o pares de intenciones (p. ej. *one-against-one*), y recurriendo a estrategias de *voting* (Hsu y Lin, 2002, pp. 2-4).

4.8.3. Servicio de clasificación de intenciones

Para entrenar un servicio de clasificación de intenciones basado en máquinas de soporte vectorial, debemos crear e inicializar una instancia del servicio `IntentClassifier`. Esta clase es un *wrapper* de la implementación de máquina lineal de soporte vectorial de la librería Sklearn. Se recurre al método *one-against-one* para permitir la clasificación multinomial.

Uno de los parámetros de inicialización más relevantes es `output_mode`. En función de su valor, la salida del servicio será una intención (`output_mode='intent'`) o un diccionario con probabilidades por cada intención definida (`output_mode='probabilities'`). Su valor predeterminado es `'intent'`.

```
1 from smartbot.services.nlu.intent.svm_intent import IntentClassifier
2
3 intent_cls = IntentClassifier(
4     model_path="models/clasifier.pkl",
5     output_mode="probabilities",
6     cv_folds=2,
7     min_confidence=0.45,
8     intent_fallback="didnt_understand",
9     grid_search_parameters={
10         "C": [0.1, 10, 100, 1000],
11         "gamma": [0.003, 0.004, 0.005],
12         "kernel": ["linear", "poly", "rbf", "sigmoid"]
13     }
14 )
```

Código 18: Ejemplo de inicialización del servicio de clasificación de intenciones.

Para entrenar este servicio debemos invocar al método `train` y proporcionarle una lista de vectores `vectors` que codifican los mensajes y una lista de intenciones `intents`. En nuestra

implementación, se aplica una división del *dataset* en un conjunto de entrenamiento y de pruebas. El tamaño de este último se puede configurar a través de `test_size`.

El parámetro `grid_search_parameters` permite definir un espacio de hiperparámetros que será utilizado para optimizar estos parámetros a través de una búsqueda exhaustiva. Se proporcionan más detalles sobre estos parámetros y los que no mencionamos explícitamente en esta sección en la documentación del *framework*.

Otros de los parámetros a destacar de la implementación de este servicio son `intent_fallback` y `min_confidence`. El primero de ellos es el identificador de una intención especial que será el resultado de la clasificación de una instancia si y solo si la confianza del modelo sobre la predicción es notablemente baja. Se introduce el parámetro `min_confidence` para establecer una cota inferior a la confianza necesaria para que la predicción sea considerada fiable. Si la probabilidad de que la predicción sea fiable es estrictamente inferior a dicho parámetro, el servicio determinará que la intención del mensaje es `intent_fallback`.

Para no entrenar el modelo en cada ejecución debemos proporcionar una ruta en la que almacenar el modelo. El atributo `model_path` permite especificar una localización para el modelo. Este campo está disponible en el resto de servicios entrenables del *framework* y es utilizado tanto para cargar como guardar un modelo. Al inicializar el servicio entrenable se hará una primera comprobación para determinar si existe ya un modelo en esa localización. En caso afirmativo, se carga el modelo y no se entrena el servicio.

En el siguiente ejemplo continuamos el anterior y entrenamos el servicio con un conjunto de intenciones. Omitimos la segmentación y codificación para hacer énfasis en cómo hay que entrenar y consumir este servicio.

```
1 X, y = intent_dataset()  
2  
3 intent_cls.train(vectors=X, intents=y)
```

Código 19: Ejemplo de entrenamiento del servicio de clasificación de intenciones.

El conjunto de intenciones del ejemplo define las intenciones `'greeting'` y `'farewell'`. Una vez entrenado el servicio, procedemos a consumirlo con un nuevo mensaje que debería ser clasificado como `'greeting'`, y que no está presente en el *dataset*. Dado que `output_mode`

es `'probabilities'`, la salida será un diccionario con la probabilidad de que el mensaje sea un saludo o una despedida.

```
1 # Omitimos la segmentacion y codificacion. Tienen lugar
2 # en una funcion local preprocess
3 message_vector = preprocess("Buenos dias amigo que tal?")
4
5 output = intent_cls.execute(vector=message_vector)
6 print(output)
```

Código 20: Ejemplo de ejecución del servicio de clasificación de intenciones.

El valor de la variable `output` mostrado por consola para el ejemplo anterior es el siguiente:

```
{ 'farewell': 0.22633621458172648, 'greeting': 0.7736637854182736 }
```

Tal y como podemos apreciar, el servicio ha sido capaz de clasificar correctamente el mensaje en cuestión como un saludo. Si hubiéramos configurado `output_mode` como `'intent'`, la salida habría sido `'greeting'`.

4.9. Clasificación del sentimiento

4.9.1. Introducción

La no observabilidad de las emociones nos obliga a llevar a cabo una medición indirecta de las mismas (Picard, 1995). Estos estados internos no observables pueden ser representados en el modelo bidimensional de espacio afectivo circunplejo. En este modelo cada emoción se representa con una agradabilidad y activación (Russell, 1980).

En análisis de sentimiento, la polaridad es la versión discretizada de la agradabilidad del modelo circunplejo. La activación es un concepto equivalente al de intensidad del sentimiento.

En nuestro *framework* permitimos la definición de clases de sentimiento de una forma similar a las intenciones. En cada mensaje del conjunto de datos NLU, podemos especificar un sentimiento como en el ejemplo siguiente.

```
1 opinion_about_app:
2 - text: Me encanta vuestra app
```



```
3 sentiment: 1
4 - text: La aplicación está bien
5 sentiment: 1
6 - text: Bueno no está mal la app
7 sentiment: 0
8 - text: No me gusta la app no es buena
9 sentiment: -1
10 - text: Odio esta app muy mala
11 sentiment: -1
```

Código 21: Ejemplo de definición de instancias para una intención, incluyendo el campo opcional para el sentimiento.

En la comunicación verbal podemos transmitir emociones que sentimos o no a través de palabras emocionales, esto es, palabras que tienen una determinada connotación emocional (Abbassi, Blanchette, Ansaldo, Ghassemzadeh, y Joannette, 2015). Asimismo, se pueden transmitir sentimientos racionales que no contienen una carga emocional, pero son subjetivos (“A Practical Guide to Sentiment Analysis”, 2017, pp. 15-16).

Ejemplo 4.5

Las palabras 'bueno' y 'genial' son palabras emocionales con una polaridad positiva, mientras que 'malo' y 'horrible' son palabras emocionales con una polaridad negativa. En cuanto a su intensidad, las palabras 'genial' y 'horrible' transmiten una mayor intensidad que 'bueno' y 'malo'.

Ejemplo 4.6

Un ejemplo de sentimiento racional es la valoración subjetiva sobre la utilidad de un producto. En el mensaje

Estoy muy contento con la app. Es muy útil para gestionar mi día a día.

se transmite un juicio sobre la aplicación que tiene una polaridad positiva y una intensidad medio-alta por el uso del modificador 'muy'.

El problema de detección de polaridad consiste en determinar la polaridad media de un texto a partir de la polaridad e intensidad de los sentimientos transmitidos por las palabras que lo constituyen (“A Practical Guide to Sentiment Analysis”, 2017, pp. 19-22).

Nuestro *framework* proporciona la implementación de un servicio de clasificación de sentimientos que basado en Naive Bayes y regresión logística, por lo que expondremos brevemente estos modelos en las siguientes secciones.

4.9.2. Clasificación con Naive Bayes

Un clasificador bayesiano asigna la clase más probable al vector de características que debe clasificar (Rish, 2001). Naive Bayes es un clasificador bayesiano que supone que las características son independientes dada la clase (Rish, 2001). Aunque estas suposiciones sean poco realistas, este algoritmo es capaz de competir con otros modelos como el de regresión logística en ciertos casos (Jurafsky y Martin, 2020, p. 81; Rish, 2001).

Al tratarse de un modelo generativo, en lugar de calcular directamente la probabilidad de que un mensaje tenga una polaridad positiva o negativa y dado un vector de características x , computa el *likelihood* $P(x | y)$ y la probabilidad a priori $P(y)$ de las clases de polaridad.

Dada una instancia $\mathbf{x} = [f_1 \dots f_k]$ con k características, el *likelihood* $P(x | y)$ suponiendo cierta la suposición de Naive Bayes viene dado por la expresión siguiente:

$$P(x | y) = \prod_{f \in F} P(f | y) \quad (3)$$

Los clasificadores bayesianos maximizan la probabilidad a posteriori de una clase dada una instancia x (Rish, 2001), por lo que la expresión a maximizar durante la clasificación en Naive Bayes se reduce a:

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(y) \prod_{f \in F} P(f | y) \quad (4)$$

4.9.3. Clasificación con regresión logística

La regresión logística es un clasificador binario probabilístico que, a diferencia de Naive Bayes, forma parte de la familia de los modelos discriminativos. Este tipo de clasificadores se limita a aprender cómo distinguir instancias de clases distintas, mientras que los generativos, entre los que se incluye Naive Bayes, pretenden entender las características de cada clase (Jurafsky y Martin, 2020, pp. 76-77).

La diferencia fundamental entre la regresión lineal y logística radica en la naturaleza discreta y binaria de la variable dependiente del modelo de regresión logística (Hosmer, Lemeshow, y Sturdivant, 2013, pp. 1-5). En el caso de la regresión logística, el valor esperado

para la variable dependiente y dado el valor de x , denotado $E(y | x)$ se modela con la función logística 5.

$$E(y | x) = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} \quad (5)$$

El proceso de optimización de los parámetros requiere de una función de coste que, en nuestro caso, será la de entropía cruzada binaria. La función a minimizar por el algoritmo de descenso por gradiente viene dado por la expresión 6 (Jurafsky y Martin, 2020, pp. 82-84).

$$\hat{\theta} = \arg \min_{\theta} \frac{1}{k} \sum_{i=1}^k \mathcal{L}_{CE}(f(x^{(i)}; \theta), y^{(i)}) \quad (6)$$

donde k es el número de instancias, y $\mathcal{L}_{CE}(x, y)$ es la función de coste de entropía cruzada binaria.

4.9.4. Servicios de clasificación del sentimiento

La clasificación del sentimiento de un mensaje puede llevarse a cabo a través del servicio `SentimentClassifier`. Este servicio soporta actualmente los dos algoritmos expuestos en esta sección y el desarrollador deberá elegir uno de ellos: un Naive Bayes multinomial o un modelo de regresión logística. Al igual que `IntentClassifier`, es un servicio *wrapper* de los modelos de la librería Sklearn correspondientes.

El servicio `SentimentClassifier` presenta una interfaz de configuración similar al del servicio de clasificación de intenciones basado en SVM. El parámetro `solver`, por el contrario, no está disponible en el servicio basado en SVM, pero sí en el de sentimientos siempre y cuando el algoritmo sea el de regresión logística. Su valor determina el tipo de algoritmo de optimización utilizado: 'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'. Por defecto, `solver` es `saga`.

Si el algoritmo utilizado es Naive Bayes se dispone de los mismos parámetros de configuración a excepción de `solver` y `tolerance`. En el manual se proporcionan más detalles sobre los parámetros de inicialización.

A la hora de optar por un algoritmo u otro debemos tener en cuenta que un modelo de regresión logística presenta una ventaja frente a un modelo Naive Bayes al permitir que las componentes de los vectores que codifican los mensajes estén correladas, mientras que Naive Bayes supone que las características son independientes condicionalmente (Jurafsky y Martin, 2020, p. 81)

Una posible inicialización del servicio `SentimentClassifier` con el algoritmo de regresión logística se expone a continuación:

```
1 from smartbot.services.nlu.sentiment.sentiment import SentimentClassifier
2
3 sentiment_cls = SentimentClassifier(
4     algorithm="logistic_regression",
5     cv_folds=3,
6     grid_search_parameters={
7         "C": [0.1, 10, 100, 1000],
8         "penalty": ["l1", "l2"]
9     }
10 )
```

Código 22: Ejemplo de inicialización del servicio de clasificación de sentimientos.

Para entrenar este servicio debemos invocar al método `train` y proporcionarle una lista de vectores `vectors` que codifican los mensajes y una lista de sentimientos `sentiments`. Al igual que en el servicio de clasificación de intenciones, se aplica una división del *dataset* en un conjunto de entrenamiento y de pruebas. Asimismo, el tamaño de este último también se puede configurar a través de `test_size`.

```
1 sentiment_cls.train(vectors=tfidf_vector, sentiments=sentiments)
```

Código 23: Ejemplo de entrenamiento del servicio de clasificación de sentimiento.

Los sentimientos contemplados en el conjunto de entrenamiento son `1` y `-1`. Vamos a clasificar el sentimiento de una opinión negativa, por lo que el resultado debería ser `-1`.

```
1 # Omitimos la segmentacion y codificacion. Tienen lugar
2 # en una funcion local preprocess
3 message_vector = preprocess("Este producto es muy muy malo")
4
5 sentiment = sentiment_cls.execute(vector=message_vector)
6 print(sentiment) # -1
```

Código 24: Ejemplo de ejecución del servicio de clasificación de sentimientos.

Si en lugar de un modelo de regresión logística quisiéramos utilizar Naive Bayes, debemos inicializar el servicio con `algorithm` a `'naive_bayes'`.

4.10. Reconocimiento de entidades nombradas

4.10.1. Introducción

En un mensaje en lenguaje natural podemos encontrar menciones textuales a entidades de interés para el dominio del agente conversacional. Estas menciones o designadores pueden hacer referencia a varias entidades distintas como consecuencia directa del concepto de mundos posibles de la lógica modal (LaPorte, 2016).

En el problema de extracción de entidades nombradas, el objetivo es extraer aquellas entidades que son referenciadas con un designador rígido, esto es, un designador que hace referencia a una misma entidad en todos los mundos posibles (Kripke, 2001, p. 48). Estas entidades reciben el nombre de entidades nombradas (Bedekar, Saini, y Zahoor, 2016, p. 3).

Ejemplo 4.7

Si bien el designador 'presidente de España en 2020' podría hacer referencia a distintas entidades en función del mundo posible del que estemos hablando, el designador rígido '3' siempre representa al número tres en todos ellos.

En nuestros conjuntos de datos NLU podemos especificar de forma opcional una lista de entidades presentes en el mensaje. En la representación YAML de nuestro *dataset* debemos especificar para cada entidad la posición del primer carácter del designador y su longitud en caracteres. No se puede especificar la posición a nivel de *token*, porque desconocemos de antemano el resultado de la segmentación.

```
1 - text: Hola ¿qué tal? Soy Manuel
2   sentiment: 0
3   entities:
4     - class: NAME
5       start: 19
6       len: 6
7       role: user_name
8       group: 1
```

Código 25: Ejemplo de definición de instancia NLU con entidades.

El símbolo utilizado para representar las entidades extraídas durante el proceso es `NamedEntity`. Esta clase almacena el tipo o categoría de una entidad nombrada, así como el contenido de su designador.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 class NamedEntity:
3     category: str
4     value: str
5     role: Optional[str] = None
6     group: Optional[str] = None
```

Código 26: Clase de las entidades nombradas.

En secciones posteriores introduciremos los conceptos de rol y grupo de una entidad nombrada. Son atributos opcionales, porque no siempre son necesarios o existen en un mensaje.

4.10.2. Problema de etiquetado

La naturaleza secuencial del lenguaje natural permite formular el problema del reconocimiento de entidades nombradas o NER como un problema de etiquetado (Akhundov, Trautmann, y Groh, 2018). Este problema consiste en la extracción y posterior clasificación de subcadenas que se corresponden con nombres propios (Jurafsky y Martin, 2020, p. 153).

En un problema de etiquetado de secuencia tenemos una secuencia de observación. En el problema de etiquetado de secuencia, tenemos una secuencia de observación $\langle x_1, \dots, x_n \rangle$ que debemos etiquetar con etiquetas de un conjunto Y , dando lugar a una secuencia $\langle y_1, \dots, y_n \rangle$ de n etiquetas, tal que $\forall i \in \mathbb{N}_n: y_i$ es la etiqueta de x_i (Nguyen y Guo, 2007).

En nuestro *framework*, resolvemos el problema de extracción y clasificación de entidades reduciéndolo a uno de etiquetado IOB2. Definimos un conjunto de etiquetas de clases \mathcal{C} a las que concatenamos un prefijo B-, I- u O-. Un prefijo B- denota que el *token* en cuestión es el primero de una entidad nombrada. El segundo prefijo, I-, denota que el *token* actual está

incluido en la misma entidad que el *token* anterior. Con la etiqueta O- especificamos que el *token* no forma parte de ninguna entidad (Krishnan y Ganapathy, 2005).

Las etiquetas son construidas de forma automáticamente por la librería a partir de la información disponible en nuestro conjunto de datos sobre la categoría (`class` en la representación YAML del *dataset*) y la información posicional a nivel de carácter.

A la hora de diseñar la *pipeline* de preprocesamiento se recomienda la aplicación de un tratamiento independiente a la información que consumirán los servicios de etiquetado. Si bien puede ser conveniente para los servicios de clasificación de intenciones o sentimientos la eliminación de palabras vacías o normalización de palabras, la información posicional sobre las entidades textuales puede verse afectada.

4.10.3. Normalización de menciones

A la hora de consumir servicios y almacenar información es necesario identificar todas las menciones textuales posibles a una entidad con un mismo identificador único.

Ejemplo 4.8

Las menciones 'NY' y 'Nueva York' hacen referencia a la ciudad de Nueva York. El servicio que consume esta entidad necesitará probablemente una representación canónica de la entidad para realizar una petición a una API, por ejemplo.

En ausencia de información contextual puede existir ambigüedad sobre la entidad o concepto al que se refiere una mención que puede hacer referencia a más de una entidad (Khalid, Jijkoun, y de Rijke, 2008; Stern, Sagot, y Béchet, 2012). Por ejemplo, en un artículo sobre política estadounidense la entidad textual 'Clinton' podría referirse a 'Bill Clinton' o 'Hillary Clinton'.

Algunos de los algoritmos propuestos en la literatura para efectuar la normalización recurren a la información disponible en Wikipedia sobre las entidades. Khalid y cols. (2008) propone un algoritmo que vincula menciones de entidades con páginas de entidades o redirección de Wikipedia. En el caso de existir una página de desambiguación selecciona la más referenciada. Otro algoritmo a destacar construye representaciones vectoriales de entidades de Wikipedia y las asocia con las de nuestras entidades nombradas Cucerzan (2007).

4.10.4. Clasificación de roles y agrupamiento de entidades

Además de pertenecer a una categoría, una entidad tiene otras propiedades entre las que destacamos el rol que tiene en el mensaje, y el grupo de entidades al que pertenece. Un grupo de entidades representa una entidad compuesta. Estos conceptos fueron introducidos por la librería Rasa para resolver problemas relacionados con *slot-filling*, esto es, la asignación de ciertos roles o *slots* a las entidades extraídas (*NLU Training Data*, 2021).

Ejemplo 4.9

En el mensaje siguiente podemos extraer dos entidades que representan ciudades:

Quiero reservar un vuelo de Málaga a Madrid.

Las entidades nombradas 'Málaga' y 'Madrid' tienen una misma categoría, pero sus roles en el mensaje son distintos. Tienen los roles de ciudad origen y ciudad destino, respectivamente.

Ejemplo 4.10

En el mensaje

Dos pizzas con pepperoni y una hamburguesa con bacon.

podemos extraer varias entidades que representan platos principales, ingredientes o cantidades. Distinguimos dos entidades compuestas en este mensaje: 'dos pizzas con pepperoni' y 'hamburguesa con bacon'. Cada entidad compuesta es un elemento de un pedido. 'Pepperoni' y 'bacon' adoptan el rol de ingrediente en sus respectivos grupos.

4.10.5. Uso de léxicos y expresiones regulares para la detección de entidades

Una de las acepciones del diccionario Collins para el término *lexicon* es "lista alfabética de palabras usadas en un lenguaje o en una materia en particular" (Collins, s.f.). En el caso que nos ocupa, un léxico o *lexicon* es una lista de entidades nombradas conocidas de antemano y posiblemente, normalizadas. También es posible la construcción de léxicos de palabras comunes para evitar la incorrecta extracción de entidades nombradas en contextos ambiguos como el inicio de un enunciado (Nadeau y Sekine, 2007, p. 10).

Existen varias técnicas para mejorar el rendimiento de los extractores de entidades basados en léxicos. Una de ellas consiste en normalizar y reducir las entidades a su raíz, permitiendo

un grado de flexibilidad en número y género a la hora de considerar si ciertos *tokens* deben ser extraídos como entidades según el léxico. También se puede recurrir a funciones de distancia de edición para extraer posibles entidades nombradas que contienen alguna errata (Nadeau y Sekine, 2007, pp. 10-11).

El uso de reglas y expresiones regulares también es posible para la extracción de entidades nombradas. La principal ventaja frente a los léxicos es que no tenemos que proporcionar una lista exhaustiva de las posibles entidades.

4.10.6. Modelos ocultos de Markov

Actualmente, el aprendizaje supervisado predomina en los sistemas extractores de entidades. Entre algunos de los algoritmos utilizados para estos sistemas se encuentran los modelos ocultos de Markov y los campos aleatorios condicionales o CRF (Nadeau y Sekine, 2007, p. 4). En algunos experimentos, la combinación de estos modelos en un CRF semi-Markov obtuvo mejores resultados que ambos por separado (Sarawagi y Cohen, 2004).

Las cadenas de Markov son secuencias de variables aleatorias que nos permiten modelar el modelo de transiciones de un sistema. La variable X_i de la cadena representará el estado del sistema en el instante i (Haight, 1981, pp. 91-92).

La actualización del estado puede tener en cuenta la evolución de estados por los que ha pasado el sistema. Decimos que una cadena de Markov de orden $n \in \mathbb{N}$, tiene una memoria finita limitada al estado actual últimos m estados del sistema. Si la cadena tiene la propiedad de Markov o lo que es lo mismo, es de primer orden, solo almacena el estado actual (Haight, 1981, p. 90).

En el contexto de un problema de etiquetado, el estado X_i en el instante i es la etiqueta asociada al *token* i ésimo. El espacio de estados o posibles valores de las variables aleatorias de la cadena coincide con el espacio de etiquetas inducido por el esquema de etiquetado elegido.

Dada una determinada secuencia de etiquetas, podemos computar la probabilidad de dicha secuencia recurriendo a una cadena de Markov. No obstante, en el problema de etiquetado no observamos secuencias de etiquetas sino de palabras o *tokens*. A partir de una secuencia de observaciones, debemos inferir la secuencia de estados del sistema que no es observable. Los modelos ocultos de Markov extienden el concepto de cadena de Markov para incluir un modelo de emisión de observaciones y, poder modelar así, este tipo de sistemas con un estado interno

oculto. Se supone que las observaciones solo dependen del estado en el que se generaron (Jurafsky y Martin, 2020, p. 156).

$$P(O_{t+1} | X_0, \dots, X_t, O_0, \dots, O_t) = P(O_{t+1} | O_t) \quad (7)$$

4.10.7. Modelos de máxima entropía de Markov

Una alternativa a los modelos ocultos de Markov son los modelos de Markov de entropía máxima o MEMM que tratan de solventar dos problemas que podemos encontrar al utilizar un HMM en tareas de procesamiento de lenguaje natural (McCallum, Freitag, y Pereira, 2000):

- **Representación pobre de las observaciones.** Las observaciones en un HMM se reducen al contenido textual del *token* y no se utiliza información lingüística o posicional de interés como la etiqueta POS de la palabra.
- **Enfoque inadecuado.** Recurrir a un modelo generativo para resolver un problema condicional no es adecuado. Los parámetros de un HMM se obtienen maximizando la verosimilitud de la secuencia de observación, cuando la tarea consiste en predecir la secuencia de etiquetas dado la secuencia de observaciones.

Un MEMM introduce cambios en los modelos utilizados para representar el sistema. En lugar de definir un modelo de emisión de observaciones y un modelo de transiciones, define un modelo observación-estado que modela la probabilidad de estar en un determinado estado en función del estado anterior y la observación emitida en el estado actual. Para cada posible estado $s \in S$, definimos una función de transición $P_s(s' | o) = P(s' | s, o)$ dada por un modelo exponencial ajustado por entropía máxima (McCallum y cols., 2000).

Para entrenar un MEMM, construimos n características f_1, \dots, f_n que dependen de la observación y estado actual. La distribución con entropía máxima que es consistente con las restricciones derivadas del conjunto de entrenamiento dadas nuestras características es exponencial (McCallum y cols., 2000).

A continuación, exponemos una lista no exhaustiva de propiedades que pueden ser de interés a la hora de reconocer entidades (Nadeau y Sekine, 2007, pp. 8-9):

- 1 si el *token* empieza o no por mayúsculas. En otro caso, 0.
- 1 si el *token* es un número o no. En otro caso, 0.

- 1 si el *token* tiene un sufijo determinado. En otro caso, 0.
- Número de caracteres del *token*.
- La identidad, esto es, el *token* en sí.
- El *token* en minúsculas. Por ejemplo, 'ana' si el *token* es 'Ana'.
- El *token* en mayúsculas. Por ejemplo, 'ANA' si el *token* es 'Ana'.
- La etiqueta POS del *token*.

4.10.8. Campos aleatorios condicionales (CRF)

Tal y como mencionamos anteriormente los modelos de máxima entropía de Markov (MEMM) son modelos no generativos de estados finitos que modelan la transición al siguiente estado a través de un modelo exponencial (McCallum y cols., 2000; Lafferty, McCallum, y Pereira, 2001). Si un MEMM tiene $|S|$ estados, se entrenarán de forma independiente $|S|$ funciones de transición $P_{s'}(s | o)$ donde s' es el estado actual, s es un estado al que podríamos transitar, y o la observación que determinará la probabilidad de que transitemos a s desde s' (McCallum y cols., 2000).

Si bien las funciones de transición $P_{s'}(s | o)$ nos permiten conocer la probabilidad de transitar a un estado s dado un par estado-observación (s', o) , la normalización local o a nivel de función de transición conlleva la pérdida de información sobre la incertidumbre o confianza del modelo acerca de esa probabilidad (Lafferty y cols., 2001). Si durante el entrenamiento el modelo ha observado solo un par de veces la observación o en el estado s' , la confianza sobre las probabilidades estimadas de transición dado el par (s', o) será inferior en magnitud.

Los modelos probabilísticos o no de etiquetado de secuencias que entrenen de forma independiente modelos de transición para cada estado son susceptibles del fenómeno conocido como *label bias*, por el cual, en el peor de los casos, una observación es completamente ignorada en la predicción del siguiente estado (Le-Hong, Phan, y Tran, 2013). Este fenómeno tiene su origen en la normalización local de la función de transición. La puntuación o probabilidad de una transición a un estado s' se distribuye entre todos sus posibles sucesores en función de la observación recibida en s' . Al no disponer de información sobre la incertidumbre de las pro-

babilidades, la observación determina cómo se hace la distribución, pero no puede penalizar una elevada incertidumbre (Lafferty y cols., 2001).

Supongamos que dada una observación o en un estado s' , la probabilidad de transitar a s es 1. Si la certeza de la transición se debe a un número reducido de observaciones durante el entrenamiento, la incertidumbre sobre la decisión de transitar a s será elevada, pero esta información ha sido descartada al normalizar localmente. Por lo anterior, la normalización a nivel de estado beneficiará a aquellas transiciones que no reduzcan apenas la probabilidad total de la secuencia de estados.

Los campos aleatorios condicionales son modelos gráficos no dirigidos similares a los modelos de máxima entropía de Markov con la principal diferencia de que la normalización no es local, sino global, esto es, se preserva la información sobre la incertidumbre de las probabilidades de las transiciones.

4.11. Servicio de etiquetado de secuencias

En la sección dedicada a la introducción del problema de extraer entidades de un texto mencionamos que podemos reducir este problema al de etiquetado de secuencias. En nuestra librería proporcionamos un servicio genérico `IOBTagger` de etiquetado de *tokens* basado en campos aleatorios condicionales. Este servicio debe ser entrenado con una lista de vectores de características de *tokens* y una lista de etiquetas. La primera lista puede computarse recurriendo a un servicio de construcción de características.

```
1 ner_tagger = IOBTagger()
2
3 ner_tagger.train(
4     features=features, # lista de listas de diccionarios (1 dict por token, 1 lista por mensaje),
5     tokens=tokens,
6     tags=Y # lista de diccionarios que representan entidades (vease el formato YAML)
7 )
8
9 x = [{...}, ..., {...}] # 1 dict de características por token
10 y = ner_tagger.execute(features=x)
```

Código 27: Ejemplo de inicialización, entrenamiento y ejecución del servicio genérico de etiquetado de *tokens*.

4.12. Servicio de clasificación de roles y agrupamiento de entidades

El servicio `MultiCRFSlotFiller` asigna roles y grupos a las entidades nombradas textuales extraídas por uno o más extractores en etapas anteriores. Su implementación está basada en el uso de dos campos aleatorios condicionales reduciendo el problema de asignar un rol y un grupo al de etiquetar una secuencia con dos etiquetas por elemento. Un primer CRF utiliza el mismo vector de características de *tokens* utilizado por los servicios de extracción, pero incluye una nueva característica con la etiqueta IOB2 de cada *token*. La información del etiquetado de anteriores se tiene en cuenta en el proceso de etiquetar cada *token* con un rol. Posteriormente, la información sobre la etiqueta del rol y de entidad se incluye como características al entrenar el CRF que asigna grupos.

```
1 role_group_tagger = MultiCRFSlotFiller()
2
3 role_group_tagger.train(
4     features=X_f, # lista de listas de diccionarios (1 dict por token, 1 lista por mensaje)
5     tokens=tokens,
6     ner_tags=X_ner_tags # lista de listas de etiquetas de tipo string
7     role_tags=y_role_tags, # lista de listas de etiquetas de tipo string
8     group_tags=y_groups_tags # lista de listas de etiquetas de tipo string
9 )
10
11 x = [{...}, ..., {...}] # 1 dict de características por token
12 tags = [...] # 1 etiqueta IOB2 de entidad por cada token
13 y = role_group_tagger.execute(features=x, ner_tags=tags)
```

Código 28: Ejemplo de inicialización, entrenamiento y ejecución del servicio de etiquetado de roles y grupos de entidades.

4.13. Servicio de extracción de entidades nombradas

El servicio `NamedEntityExtractor` consume listas de etiquetas de presencia o ausencia de entidades textuales, de roles y grupos, y proporciona una lista de entidades `NamedEntity`. En otras palabras, construye una lista de entidades para el proceso *core* a partir de entidades textuales.

```

1  from smartbot.services.preprocessing.segmentation import WhitespaceTokenizer
2
3  tokenizer = WhitespaceTokenizer()
4  ner = NamedEntityExtractor()
5
6  tokens = tokenizer.execute(text="Quiero ir de Nueva York a California")
7  entities = ner.execute(
8      tokens=tokens,
9      ner_tags=["O", "O", "O", "B-LOC", "I-LOC", "O", "B-LOC"],
10     role_tags=["O", "O", "O", "B-from", "I-from", "O", "B-to"],
11     group_tags=["O", "O", "O", "1", "1", "O", "1"]
12 )
13
14 print(entities)
15 # [
16 #   NamedEntity(
17 #     category='LOC',
18 #     value='Nueva York',
19 #     role='from',
20 #     group='1'
21 #   ),
22 #   NamedEntity(
23 #     category='LOC',
24 #     value='California',
25 #     role='to',
26 #     group='1'
27 #   )
28 # ]

```

Código 29: Ejemplo de inicialización y ejecución del servicio de extracción de entidades nombradas a partir de secuencias de etiquetas.

4.14. Servicios de normalización de entidades

4.14.1. Corrección de posibles erratas

En aquellos casos en los que sea necesario realizar una normalización del valor de una entidad nombrada disponemos de servicios que intentan obtener una representación canóni-

ca de las entidades introducidas por el usuario. El servicio `NamedEntitiesSpellChecker` es una implementación de un corrector simple de cadenas a partir de un vocabulario de cadenas válidas. En el caso que nos ocupa, este vocabulario será una lista de entidades nombradas válidas. Por ejemplo, el asistente de un restaurante italiano podría tener un vocabulario con los platos, bebidas o ingredientes que se encuentran en el menú. Este servicio no es *case sensitive*, por lo que cadenas que solo difieran en la capitalización como 'Pizza' y 'pizza' serán tratadas como iguales. Por otro lado, debemos decir que este servicio aplica a lo sumo dos correcciones al valor de una entidad, por lo que si una cadena difiere en más de 2 caracteres de cualquier cadena del vocabulario, no se aplicará corrección y se notificará que no ha sido corregida con éxito.

```
1 from smartbot.services.nlu.named_entities.normalization.spell_checking import \
2     NamedEntitiesSpellChecker
3 from smartbot.services.nlu.named_entities.symbols import NamedEntity
4
5 spell_checker = NamedEntitiesSpellChecker(
6     supported_named_entities=["patatas", "peces", "limón"]
7 )
8
9 corrected_entities, status = spell_checker.execute(named_entities=[
10     NamedEntity(value="patata", ...),
11     NamedEntity(value="peces", ...),
12     NamedEntity(value="limon", ...),
13     NamedEntity(value="pez", ...),
14 ])
15
16 print(corrected_entities[0], status[0]) # NamedEntity(value="patatas", ...), true
17 print(corrected_entities[1], status[1]) # NamedEntity(value="peces", ...), true
18 print(corrected_entities[2], status[2]) # NamedEntity(value="limón", ...), true
19 print(corrected_entities[3], status[3]) # NamedEntity(value="pez", ...), false
```

Código 30: Ejemplo de inicialización y ejecución del servicio de corrección de erratas en entidades.

4.14.2. Sinónimos

Un paso recomendado posterior a la corrección de posibles erratas es la obtención de la forma canónica de la entidad nombrada. Si nuestro agente ha de consumir servicios de lógica de negocio, es posible que sea necesario hacer comprobaciones u operaciones sobre las entidades nombradas extraídas, lo cual hace necesario una normalización que reduzca todo el espacio de entidades nombradas soportadas a unas cuantas formas canónicas.

Supongamos que un *bot* de un restaurante debe comprobar si un ítem está en la carta. Un usuario puede referirse a un mismo ítem de varias formas, por lo que es necesario asociar a una lista de nombres alternativos un único representante. Esto permite simplificar la lógica de negocio.

El servicio que proporciona nuestro *framework* para este tipo de normalización se denomina `SynonymsNormalizer`. Su inicialización es sencilla, pues solo necesita un diccionario donde las claves serán las formas canónicas y los valores las listas de nombres alternativos.

```
1 from smartbot.services.nlu.named_entities.normalization.spell_checking import \
2     NamedEntitiesSpellChecker
3 from smartbot.services.nlu.named_entities.symbols import NamedEntity
4
5 syn = SynonymsNormalizer(classes={
6     "queso": ["cheese"],
7     "hamburguesa": ["burger"],
8     "2": ["dos"]
9 })
10
11 syn.execute(named_entities=[
12     NamedEntity(category='NUMBER', value='dos', role='quantity', group='1'),
13     NamedEntity(category='FOOD', value='burger', role='meal', group='1')
14 ])
15
16 # [NamedEntity(category='NUMBER', value='2', role='quantity', group='1'),
17 # NamedEntity(category='FOOD', value='hamburguesa', role='meal', group='1')]
```

Código 31: Ejemplo de inicialización y ejecución del servicio de normalización de entidades.

5

Diseño del proceso *core*

El proceso de comprensión de lenguaje natural genera una representación estructurada del mensaje del usuario, que proporciona, entre otros datos, una intención. Cada intervención en un diálogo tiene una intención que es transmitida con el objetivo de inducir una cierta conducta en el receptor (Brady y Berwick, 1989, p. 108). Un sistema de gestión de diálogo procesa esta información y, teniendo en cuenta la información contextual que almacena y fuentes externas de conocimiento, determina el contenido del mensaje que se debe transmitir (Clark y cols., 2013, pp. 443-444).

En este capítulo expondremos modelos y técnicas para implementar lo que podría considerarse el *core* de un agente conversacional, su sistema de gestión de diálogo.

5.1. Autómatas finitos

Un sistema de gestión de diálogo puede implementarse como un autómata finito en el que los estados son peticiones o conjuntos de peticiones para el sistema de generación de lenguaje natural, y las transiciones tienen lugar al producirse determinadas respuestas del usuario (Clark y cols., 2013, p. 444).

Dos de las principales ventajas de este modelo son la simplicidad y determinismo del *core* de nuestro agente conversacional. Suponiendo que el proceso de comprensión logra capturar correctamente la información necesaria, tendremos la certeza de que el agente va a responder de una cierta forma.

Debemos mencionar que uno de los puntos débiles de la implementación basada en autómatas finitos es la suposición de que cada intervención del usuario es una respuesta a la intervención anterior del agente artificial. Aunque esta suposición simplifica aparentemente

el modelo, deberemos introducir un tratamiento adecuado a aquellas respuestas que introduzcan un cambio de t3pico o carezcan de sentido dada la intervenci3n anterior del agente artificial. Una posible decisi3n de implementaci3n podr3a ser ignorar estos mensajes o solicitar al usuario que responda de otra forma haciendo la suposici3n de que no se va a cambiar el t3pico de la conversaci3n.

Podemos recurrir a este modelo para implementar agentes conversacionales con una l3gica sencilla como los agentes FAQ. Un ejemplo de aplicaci3n ser3a la implementaci3n de un asistente que proporciona informaci3n sobre s3ntomas o medidas preventivas de una enfermedad determinada en una epidemia.

5.2. Di3logo guiado por *frames*

Un *frame* representa el estado o contexto de la conversaci3n en forma de un formulario con varios *slots* que tienen un valor actual. Cada *slot* puede tener asociado un conjunto de peticiones al sistema NLG (Clark y cols., 2013, pp. 445).

Slot	Valor	Peticiones posibles
NOMBRE_CLIENTE	Luis L3pez Reviriego	¿Cu3l es su nombre completo?
EDAD	null	¿Cu3ntos a3os tiene?, ¿Cu3l es su edad?

En el ejemplo anterior, el campo EDAD no ha sido determinado todav3a y el gestor de di3logo podr3a elegir una de las dos peticiones NLG para obtener su valor en el siguiente turno de la conversaci3n.

Un algoritmo de control inspecciona el *frame* actual y determina siguiendo una pol3tica el siguiente *slot* que debemos conocer (Clark y cols., 2013, p. 445). Dado que un usuario puede proporcionar el valor de *slots* en un mismo mensaje, el flujo del di3logo no est3 determinado en tiempo de dise3o como en el caso de los aut3matas (Clark y cols., 2013, p. 446). Algunos ejemplos de pol3ticas para el algoritmo de control son las siguientes:

- Elegir el primer *slot* que tenga un valor desconocido.
- Asignar prioridades a cada *slot* y elegir aquel que tenga una mayor prioridad y su valor sea desconocido.

- Crear una serie de reglas que tengan en cuenta si uno o varios *slots* tienen un valor conocido o no. Un ejemplo de regla sería 'Si USER_NAME no es null y AGE es null, pregunta por USER_NAME. En otro caso, pregunta por AGE'.

5.3. Gestión de diálogo basado en transformadores

Mientras que todos los modelos anteriores podrían clasificarse como soluciones posibles al problema de la gestión del diálogo desde un punto de vista de inteligencia artificial simbólica, existen modelos estadísticos basados en transformadores que permiten resolver este problema haciendo uso de aprendizaje automático.

5.3.1. Tópicos de una conversación

Uno de los conceptos sobre los que se asienta la arquitectura TED propuesta por Rasa es el segmento de discurso. Definimos un segmento de discurso como un conjunto de mensajes no necesariamente consecutivos en el flujo de mensajes que forman parte de un mismo tópico y que se responden unos a otros. En una conversación, por ejemplo, un agente artificial puede hacer una pregunta y el usuario puede responder con otra pregunta sobre otro tema o tópico. El agente respondería al usuario, pero volvería, probablemente, a volver a repetir la pregunta volviendo al tópico anterior. En este fragmento de la conversación habría presente dos segmentos de discurso o tópicos (Vlasov, Mosig, y Nichol, 2019).

5.3.2. Política de diálogo TED

La hipótesis de Rasa es que un transformador, a través de su mecanismo de atención, puede ser capaz de aprender a separar segmentos del discurso entrelazados. Lo anterior supondría que el agente conversacional lidiaría correctamente con conversaciones en las que se producen subdiálogos que introducen un cambio temporal del tópico para volver nuevamente al que se estaba tratando anteriormente (Vlasov y cols., 2019).

El primer aspecto a considerar es la creación de características que describan de forma suficiente el contexto del diálogo en un instante de tiempo dado. En el *framework* de Rasa, debemos proporcionar de antemano una lista finita de intenciones, entidades, *slots* y acciones del asistente. Esto permite aplicar una codificación One-Hot a esta información obtenida a

través del proceso de comprensión. En el caso de los *slots*, no se tiene en cuenta el valor de los mismos, sino su estado, conocido o no. El vector contextual es el resultado de concatenar la codificación de la intención, entidades, *slots* y la acción anterior del asistente.

Tras recibir como entrada un vector contextual, el transformador generará la representación One-Hot correspondiente a la acción a ejecutar por el asistente. Dado que un transformador bidireccional tiene en cuenta el futuro durante el entrenamiento y en el caso que nos ocupa, no podemos disponer de esa información en el momento de la selección de la acción, se recurre a un transformador unidireccional.

El mecanismo de atención permite al modelo acceder a aquellas partes de interés del histórico del diálogo, lo cual permite ignorar o no ciertos turnos en función del contexto.

5.4. Lógica del agente conversacional en nuestro *framework*

Tras haber introducido en el resto de capítulos algoritmos y técnicas para el proceso de preprocesamiento y comprensión de lenguaje natural, hemos dedicado este capítulo al proceso *core* de los agentes conversacionales. Dos de las propiedades más destacables de este proceso son la presencia de un mecanismo de memoria que preserva parte de la información generada durante el análisis de un estímulo antiguo, y la función cognitiva de elección de la mejor reacción a un estímulo reciente en el contexto actual.

5.4.1. Estímulos

En general, denominamos estímulo a todo mensaje recibido por nuestro agente desde el entorno en el que opera. En el caso del proceso *core* un estímulo es un mensaje externo ya procesado por el proceso de comprensión. La clase base de estos mensajes es `BaseIncomingMessage` que incluye tres propiedades `text`, `channel_id` y `timestamp`. La primera de ellas es el contenido textual a procesar por el proceso de comprensión, la segunda es un identificador del canal por el que fue recibido el mensaje y la tercera y última es la fecha en la que fue recibido o enviado.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 class BaseIncomingMessage:
3     text: str
```

```
4 channel_id: str
5 timestamp: datetime
```

Código 32: Clase base de los mensajes externos que espera recibir el agente.

El identificador del canal nos puede permitir identificar la plataforma que utilizó el usuario para enviar el mensaje, podemos determinar la identidad del emisor, etc. Con esta información el agente puede enviar sus mensajes a través de ese mismo canal u optar por otro que tenga unas propiedades más ideales para la transmisión que quiere hacer. El *timestamp* podría ser un parámetro opcional, pero conocer su valor puede permitir al agente conocer el tiempo que ha esperado el usuario a una respuesta. Supongamos, por ejemplo, que el servicio del agente no está operativo, pero se almacenan los mensajes enviados por el usuario. Al volver a funcionar el agente, podría disculparse en aquellos casos en los que ha pasado más de 20 segundos sin responder.

El resultado del procesamiento de los mensajes externos son mensajes *core* de la clase `CoreMessage`. Estos mensajes contienen la información extraída por el proceso de comprensión del mensaje original.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 class BaseCoreMessage:
3     timestamp: datetime
4
5
6 @dataclasses.dataclass(init=True, repr=True)
7 class CoreMessage(BaseCoreMessage):
8     original_message: BaseIncomingMessage
9     intent: str
10    sentiment: Optional[float]
11    entities: Optional[List[NamedEntity]]
```

Código 33: Clases de los mensajes que espera recibir el proceso *core*.

Se proporciona un servicio `CoreMessageBuilder` que puede ser consumido en el proceso de comprensión para construir el mensaje *core* a partir de la información extraída durante la

ejecución de dicho proceso. El servicio debe recibir una intención, sentimiento si aplica, una lista de entidades y el mensaje original.

Existe la opción de sobrescribir desde `CoreMessageBuilder` la intención de un mensaje si se inicializa con `overrides_intent`. Solo es posible sobrescribir este valor si el mensaje es una entidad en sí, esto es, el usuario ha emitido un mensaje que no contiene nada más que la entidad que ha mencionado. A través del parámetro `entity_intent`, podemos especificar el nombre de la intención de los mensajes que se reducen a una entidad. Por defecto, su valor es `'entity'`.

5.4.2. Estado interno

El estado interno es un diccionario extendido que es una instancia de `BotInternalState`. Denominaremos *slot* a los pares clave-valor que lo constituyen. A continuación, describimos los *slots* gestionados de forma automática por el agente:

- **El último estímulo recibido:** El atributo opcional `latest_message` almacena el último mensaje de tipo `BaseIncomingMessage` recibido en esta conversación. Durante la selección de la reacción, este atributo almacena el estímulo actual al que debemos reaccionar.
- **Flags de inicio de conversación:** Mientras que la propiedad `conversation_started` es `True` si y solo si el agente no ha recibido ningún mensaje, la propiedad `is_first_message` es `True` desde que se ha recibido el primer mensaje hasta que se recibe el siguiente.
- **La definición del flujo activo actual:** `active_flow` almacena la definición del flujo activo en el agente. En el caso de que no haya ninguno activo, su valor será nulo. Introduciremos en una sección posterior la utilidad de los flujos en el agente.
- **Histórico de la conversación:** Al crear un estado debemos especificar la capacidad máxima de este histórico. Un histórico es una lista de pares estímulo-reacción que representa la principal fuente de información sobre el pasado reciente de la conversación. Esta información está disponible a través del atributo `history`. Se proporcionan los métodos `add_turn` y `set_reaction_for_last_turn` para añadir una nueva entrada y actualizarla al computar la reacción más adelante, respectivamente. Esta estructura será

necesaria para la implementación de los modelos de reacción dependientes del pasado de la conversación que serán expuestos en secciones posteriores.

- **Nuevos *slots* introducidos en la memoria al procesar el estímulo actual:** `was_set` es un conjunto de claves que fueron introducidas en el diccionario tras aplicar *slot filling* automático en *core*.
- ***Slots* que el agente espera que sean introducidos en la memoria al procesar el siguiente estímulo:** `expected_slots` es un conjunto de claves que deberían ser introducidas al procesar el siguiente mensaje, puesto que representan datos que han sido solicitados al usuario. Tras aplicar el *slot filling* automático del siguiente mensaje, el agente puede comparar `was_set` y `expected_slots` para determinar si el usuario ha ignorado completamente la petición o ha aportado un subconjunto de la información solicitada.

Debido a que el estado se trata de un diccionario extendido, el desarrollador puede almacenar nuevos *slots* en la memoria como si se tratara de un diccionario convencional. Si se desea reiniciar el estado de la memoria, se puede recurrir al método `reset` de `BotInternalState` que eliminará completamente los *slots* de la memoria y recreará los gestionados automáticamente por el agente.

5.4.3. Preprocesamiento en *core*

Uno de los primeros servicios *core* que introducimos es `CorePreprocessor`. Dado el estado anterior `state` y un mensaje `user_message` de tipo `BaseIncomingMessage`, este servicio actualiza la memoria y da lugar a un nuevo estado de la misma. Si se trata del primer mensaje, el preprocesador no recibirá un estado anterior `state` al ser ejecutado, por lo que creará una nueva instancia de `BotInternalState`.

Otras de las tareas efectuadas por este servicio de preprocesamiento es la actualización de un histórico de turnos de diálogo que puede ser utilizado por algunos de los modelos de comportamiento para tener información sobre el contexto. En etapas posteriores del procesamiento del mensaje en *core*, se determina la reacción y se actualiza este campo en el histórico recurriendo a `set_reaction_for_last_turn` de `BotInternalState`.

5.4.4. *Slot filling*

Los estímulos de mensajes de los usuarios pueden contener varios símbolos conocidos como entidades. Es necesario introducir un mecanismo que automatice la introducción de algunos de estos símbolos en el estado interno del agente conversacional.

Distinguimos entre dos tipos de *slots*: globales y de flujo. Los primeros pueden ser rellenados en cualquier momento por el agente, mientras que los segundos solo son rellenados al haber un flujo activo. Otra diferencia fundamental entre estos tipos es que un *slot* de flujo no será rellenado, en principio, dos o más veces, mientras que uno global sí puede ser sobrescrito varias veces.

Para definir un *slot* global debemos crear una instancia `BotSlotDefinition` que defina algunas de las propiedades de las entidades que serían candidatas para rellenar un *slot* `name`. Podemos restringir el *slot filling* automático a mensajes con una intención presente en la lista `in_messages_with_intents` y no incluida en `exclude_messages_with_intents`. Además de estas condiciones, se puede añadir una condición adicional a través de `enabled`. Esta atributo es una función *lambda* que recibe como argumento un estado y devuelve `False` si y solo si el *slot filling* está activado para este *slot* en este estado.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 class BotSlotDefinition:
3     name: str
4     category: str
5     role: str = None
6     group: str = None
7     in_messages_with_intents: List[str] = None
8     exclude_messages_with_intents: List[str] = None
9     reaction_on_invalid_value: List[str] = None
10    validate: Callable = lambda state, slot_def, value: True
11    transform: Callable = lambda state, slot_def, value: value
12    enabled: Callable = lambda state: True
```

Código 34: Clases de los *slots* globales.

Para definir un *slot* de flujo disponemos de una clase `FlowBotSlotDefinition` que es similar a la anterior pero incluye un atributo `reaction`, una valor de prioridad u orden y un

mecanismo de validación y activación. Estos *slots* forman parte de un flujo y, por ende, un modelo de selección de reacción basado en flujo que expondremos más adelante necesitará información sobre cómo reaccionar ante la necesidad de conocer el valor de un *slot*. La prioridad permite a este modelo de selección elegir el siguiente *slot* desconocido a rellenar, definiendo un orden coherente a la hora de solicitar información. El atributo `enabled` permite establecer una regla para determinar cuando el *slot* puede ser solicitado o no. Por ejemplo, si el usuario es menor de edad, puede ser necesario preguntar un *slot* que no es necesario si el usuario es mayor de edad.

La validación de *slots* de flujo es posible a través del atributo `validate`. El servicio de *slot filling* comprobará si el valor aportado para el *slot* es válido o no haciendo uso de esta función. En el caso de que el valor no sea válido, este servicio almacenará en el estado interno del agente una instancia de `SlotValidationError` que representa un error de validación. Además de esto, se ejecutará la reacción `reaction_on_invalid_value` en el servicio de lógica del agente.

El servicio de *slot filling* no intentará rellenar aquellos *slots* globales que no estén disponibles para el estado actual (i.e. para el *slot*, `enabled(state) == False`). Nótese que el estado de disponibilidad de un *slot* no debe depender de propiedades del estado interno que pueden ser actualizadas durante el proceso de *slot filling*.

Al igual que con los *slots* globales, la validación de *slots* de flujo también es posible a través del atributo `validate`.

```
1 @dataclasses.dataclass(init=True)
2 class SlotValidationError:
3     slot_def: FlowBotSlotDefinition
4     value: str
```

Código 35: Clase de los errores de validación.

En el siguiente ejemplo, definimos dos *slots* de un mismo flujo: 'to' y 'from'. Solo se tienen en cuenta durante el *slot filling* si los mensajes tienen una intención 'entity' o 'book_flight', por lo que si el usuario envía un mensaje con una intención 'bot_challenge', por ejemplo, que contiene una entidad que tiene las mismas propiedades que la que exige el slot 'to', no se vinculará esa entidad a este *slot*. En el caso de que un modelo de selección de reacción tuviera que elegir entre estos dos *slots* vacíos, daría prioridad a rellenar el slot 'from', porque tiene una

mayor prioridad.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 flow_slot_to = FlowBotSlotDefinition(
3     name="to",
4     in_messages_with_intents=["entity", "book_flight"],
5     role="to",
6     category="LOCATION",
7     reaction=["action_ask_destination"],
8     priority=9
9 )
10
11 flow_slot_from = FlowBotSlotDefinition(
12     name="from",
13     in_messages_with_intents=["entity", "book_flight"],
14     role="from",
15     category="LOCATION",
16     reaction=["action_ask_origin"],
17     priority=10
18 )
```

Código 36: Ejemplo de definición de dos *slots* de flujo.

El servicio `CoreSlotFiller` lleva a cabo un proceso de *slot filling* en *core*. En su inicialización podemos especificar una lista de *slots* globales de `BotSlotDefinition`. Al ejecutar este servicio, debemos proporcionarle el estado interno del agente, puesto que incluye toda la información necesaria para rellenar los *slots* globales y de flujo.

Si no hay ningún flujo activo en el estado, este servicio intentará rellenar aquellos *slots* globales que deban considerarse para el mensaje actual. Independientemente de si su valor ya está presente en memoria o no, se intentan rellenar.

Si existe un flujo activo, este servicio dará prioridad a los *slots* vacíos de flujo. Por lo anterior, si una entidad podría rellenar un *slot* global y otro de flujo, se rellenará el segundo. Aunque el agente haya especificado una lista de *slots* cuyos valores espera que el usuario proporcione, este servicio no se limita a rellenar los *slots* vacíos de flujo especificados en el atributo `expected_slots` de la memoria, porque el usuario podría proporcionar más información de la solicitada y no debe ser descartada. Por ejemplo, si el agente preguntó al usuario la ciudad

de origen y el mensaje contiene una entidad para la ciudad de origen y otra para la de destino, no se descarta esta última entidad por no haber sido solicitada todavía.

Existe un caso especial de *slot* de flujo en el que el usuario proporciona su valor de forma implícita. Por ejemplo, una respuesta afirmativa o negativa a una pregunta en un flujo es una forma implícita de rellenar un *slot* booleano. Para implementar este tipo de *slots* proporcionamos la clase `FlowConfirmationBotSlotDefinition`.

```
1 class FlowConfirmationBotSlotFillingMode(Enum):
2     DEFAULT = 0
3     COMPLETED_IF_ACCEPTED = 1
4     COMPLETED_IF_REJECTED = 2
5
6
7 @dataclasses.dataclass(init=True, repr=True, eq=True)
8 class FlowConfirmationBotSlotDefinition(FlowBotSlotBaseDefinition):
9     first_reaction: List[str]
10    reaction_accept: List[str]
11    reaction_reject: List[str]
12    intents_accept: List[str]
13    intents_reject: List[str]
14    enabled: Callable = lambda state: True
15    filling_mode: FlowConfirmationBotSlotFillingMode
```

Código 37: Clase de los *slots* de flujo de confirmación.

Un *slot* de flujo de confirmación se rellena al recibirse mensajes con ciertos tipos de intenciones. Si la intención del mensaje se encuentra presente en la lista `intents_accept`, se rellenará a `True` este *slot*. Si, por el contrario, la intención está presente en `intents_reject`, se rellenará a `False`.

Al rellenarse un *slot* de confirmación tenemos la opción de ejecutar o no una reacción. Se puede proporcionar una reacción para el caso de respuesta positiva y otra para el caso de respuesta negativa. Supongamos que el *bot* pregunta al usuario si una determinada fecha es adecuada para una reunión. Si el usuario responde negativamente, el *slot* se rellenaría y el flujo podría terminar sin un resultado satisfactorio. Para evitar esto es necesario introducir dos elementos. El primero de ellos es definir en que casos se considera que el *slot* de confirmación

se ha rellenado con un valor aceptable. El segundo elemento son las reacciones dependientes de la respuesta que acabamos de introducir.

El atributo `filling_mode` puede tener tres valores. Si se mantiene el valor por defecto, el *slot* de confirmación se considerará relleno tanto con una respuesta positiva como negativa. Los otros dos valores posibles solo consideran que el usuario ha rellenado el *slot* si se proporciona una respuesta negativa o positiva. No obstante, en todos los casos el valor booleano del *slot* se rellena al recibir una respuesta. El atributo `filling_mode` solo afecta al criterio del modelo de comportamiento basado en flujos para determinar si el *slot* se ha rellenado y hay que terminar el flujo, solicitar otro *slot* o dejar en pendiente el de confirmación.

Finalmente, hemos de comentar que el atributo `first_reaction` es equivalente al atributo `reaction` de `FlowBotSlotDefinition` y solo se ejecuta la primera vez.

5.4.5. Servicio de lógica del agente

El servicio `BotLogicService` es el responsable de generar y ejecutar una reacción en un estado `state`. En la inicialización del servicio, debemos incluir aquellos modelos de comportamiento que queremos utilizar en nuestro agente. Un modelo de comportamiento propone una reacción (i.e. una lista de acciones) para un estímulo. `BotLogicService` puede consultar a varios modelos y elegir aquel que haya sido generado por el modelo que presente mayor prioridad. Como norma general, una elevada prioridad denota una elevada certeza de que la reacción elegida por el modelo es adecuada.

La clase base de los modelos de comportamiento es `BaseBehaviorModel` cuya definición es la siguiente:

```
1 class BaseBehaviorModel(ABC):
2     def __init__(self, **kwargs):
3         self.priority = kwargs.get("priority")
4         if self.priority is None:
5             self.priority = DEFAULT_PRIORITY
6
7     @abstractmethod
8     def compute_reaction(self, state: BotInternalState):
9         pass
```

Código 38: Clase base de los modelos de comportamiento.

Además de inicializar este servicio con los modelos de comportamiento `recall_model`, `rules_model`, `flow_model` o `keras_model`, es necesario proveer al servicio de un diccionario de acciones `actions` en el que las claves serán los identificadores de las acciones y los valores referencias a funciones. Internamente, el servicio crea e inicializa una instancia del servicio de acciones `ActionsService` con este diccionario. Este último servicio será introducido en la última sección de este capítulo. Asimismo, se recomienda especificar una acción `action_fallback` que será seleccionada cuando ningún modelo de comportamiento sea capaz de elegir una reacción.

Los modelos de comportamiento ofrecido por nuestro *framework* serán expuestos en detalle en las siguientes secciones. Dedicaremos el resto de esta sección a exponer el procedimiento de elección de reacción.

Al ser ejecutado el servicio `BotLogicService` con un estado interno `state`, la primera comprobación que hace el agente es determinar si hay un flujo activo. En el caso de que no haya ninguno activo, se computa una reacción para el estado actual por cada modelo de comportamiento a excepción del modelo basado en flujo. Cada reacción computada tiene un peso que coincide con la prioridad del modelo que la ha generado. Si existe al menos una reacción candidata, se selecciona la que tiene asociada un mayor peso o prioridad. En el caso de que ningún modelo haya generado ninguna reacción, se selecciona `action_fallback` si se ha especificado. En otro caso, no hay reacción alguna por parte del agente al estímulo.

Si en la primera comprobación se detecta que hay un flujo activo, se comprueba si el agente había solicitado *slots* en el paso anterior. Si fueron solicitados *slots* y el usuario no proporcionó esa información, el agente debe recurrir al resto de modelos para actuar ante este *unhappy path*. Si, por el contrario, fue proporcionado al menos uno de los que pidió en el turno anterior, se computa una nueva reacción con el modelo basado en flujo. Asimismo, se verifica si alguno de los *slots* proporcionados era de confirmación y, en caso afirmativo, se ejecuta la reacción correspondiente. Si no se genera ninguna reacción, ya se han rellenado todos los *slots* del flujo y debe finalizarse. Tras ser finalizado se recurre a los otros modelos de comportamiento para actuar ante la finalización del flujo.

5.4.6. Modelo de comportamiento basado en recuerdos

Este modelo se basa en la idea de que ante un estímulo podemos actuar de una forma que ya funcionó en el pasado. Dado un estado interno, se comprueba si el contexto, esto es, las intervenciones anteriores y la actual presentan características similares o idénticas a un contexto pasado. Si es posible encontrar una experiencia que cumple con estas características, se elige la reacción del último turno de la misma.

Una experiencia o fragmento de diálogo `DialogueFragment` es una lista de turnos de la clase `DialogueTurn`. Un turno es un par estímulo-reacción donde el estímulo está caracterizado por su intención, sentimiento, entidades y el nombre de los *slots* que fueron rellenados. A excepción de la intención, el resto de campos son opcionales. En el caso de las entidades, se recurre a una clase `DialogueEntity` para poder describirlas parcialmente. Los campos `value`, `category`, `role` y `group` de `DialogueEntity` son opcionales, pero al menos uno de ellos debería ser especificado.

La implementación de este modelo comportamiento es la clase `RecallBasedBehaviorModel`. No requiere entrenamiento, puesto que este modelo no aprende del conjunto de experiencias pasadas o lista de `DialogueFragment` que le proporcionamos, sino que efectúa una búsqueda en la misma cuando es necesario. El servicio obtiene el contexto de la conversación a partir del atributo `history`. El formato de este contexto es idéntico al utilizado en la representación de las experiencias pasadas. Los servicios de preprocesamiento de *core* y el de lógica del agente garantizan que el contexto siempre está actualizado.

Las experiencias pasadas pueden ser especificadas en un fichero YAML con un objeto por cada experiencia o fragmento de diálogo. Cada uno de estos objetos tendrá una lista de turnos cuyos atributos coinciden en nombre con los de las clases de `DialogueTurn` y `DialogueEntity`.

```
1 experience 1:
2   - intent: greeting
3     reaction:
4       - action_greet
5
6   - intent: order_food
7     entities:
8       - value: pizza
```

```

9     role: meal
10    reaction:
11      - action__add__meal
12
13    - intent: thanks
14      reaction:
15        - action__you__are__welcome
16        - action__enjoy__pizza

```

Código 39: Ejemplo de experiencia en formato YAML con 3 turnos.

Tal y como podemos apreciar en el ejemplo, una experiencia es una lista de pares estímulo-reacción. El primer estímulo o turno está caracterizado por tener una intención 'greeting'. Cualquier mensaje que tenga esta intención sería compatible con esta definición parcial del estímulo y habría un *match*. Si especificamos el valor de más campos como `sentiment`, `entities` o `was_set`, el espacio de mensajes que haría *match* será menor.

Supongamos que la representación YAML del contexto del agente coincide con la experiencia definida en el código 40.

```

1  experience 1:
2    - intent: greeting
3      sentiment: 0
4      entities:
5        - value: Manuel
6          category: PERSON
7          role: user__name
8          group: 1
9      reaction:
10     - action__greet
11
12   - intent: order__food
13     entities:
14       - value: pizza
15         category: FOOD
16         role: meal
17         group: 1
18     reaction:

```



```
19     - action_add_meal
20
21 - intent: thanks
```

Código 40: Ejemplo de experiencia en formato YAML con 3 turnos.

El turno más antiguo del que se tiene constancia hace *match* con el turno más antiguo de la experiencia pasada expuesta anteriormente. El sentimiento y las entidades no son especificados en dicho turno de la experiencia pasada, por lo que no se tienen en cuenta para comprobar si hay *match*. Las reacciones también coinciden, por lo que ambos turnos son compatibles. Los segundos turnos de ambos fragmentos de diálogo también hacen *match*.

Finalmente, el mensaje del tercer turno, no el turno en sí, hace *match* con el último turno de la experiencia pasada, puesto que tienen la misma intención. Dado que el agente está decidiendo cómo reaccionar ante un último mensaje con la intención 'thanks', la reacción no está disponible. Al coincidir todos los turnos anteriores y las propiedades de los estímulos del último turno, el modelo llega a la conclusión de que son experiencias lo suficientemente similares y decide reaccionar de la misma forma. La reacción elegida para el mensaje actual será la misma que la del último turno de la experiencia pasada: ['action_you_are_welcome', 'action_enjoy_pizza'].

Para cargar las experiencias pasadas desde un fichero YAML, utilizaremos la función o *loader* `load_experiences` como en el código 41.

```
1 from smartbot.loader.behavior_models.memories_loader import load_experiences
2
3 exp = load_experiences(
4     file_path=path,
5     true_to_str="Yes", false_to_str="No",
6     augmentation=True,
7     new_fragments_by_augmentation=300
8 )
```

Código 41: Ejemplo de experiencia en formato YAML con 3 turnos.

Si el valor del parámetro `augmentation` es `True`, además de cargar las experiencias, se eligen

aleatoriamente pares de experiencias y se concatenan para enriquecer el conjunto de experiencias creando combinaciones compatibles de fragmentos de forma automática.

`new_fragments_by_augmentation` establece un límite para el número de nuevos fragmentos que pueden generarse con esta técnica. Otro factor a tener en cuenta es que si el número de turnos de alguno de los fragmentos excede la longitud máxima k del histórico especificado al inicializar el estado interno del agente, solo se inspeccionará los k últimos turnos del fragmento.

Los campos `true_to_str` y `false_to_str` solo son necesarios si existen ocurrencias de 'yes' o 'no' en las claves o valores del YAML. En la versión utilizada del *parser* estos valores se interpretan como booleanos, por lo que es necesario corregir esta transformación y para ello se dispone de estos campos que indican la traducción en sentido inverso.

5.4.7. Modelo de comportamiento basado en reglas

El modelo de comportamiento basado en reglas es una generalización del basado en recuerdos. En el modelo anterior se definen una serie de reglas o condiciones de igualdad de forma implícita que, si bien pueden simplificar en gran medida la definición de reglas, pueden no ser lo suficientemente expresivas para hacer comprobaciones más complejas.

El modelo `RuleBasedBehaviorModel` es inicializado con un conjunto de reglas de la clase `BehaviorRule`. Para cada una de estas reglas disponemos de un conjunto de precondiciones sobre el estado interno que determinan si la regla es aplicable o no. En el caso de que la regla sea aplicable, se selecciona la reacción asociada a dicha regla.

Por ejemplo, la siguiente regla sería aplicable si y solo si la intención del mensaje actual es 'book_flight'.

```
1 BehaviorRule(  
2     description="Iniciar el flujo de reserva de vuelo si la intencion es book_flight",  
3     preconditions=[  
4         BehaviorRuleCondition(func=lambda s: s.get_intent() == "book_flight")  
5     ],  
6     reaction=["flow_start_book_flight"]  
7 )
```

Código 42: Ejemplo de regla para iniciar un flujo.

La clase base de las condiciones de las reglas `BaseBehaviorRuleCondition` define una operación `check(state: BotInternalState)` que devuelve `True` si y solo si se da la condición en un estado interno `BotInternalState`. La clase `BehaviorRuleCondition` nos permite definir condiciones con una lambda `func`, tal y como en el ejemplo anterior.

En nuestro *framework* se proporcionan 25 subclases de `BaseBehaviorRuleCondition` que preveemos que serán utilizadas frecuentemente por los usuarios. Algunas de estas condiciones *built-in* son las siguientes:

- `BehaviorIntentRuleCondition` establece la precondition de que el mensaje tenga una intención. También disponemos de `BehaviorIntentSetRuleCondition` para cuando la condición sea más flexible e incluya un conjunto de intenciones y no solo una.
- `BehaviorPositiveSentimentRuleCondition` establece la precondition de que el mensaje tenga un sentimiento positivo. Disponemos de clases de condición para establecer la precondition de que el mensaje tenga un sentimiento negativo, neutral o dentro de un rango.
- La condición `BehaviorAnyEntitiesRuleCondition` establece la restricción de que el mensaje tenga al menos una entidad. Si especificamos un rol o categoría, se evaluará a verdadero si y solo si existe al menos una entidad en el mensaje con el rol o categoría dados. Además de esta regla disponemos de otra para el número de entidades.
- Con `BehaviorActiveFlowRuleCondition` y `BehaviorNotActiveFlowRuleCondition` podemos comprobar si hay un flujo activo o no, respectivamente.
- Disponemos de clases de condiciones para los campos `was_set` y `expected_slots`. Asimismo, tenemos clases de condiciones para los casos en los que el usuario ha ignorado una petición o ha generado un error de validación de un *slot*.
- Si queremos comprobar si la conversación se empezó hace poco, podemos recurrir a `BehaviorHistoryAtMaxCapacityRuleCondition`.
- La condición `BehaviorFirstMessageRuleCondition` también está disponible si queremos aplicar una regla solo cuando el usuario ha enviado el primer mensaje.

- Existen más clases de condiciones para establecer la restricción de que un campo sea desconocido o no, nulo o no, etc.

En lugar de crear instancias de `BehaviorRuleCondition` y definir una *lambda* podemos recurrir a las clases de condiciones que hemos mencionado anteriormente. Supongamos que queremos crear una regla que se aplique cuando el usuario se encuentra en un flujo 'subscribe' y haya ignorado la pregunta anterior del agente sobre su correo electrónico. La reacción deseada es notificar al usuario que no ha contestado e informar de que no se le va a enviar promociones por si el usuario no proporcionó su email por ese motivo.

```

1 BehaviorRule(
2   description="Notificar que no ha contestado e informar de que no habra spam",
3   preconditions=[
4     BehaviorActiveFlowRuleCondition(flow_name="subscribe"),
5     BehaviorAnyExpectedSlotWasIgnoredRuleCondition(slots={"email"})
6   ],
7   reaction=["didnt_reply", "wont_send_you_spam"]
8 )

```

Código 43: Ejemplo de regla para controlar el caso en el que el usuario ignora una pregunta durante un flujo.

Otro posible ejemplo es que queramos que el agente pregunte el nombre al usuario al principio de la conversación si no lo conoce. Esta funcionalidad se podría implementar con la siguiente regla:

```

1 BehaviorRule(
2   description="Si no conocemos al usuario, le preguntamos su nombre",
3   preconditions=[
4     BehaviorFirstMessageRuleCondition(),
5     BehaviorIsUnknownRuleCondition(key="user_name"),
6     BehaviorNoEntitiesRuleCondition(with_category="NAME")
7   ],
8   reaction=["greeting", "ask_name"]
9 )

```

Código 44: Ejemplo de regla con precondiciones que establecen restricciones sobre el tipo de entidades que no deben estar presentes y los datos que no debe ser conocidos por el agente.

En el caso de que ninguna de las condiciones *built-in* satisfaga las necesidades del desarrollador, deberá recurrir a `BehaviorRuleCondition` y definir una *lambda*.

El modelo de comportamiento basado en reglas puede llegar a la conclusión de que más de una regla es aplicable. En estos casos se selecciona aquella que tenga más prioridad. Además de ser un mecanismo necesario para la selección, podemos utilizar las prioridades para definir reglas generales con prioridad baja y reglas específicas con prioridad alta. Por ejemplo, supongamos que tenemos una regla que tiene como condición que el sentimiento sea negativo y que la intención del mensaje sea 'user_review'. Podemos definir una segunda regla con exactamente las mismas condiciones y una adicional que establezca la restricción de que el usuario sea habitual. Si esta última regla es aplicable, la primera también lo será por lo que la prioridad decidirá cuál seleccionará el modelo. Dado que la regla con la condición adicional es más específica, debe ser seleccionada, por lo que le asignaríamos una mayor prioridad.

5.4.8. Modelo de comportamiento basado en flujos

En la sección 5.4.4 describimos la definición y uso de los *slots* de flujo. Estos *slots* se definen como instancias de la clase `FlowBotSlotDefinition`. El modelo de comportamiento basado en flujos selecciona la reacción a un estímulo en base a los *slots* de flujo pendientes definidos para el flujo activo. La definición de un flujo viene dada por una instancia de `FlowStateDefinition`.

```
1 @dataclasses.dataclass(init=True, repr=True)
2 class FlowStateDefinition:
3     name: str
4     slots: Dict[str, FlowBotSlotDefinition]
```

Código 45: Clase para la definición de los flujos.

La definición de un flujo viene dada por un nombre `name` y un diccionario de identificadores de *slots* de flujo y su correspondiente definición. El nombre es necesario en aquellos casos en los que el agente tenga que ejecutar una acción especial que inicie, reinicie o finalice

un flujo. Por ejemplo, 'flow_start_book_flight' es una acción del sistema que inicia un flujo con el nombre 'book_flight'.

La clase `FlowBehaviorModel` que implementa este tipo de modelo dispone de un diccionario de nombres y definiciones de flujo. Al iniciar un flujo, se almacena en la variable `active_flow` del estado interno la definición del flujo que ha sido iniciado. Al finalizar un flujo, se elimina este valor del estado interno.

Un detalle fundamental a tener en cuenta es la necesidad de ejecutar una segunda vez un servicio de *slot filling* en *core*. Supongamos que el usuario ha enviado un mensaje que inicia un flujo F e incluye datos que se corresponden con *slots* del flujo F . Durante el preprocesamiento en *core*, suponiendo que no había ningún flujo activo, solo se intentará aplicar un *slot filling* de *slots* globales. En una etapa posterior, uno de los modelos de comportamiento deduciría que este mensaje ha iniciado un flujo y procedería a delegar en la instancia de `FlowBehaviorModel` la generación de una reacción. Al ignorarse los datos iniciales proporcionados por el usuario, el agente volverá a preguntarlos si no tiene lugar un segundo *slot filling* tras haberse iniciado el flujo.

El procedimiento para seleccionar un *slot* consiste en obtener una lista de los *slots* pendientes de rellenar y elegir aquel que tenga una mayor prioridad. Tras seleccionar el *slot*, se procede a almacenar en la variable `expected_slots` del estado interno para que el agente pueda determinar si la petición ha sido ignorada por el usuario en su siguiente intervención.

5.4.9. Modelo de comportamiento basado en modelos estadísticos

Los modelos de comportamiento expuestos hasta el momento modelan mecanismos de reacción deterministas. En ciertos contextos es necesario que un agente reaccione de una forma predecible, pero también debemos recurrir a modelos que generalicen y sean capaces de determinar una reacción ante una situación inesperada.

El modelo `KerasBehaviorModel` permite recurrir a un modelo de red neuronal de Keras para la generación de una reacción. Si queremos utilizar un modelo ya compilado y no entrenado debemos pasar este modelo en la inicialización de `KerasBehaviorModel` con el parámetro `model`.

El modelo Keras utilizado por defecto es una red neuronal recurrente que consta de una capa `Masking` que ignorará los vectores con todas las componentes a cero (i.e. vectores que

hacen de *padding*), una capa de unidades LSTM y una capa de salida `Dense` con una función de activación *softmax*. Dado que debemos aplicar una codificación a las acciones, esta capa de salida tendrá tantas neuronas como componentes los vectores que codifiquen una acción. A continuación mostramos una lista de algunos de los parámetros que nos permiten personalizar este modelo predeterminado:

- **Número de unidades LSTM:** `lstm_units`. Por defecto, su valor es 100.
- **Función *loss* u objetivo de la optimización:** `loss`. Por defecto, es `'categorical_crossentropy'`, esto es, se computa la entropía cruzada entre acciones y predicciones.
- **Optimizador:** Por defecto, se recurre al algoritmo RMSprop que tiene asociado el identificador `'rmsprop'`. Podemos seleccionar otro optimizador a través del parámetro `optimizer`.
- **Métricas:** Una lista de métricas para evaluar el rendimiento del modelo. Por defecto, su valor es `['accuracy']`. Podemos configurar otra lista de métricas a través del parámetro `metrics`.

Este modelo de comportamiento no es de paso grande como los anteriores. En cada iteración se computa la siguiente acción a incluir en la reacción, dado un contexto que incluye información sobre el último mensaje y la última acción ejecutada, independientemente de si esta acción fue ejecutada como parte de otra reacción generada por un modelo anterior.

Las estructuras de datos utilizadas para representar el contexto están basadas en las utilizadas en el modelo basado en recuerdos. Un conjunto de entrenamiento es una lista de listas de `DialogueTurn` en el que se proporciona información como la intención, el sentimiento o los *slots* que fueron rellenados. El modelo de comportamiento asume que al final de la reacción se ejecuta una acción `listen` que no es necesaria incluir explícitamente en la lista.

El diseño de este modelo de comportamiento llevo a la creación de dos estructuras mentales adicionales. La primera de ellas, `StateSummary`, proporciona un resumen del estado, esto es, contiene información básica sobre el estímulo recibido (i.e. intención, sentimiento, *slots* completados, etc.), la última acción ejecutada por el agente y la siguiente a ejecutar. El resultado de codificar un mensaje y una reacción de k acciones al mismo es una lista de k instancias de `StateSummary` que solo difieren en la última y siguiente acción. El resultado de la codificación de una instancia de `StateSummary` es una instancia de `EncodedStateSummary` que es

un *wrapper* de una instancia de `StateSummary` e incluye los vectores que codifican el estado que resume y la siguiente acción.

El modelo Keras debe ser capaz de predecir la siguiente acción a partir de una lista de `max_history` instancias de `EncodedStateSummary`. La implementación de nuestro *featurizer* o codificador de estados es `StateFeaturizerService`. Se pueden proporcionar codificadores personalizados a través del parámetro `featurizer` del modelo de comportamiento `KerasBehaviorModel`.

Al inicializar `StateFeaturizerService` debemos especificar una lista de intenciones, sentimientos, *slots* y acciones. A excepción de los *slots*, este servicio aplica una codificación *One-Hot*. Por ejemplo, si tenemos k intenciones y m acciones, los vectores que codifican las intenciones y acciones tendrán k y m componentes, respectivamente. En el caso de los *slots* no se codifica su valor, sino su presencia o ausencia.

El vector que representa el contexto es el resultado de la concatenación de los vectores de las intenciones, sentimientos y *slots*. El vector introducido en la red neuronal es el resultado de concatenar este vector contextual con la codificación de la última acción estimada por el modelo. En el caso de que no haya una última acción, se asume que es 'listen'.

En cada iteración, se reemplaza el vector de la última acción y se mantiene intacto el vector contextual que codifica el estímulo. El resultado de la red es decodificado y la acción se añade a la lista de acciones si no es 'listen'.

Para concluir esta sección mostramos un ejemplo de consumo de este modelo de comportamiento. Nuestro conjunto de datos será el siguiente:

```
1 from smartbot.services.core.symbols import DialogueTurn
2
3 experiences = [
4     [
5         DialogueTurn(
6             intent="greeting",
7             reaction=["action_greet", "action_ask_name"],
8             was_set=set()
9         ),
10    ],
11    [
12        DialogueTurn(
```



```

13     intent="greeting",
14     reaction=["action_greet", "action_tell_name"],
15     was_set={"name"}
16 ),
17 ],
18 [
19     DialogueTurn(
20         intent="greeting",
21         reaction=["action_greet", "action_ask_name"],
22         was_set=set()
23     ),
24     DialogueTurn(
25         intent="say_name",
26         reaction=["action_tell_name"],
27         was_set={"name"}
28     ),
29 ],
30 [
31     DialogueTurn(
32         intent="greeting",
33         reaction=["action_greet", "action_ask_name"],
34         was_set=set()
35     ),
36     DialogueTurn(
37         intent="chitchat",
38         reaction=["action_chitchat", "action_ask_name"],
39         was_set=set()
40     ),
41 ],
42 ...
43 ]

```

Código 46: Ejemplo de definición de experiencias desde código.

Si se consume el servicio directamente y no forma parte de un agente, debemos inicializar el *featurizer* y el modelo de comportamiento. Recomendamos que el parámetro `max_history` sea próximo a 10 para que el modelo tenga suficiente información durante el entrenamiento para poder realizar predicciones de calidad.

```

1 featurizer = StateFeaturizerService(
2     intents=["greeting", "say_name", "chitchat"],
3     slots=["name"],
4     actions=["action_greet", "action_ask_name", "action_chitchat", "action_tell_name"]
5 )
6 featurizer.set_log_level(logging.DEBUG)
7
8 keras_model = KerasBehaviorModel(
9     featurizer=featurizer,
10    max_history=10,
11    lstm_units=100
12 )
13 keras_model.set_log_level(logging.DEBUG)

```

Código 47: Instanciación del modelo de comportamiento de Keras y ejemplo de instanciación del servicio de codificación del estado de las conversaciones.

Dado el contexto de que un usuario ha enviado un primer mensaje con la intención `greeting` en el que no menciona su nombre, el modelo de Keras ha propuesto la siguiente reacción: `['action_greet', 'action_ask_name', 'listen']`. La última acción será ignorada por el servicio de acciones al tratarse de un delimitador.

5.4.10. Servicio de ejecución de acciones

El servicio `ActionsService` ejecuta las reacciones o listas de acciones generadas por los modelos de comportamiento de forma secuencial. Se inicializa con un diccionario `actions` que asocia punteros de funciones a identificadores de acciones, permitiendo la ejecución de los símbolos que representan las acciones. En cuanto al formato utilizado para estos identificadores, recomendamos que empiecen con el prefijo `'action_'`, puesto que de esta forma se evitará conflictos con las acciones especiales que existen actualmente en el *framework* y las que se introducirán en un futuro.

Uno de los requisitos de las funciones asociadas a acciones es que deben recibir un único argumento que será una petición `ActionRequest`. A través de esta petición se puede acceder al estado interno del agente, una lista de canales de salida en los que emitir el mensaje y un canal *broadcast* de salida.

Recomendamos la creación de un módulo `actions.py` con las acciones del agente para separar el código de inicialización del agente de las acciones. A continuación mostramos un ejemplo de cómo se definen las funciones asociadas a los símbolos de acciones.

```
1 # Definimos un generador de plantillas NLG (vease seccion sobre NLG)
2 nlg = TemplateGeneratorService(templates=...)
3
4 def generate_date(request: ActionRequest):
5     import datetime
6
7     if "make_appointment/proposed_date" not in request.state:
8         request.state["proposed_date"] = ...
9     else:
10        # if the user rejected the last date we proposed
11        request.state["proposed_date"] += datetime.timedelta(days=7)
12
13
14 def suggest_date(request: ActionRequest):
15     date = request.state["proposed_date"]
16
17     template = nlg.execute(intent="suggest_date")
18     text = template.instantiate(date=date.strftime("%d/%m/%Y, %H:%M"))
19     request.broadcast_channel.emit(text)
```

Código 48: Ejemplos de acciones.

Existe un grupo de acciones de sistema que son tratadas de forma especial. No es necesario que el desarrollador las registre, pero sí pueden ser mencionadas en los conjuntos de entrenamiento de los modelos de comportamiento. Las tres acciones especiales soportadas son el inicio, *reset* y finalización de un flujo. La acción *listen* se utiliza como delimitador de final de secuencia de acciones y, por ende, se ignora, por lo que no se considera una acción del sistema.

```
1 class SystemAction(Enum):
2     START_FLOW = 0
3     RESET_FLOW = 1
4     FINISH_FLOW = 2
5     RESUME_FLOW = 3
```

```
6 REQUEST_CONFIRMATION_FOR_RESET_FLOW = 4
7 REQUEST_CONFIRMATION_FOR_FINISH_FLOW = 5
8 REQUEST_CONFIRMATION_FOR_FINISH_FLOW_AND_START ...
```

Código 49: Acciones especiales soportadas.

El servicio `ActionsService` es capaz de detectar acciones especiales inspeccionando el prefijo o cadena completa del identificador de la acción. Las acciones especiales actuales controlan el inicio, reinicio, finalización y continuación de un flujo. Asimismo, se incluyen acciones para aquellos casos en los que el agente solicita una confirmación al usuario sobre la terminación o reinicio del flujo. La última acción del enumerado es de finalización del flujo actual para iniciar otro distinto.

Todo símbolo de acción que empiece con el prefijo `'flow_start_'` y termine con el nombre de un flujo F , se interpreta como la acción especial de iniciar F . Los símbolos de las acciones de reinicio, finalización y continuación son `'flow_reset'`, `'flow_finish'` y `'flow_resume'`, respectivamente. Las de confirmación de reinicio y finalización son `'flow_reset_confirmation'` y `'flow_finish_confirmation'`.

Si bien el servicio es capaz de detectar acciones de sistemas, se delega en el servicio de planificación la tarea de ejecutar las acciones de sistema actualmente soportadas. Esto se debe a que la ejecución de este tipo de acciones especiales suele conllevar la planificación de acciones utilizando el modelo de comportamiento basado en flujos. Al inicializar el servicio debemos especificar el *callback* `on_special_action` que será invocado al detectar una acción especial.

5.5. Generación de lenguaje natural

5.5.1. Introducción

La generación de lenguaje natural (NLG) consiste en la automatización de la redacción de textos comprensibles en un determinado lenguaje natural (Reiter y Dale, 2000, p. 1). Una de las aplicaciones más comunes de la tecnología NLG disponible es la generación de textos comprensibles para el usuario a partir de representaciones complejas, posiblemente en un formato no legible, que podrían requerir conocimientos avanzados para su comprensión (Reiter y Dale, 2002). Por ejemplo, si la información sobre un pedido está almacenada de forma estructurada

en una base de datos SQL, el sistema NLG deberá generar una representación textual (i.e. no estructurada) y, posiblemente simplificada de los registros correspondientes.

Aunque existen métodos de generación texto a texto que nos permiten, entre otras aplicaciones, generar un resumen del texto introducido como entrada en el sistema, nuestro trabajo se ha limitado a aquellos algoritmos y técnicas que permiten generar un texto a partir de representaciones no lingüísticas (Gatt y Krahmer, 2017, p. 5).

Reiter y Dale (2000) subdividen el problema de la generación natural en seis tareas que pueden ser obligatorias u opcionales en función de los requisitos de nuestro sistema NLG:

- **La determinación del contenido de un mensaje:** Podemos definir el contenido del mensaje como la información seleccionada para ser incluida en el mismo, la cual se puede representar como un conjunto de estructuras preverbales expresadas en algún lenguaje formal (Gatt y Krahmer, 2017, pp. 10-11).
- **La generación de la estructura de un mensaje:** Dada la naturaleza secuencial del lenguaje, la siguiente etapa en el proceso de generación del mensaje es determinar en qué orden debemos expresar la información seleccionada (Reiter y Dale, 2000).
- La unificación o agregación de enunciados para suprimir la redundancia sintáctica y semántica.
- La lexicalización o tarea de determinar las palabras o frases adecuadas para transmitir cierta información.
- La tarea de generar expresiones de referencia a entidades que sean simples, pero lo suficientemente detalladas para que el receptor pueda identificar a la entidad sin ambigüedad (Bohnet y Dale, 2005).
- **Realización lingüística:** La etapa final del proceso de generación es la construcción del mensaje en el lenguaje natural objetivo. Algunas de las tareas de esta etapa son la inserción de símbolos de puntuación, palabras vacías necesarias, la conjugación de verbos, etc (Gatt y Krahmer, 2017, p. 19).
 - **Plantillas:** Una plantilla es una representación lingüística que contiene *slots* para introducir la representación textual de estructuras no necesariamente lingüísticas

(van Deemter, Theune, y Krahmer, 2005, pp. 1-2). Dada su simplicidad puede ser adecuada para dominios simples, pero hemos de tener en cuenta que los sistemas basados en plantillas pueden ser más difíciles de mantener (van Deemter y cols., 2005, p. 4). Si el dominio del agente pasa a ser más complejo, puede ser necesario recurrir a un método más sofisticado (Gatt y Krahmer, 2017, p. 19).

- **Gramáticas:** Una alternativa a la construcción manual de una lista de plantillas es el uso de gramáticas que pueden ser construidas manualmente o generadas a través de métodos estadísticos (Gatt y Krahmer, 2017, p. 20). Destacamos el *framework* de generación probabilística del lenguaje pCRU, que trata el conjunto de reglas de generación de lenguaje como la definición de una gramática de contexto libre. Tras definir una gramática de este tipo de forma manual, se estima una distribución de probabilidad a partir de un *corpus* que modela el proceso de decisión que elige las reglas a aplicar para generar el mensaje (Belz, 2007).

5.5.2. Enfoque de nuestro *framework* para la generación de texto

Nuestro *framework* recurre a un enfoque basado en plantillas para la generación de mensajes. Una plantilla está constituida por una serie de *slots* $\langle s_1 \rangle, \dots, \langle s_k \rangle$ de la clase `TemplateSlot` y un *string* en el que podemos encontrar al menos una ocurrencia de cada uno de ellos. Un ejemplo de la representación textual de una plantilla podría ser el siguiente:

Su destino está a <distance> <distance_unit>. Hora estimada: <arrival_time>.

En este ejemplo, la plantilla tiene tres *slots*: `distance`, `distance_unit` y `arrival_time`. Tal y como podemos apreciar, los *slots* aparece en la representación textual de la plantilla entre paréntesis angulares.

5.5.3. Definición de una plantilla desde código

Un modelo contiene los datos necesarios para rellenar los *slots* de una plantilla. La estructura de datos utilizada para construir un modelo será un diccionario de Python.

```
1 model = {  
2     "distance": Distance(7.3, DistanceUnit.KM),  
3     "arrival_time": datetime.datetime(2021, 1, 25, 23, 55, 59),
```

```

4     "location": (36.6628707,-4.5521375)
5 }

```

Código 50: Ejemplo de modelo de datos.

Al crear un *slot* asociamos a un identificador `name` de tipo *string* un *slot* `bind_to`. Por ejemplo, podemos crear un *binding* entre una propiedad `user_name` de un modelo y un *slot* `name` de una plantilla. Debemos mencionar que no es necesario que el nombre de la propiedad y el identificador del *slot* coincidan. Se introduce esta flexibilidad para permitir que más de un *slot* esté vinculado a una misma propiedad. Para definir un *slot* debemos crear una instancia de la clase `TemplateSlot`:

```

1 user_name_slot = TemplateSlot(
2     name="user_name",
3     bind_to="name"
4 )

```

Código 51: Ejemplo de definición de *slot*.

En el modelo expuesto anteriormente la propiedad `distance` es una instancia de una clase con un atributo `value` que almacena el valor numérico de la distancia al destino, y un atributo `unit` de tipo enumerado que almacena la unidad utilizada para expresar la distancia. Queremos utilizar esta información para rellenar los *slots* de la siguiente plantilla:

Su destino está a <distance> <distance_unit>. Hora estimada: <arrival_time>.

Los dos primeros *slots* serán vinculados a la propiedad `distance` del modelo. En ambos casos será necesario aplicar una transformación previa del valor de esta propiedad para obtener una representación textual adecuada. Esta transformación puede ser aplicada a través de un *converter* de la clase `Converter`. Nuestro *framework* proporciona una serie de *built-in converters*, pero es posible crear uno personalizado creando una subclase de la clase abstracta `Converter` y definiendo el método `convert`. Para evitar en la medida de lo posible que el desarrollador tenga que crear subclases, proporcionamos la clase `LambdaConverter` que recibe como argumento una función lambda de un parámetro y devuelve el resultado de la transformación. Para utilizar un *built-in converter* debemos simplemente proporcionar su identificador:

```

1 user_name_slot = TemplateSlot(
2     name="user_name",
3     bind_to="name",
4     converter="upper"
5 )

```

Código 52: Ejemplo de definición de *slot* con *converter*.

A continuación exponemos una lista de los *built-in converters* definidos por nuestra librería:

- **upper**: Muestra el valor de la propiedad de tipo `str` en mayúsculas.
- **lower**: Muestra el valor de la propiedad de tipo `str` en minúsculas.
- **capitalize**: Muestra el primer caracter del valor en mayúsculas y el resto en minúsculas.
- **day**: Si el valor es una fecha de tipo `datetime`, se muestra el día.
- **month**: Si el valor es una fecha de tipo `datetime`, se muestra el mes.
- **year**: Si el valor es una fecha de tipo `datetime`, se muestra el año.
- **time**: Si el valor es una fecha de tipo `datetime`, se muestra la fecha y hora en un formato hh:MM.

Si queremos utilizar *converters* personalizados en los ficheros YAML en los que podríamos definir plantillas, debemos registrarlos y asociarles un identificador no utilizado por otro *converter*, incluyendo los definidos por la librería.

En el ejemplo de esta sección recurriremos a dos *converters* personalizados. El *slot* del valor numérico de la distancia utilizará un `LambdaConverter` que transformará una instancia de `Distance` en un número decimal que cuantifica la distancia. El *converter* del *slot* de la unidad de la distancia será una subclase de `Converter` que obtendrá el valor del atributo `unit` de la instancia de `Distance` y seleccionará una representación textual para el valor del enumerado.

```

1 class DistanceUnitConverter(Converter):
2     def convert(self, value: Distance):
3         if value.unit == DistanceUnit.KM:
4             return "km"

```



```

5     elif value.unit == DistanceUnit.M:
6         return "m"
7     return "?"

```

Código 53: Ejemplo de definición de *converter*.

La definición completa de los *slots* del ejemplo de esta sección es:

```

1 distance = TemplateSlot(
2     name="distance",
3     bind_to="distance",
4     converter=LambdaConverter(lambda d: d.value)
5 )
6
7 distance_unit = TemplateSlot(
8     name="distance_unit",
9     bind_to="distance",
10    converter=DistanceUnitConverter()
11 )
12
13 arrival_time = TemplateSlot(
14     name="arrival_time",
15     bind_to="arrival_time",
16     converter="time"
17 )

```

Código 54: Ejemplo completo de definición de *slots* con los tres tipos de *converters*.

La plantilla, esto es, su representación textual y la definición de sus *slots*, es una instancia de la clase `Template`. El orden de los *slots* en la lista que creamos para definir la plantilla no es relevante.

```

1 template = Template(
2     "Su destino está a <distance> <distance_unit>. Hora estimada: <arrival_time>.",
3     [distance, distance_unit, arrival_time]
4 )

```

Código 55: Ejemplo de definición de plantilla.

5.5.4. Definición YAML de una plantilla

Podemos definir plantillas tal y como hemos hecho en la sección anterior desde uno o varios ficheros YAML. La estructura de estos ficheros es de un objeto raíz con un objeto por cada plantilla. El objeto YAML de la plantilla consta de un nombre, un atributo `text` que coincide con la representación textual de la plantilla, y una lista de `slots` que tienen una estructura equivalente a las instancias de `TemplateSlot`.

La plantilla de ejemplo definida al final de la sección anterior se representaría de la siguiente forma en el fichero YAML:

```
template_distance:
  text: Su destino está a <distance> <distance_unit>. Hora estimada <arrival_time>.
  slots:
    - name: distance
      bind_to: distance
      converter: distance_to_value

    - name: distance_unit
      bind_to: distance
      converter: distance_to_units

    - name: arrival_time
      bind_to: arrival_time
      converter: time
```

Código 56: Ejemplo de definición de plantilla en formato YAML.

Si la plantilla no contiene *slots* podemos omitir ese campo en la definición YAML del *slot*:

```
template_greet:
  text: Bienvenido. Hoy seré su asistente.
```

Código 57: Ejemplo de definición de plantilla sin *slots* en formato YAML.

Un ejemplo más completo de fichero YAML de plantillas se proporciona a continuación:

```
greet_unknown_user:
  text: ¡Hola!
```

```

greet_known_user:
  text: ¡Hola <name>!
  slots:
    - name: name
      bind_to: user_name
      converter: upper

ask_origin:
  text: ¿Desde qué ciudad desea tomar el vuelo?

ask_destination:
  text: ¿Qué ciudad desea visitar?

confirmation_origin:
  text: La ciudad desde la que tomaría el vuelo sería <from>. ¿Es correcto?
  slots:
    - name: from
      bind_to: flight
      converter: flight_to_origin

confirmation_destination:
  text: La ciudad desde la que tomaría el vuelo sería <dest>. ¿Es correcto?
  slots:
    - name: dest
      bind_to: flight
      converter: flight_to_dest

```

Código 58: Ejemplo completo de definición de varias plantillas en formato YAML.

Para cargar las plantillas desde un fichero YAML, utilizaremos la función o *loader* `load_templates` como sigue:

```

1 from smartbot.loader.nlg.template_loader import load_templates
2
3 exp = load_templates(
4     file_path=path,
5     true_to_str="Yes", false_to_str="No",

```

```

6     register_converters={...}
7 )

```

Código 59: Carga de plantillas definidas en YAML.

A través de `register_converters` podemos utilizar *converters* personalizados en las plantillas en formato YAML. En este campo se debe pasar un diccionario en el que las claves son los identificadores de los *converters* en el fichero YAML, y los valores son instancias de estos *converters*.

Los campos `true_to_str` y `false_to_str` solo son necesarios si existen ocurrencias de 'yes' o 'no' en las claves o valores del YAML. En la versión utilizada del *parser* estos valores se interpretan como booleanos, por lo que es necesario corregir esta transformación y para ello se dispone de estos campos que indican la traducción en sentido inverso.

5.5.5. Servicio de selección de plantillas

La selección de plantillas está guiada por una petición de generación de lenguaje natural. La clase `TemplateGeneratorService` proporciona la implementación de un servicio no entrenable de selección de plantillas que solo tiene en cuenta la intención expresada en la petición NLG. Para cada intención podemos disponer de una plantilla o una lista de plantillas. En el caso de que se proporcione una lista, se seleccionará de forma aleatoria una de ellas.

```

1 service = TemplateGeneratorService(templates={
2     "greet": [template_hello, template_hello_alt],
3     "bot_challenge": [template_iambot, template_iambot2, template_iambot3],
4     "farewell": [template_farewell]
5 })
6
7 # Seleccionamos de forma aleatoria una plantilla para la intención 'greet'
8 template = service.execute(intent="greet")
9
10 # Rellenamos la plantilla con un modelo constituido por la clave
11 # user_name: Manuel
12 msg = template.instantiate(user_name="Manuel")

```

Código 60: Selección e instanciación de plantilla.

En el ejemplo anterior la plantilla se selecciona de forma aleatoria, por lo que debemos instanciar la plantilla generada con todos los campos esperados por cualquiera de las plantillas asociadas a la intención de *bot* en cuestión. Una plantilla solo consume los campos que están vinculados a los *slots* que la constituyen. Si el modelo que le pasamos a una plantilla tiene más campos de los que necesita, serán ignorados los campos que no estén vinculados a ningún *slot*.

5.6. Otros servicios genéricos

Se proporciona una implementación de servicio `WikidataService` para tener acceso a información disponible en Wikidata. Al ejecutar el servicio podemos proporcionar un identificador de entidad de Wikidata `entity_id` para obtener información sobre la entidad. También es posible especificar un id de propiedad `property_id` para recuperar su valor para una entidad dada.

```
1 from smartbot.services.core.misc.wiki_service import WikidataService
2
3 wiki = WikidataService()
4
5 spain_entity_id = "Q29"
6 capital_property_id = "P36"
7
8 result = wiki.execute(
9     entity_id=spain_entity_id,
10    property_id=capital_property_id
11 )
12
13 print(result.label) # Prints 'Madrid'
```

Código 61: Instanciación y ejecución del servicio de Wikidata.

6

Diseño del agente conversacional

En los capítulos anteriores hemos aplicado un enfoque teórico-práctico al tratamiento del problema de la comprensión del lenguaje natural y el procesamiento de estructuras mentales en agentes conversacionales. Ambos problemas son resueltos definiendo dos procesos cognitivos que consumen servicios de nuestro *framework*. Tal y como expusimos en nuestra teoría de agentes conversacionales, un agente es una colección de procesos que interactúan entre sí dando lugar a un comportamiento externo e interno. Los objetivos de este capítulo son la descripción de la metodología a seguir para definir procesos y agentes conversacionales en nuestro *framework*, y la exposición de un caso práctico.

6.1. Arquitectura de los procesos

En la sección 3.1.3 del capítulo de teoría postulamos que un agente está constituido por procesos que a su vez están constituidos de servicios. Dado que nuestro trabajo se centra en agentes conversacionales, consideramos que como mínimo deben implementarse un proceso de comprensión y uno de *core*. En los dos capítulos anteriores expusimos los servicios de comprensión y *core* que ofrece nuestra librería, y en esta sección vamos a exponer cómo podemos crear los procesos a partir de estos servicios.

La clase base de los procesos `Process` presenta una interfaz similar a la de los servicios. Se pueden entrenar y ejecutar, esto es, tienen un método `train` y `execute` al igual que todos los servicios. Podría contemplarse a un proceso como un servicio orquestador que coordina el orden de ejecución de una serie de servicios. Los atributos a destacar de esta clase base son los siguientes:

- **Stateful:** Al recibir una petición de ejecución, el proceso recibe un diccionario con la entrada. Si el proceso es *stateless*, las ejecuciones son independientes entre sí, esto es, no se preserva información sobre un cómputo anterior. Por el contrario, si es *stateful*, además de almacenarse el estado final de la memoria en la última ejecución, esta información se unifica con el diccionario correspondiente a la nueva entrada.
- **Claves de salida:** La salida de un proceso, como norma general, será el contenido de su memoria interna al finalizar la ejecución de una entrada. A través de las claves de salida podemos aplicar un filtrado previo al resultado final de la memoria para devolver exclusivamente aquella información que sea de interés para otros procesos. Por ejemplo, si `output_keys` es una lista con las variables `x` y `z` y el contenido final de la memoria contiene estas dos variables y una variable auxiliar `y`, el proceso devolverá un diccionario filtrado en el que solo aparecen las claves `x` y `z`.
- **Nombre:** Identifica al proceso en las trazas facilitando así la depuración.

Los servicios que constituyen los procesos se encapsulan en nodos computacionales para lidiar con los problemas expuestos en 3.1.3 en relación con el acceso y modificación de la memoria del proceso del que forma parte. Cada nodo es una instancia de `ProcessNode` que encapsula un servicio de tipo `Service`, asigna un identificador en la memoria compartida a cada salida del mismo (`output_mapping`), y define una traducción del espacio de identificadores de la memoria al espacio de parámetros del servicio (`input_mapping`). Aunque esta traducción tiene lugar en el proceso de forma automática, mostramos un ejemplo del procedimiento:

```

1 from arch.process.base import ProcessNode
2 from services.preprocessing.segmentation import SpacyTokenizer
3
4 tokenizer = SpacyTokenizer(lang_model="es_core_news_md")
5
6 node = ProcessNode(
7     service=tokenizer,
8     input_mapping={
9         "text": "normalized_text" # normalized_text (memory) -> text (param)
10    },
11    output_mapping=["spacy_tokens"], # su salida se guardara como spacy_tokens

```

```

12     training_input_mapping={ # equivalente en este caso a input_mapping
13         "text": "normalized_text"
14     },
15 )
16
17 shared_memory = {
18     "text": "Buenos días!!! Qué tal?",
19     "normalized_text": "buenos dias! que tal?"
20 }
21
22 tokens = node.service.execute(text=shared_memory[node.input_mapping["text"]])
23
24 print(tokens) # buenos dias! que tal? => ha consumido normalized_text y no text

```

Código 62: Traducción del espacio de identificadores de la memoria al espacio de parámetros del servicio. Esta traducción tiene lugar de forma automática en los procesos.

La arquitectura de proceso secuencial propuesta en la teoría se implementa como una subclase `PipelineProcess` de la clase abstracta `Process`. No existe una clase para la arquitectura más general, la del grafo dirigido acíclico, porque consideramos que para alcanzar tal grado de personalización del orden y paralelismo computacional, el desarrollador debería crear una subclase propia de `Process`. La arquitectura secuencial se define como una lista de nodos computacionales de la clase `ProcessNode`.

La memoria compartida por los nodos es un diccionario que se corresponde con el parámetro `kwargs` de los métodos `execute` y `train` de los servicios. Tras la ejecución de cada servicio se actualiza esta memoria, permitiendo a los servicios utilizar información generada por otros en etapas anteriores. En la fase de entrenamiento, cada servicio que requiere entrenamiento accede al contenido de la memoria compartida y selecciona aquella información que sea necesaria para el aprendizaje o ajuste. Tras entrenar un servicio, se ejecuta en modo de ejecución múltiple para que servicios de etapas posteriores puedan usar esta nueva información para su entrenamiento. Por ejemplo, un servicio de creación de características debe ser ejecutado durante la fase de entrenamiento del proceso para que un servicio de extracción de entidades basado en CRF pueda consumir esta información durante el entrenamiento.

Siguiendo un enfoque dirigido por datos, podemos definir una *pipeline* simple y no anidada en un fichero YAML. El formato de este fichero es el siguiente:


```

name: <nombre del proceso>
stateful: <si se omite, se interpreta como False>
output_keys: # si se omite, se devuelve el estado completo del proceso
  - key_1
  - key_2
pipeline:
  - name: <nombre del primer servicio>
    <key de uno de los parametros del constructor>: <valor>
    <key de uno de los parametros del constructor>: <valor>
    <key de uno de los parametros del constructor>: <valor>
  mapping:
    inputs:
      <nombre del parametro>: <nombre del identificador en memoria>
      <nombre del parametro>: <nombre del identificador en memoria>
    outputs: [<id salida 1>, <id salida 2>, ...]

```

Código 63: Formato del fichero YAML en el que se define un proceso.

El objeto raíz de la definición de una *pipeline* contiene una lista de objetos YAML que definen instancias de nodos computacionales. La clave `name` debe contener el nombre de la clase del servicio utilizado. La clave `mapping` tiene por valor un objeto con al menos estas dos claves:

- La propiedad `input_mapping` de los nodos computacionales se especifica definiendo propiedades YAML para el objeto `inputs` de `mapping`. Estas propiedades tendrán por nombre el identificador de uno de los parámetros soportados para el método `execute` del servicio en cuestión. El valor de los mismos debe coincidir con algunos de los identificadores de las variables presentes en la memoria compartida en el momento de ejecutar el servicio.
- La propiedad `output_mapping` es una lista de identificadores. El identificador *i*ésimo será utilizado para almacenar e identificar la salida *i*ésima del servicio.

En el siguiente fragmento podemos ver la definición de una etapa de preprocesamiento que utiliza el servicio `SpacyTokenizer` y lo instancia pasándole un argumento `lang_model` con el nombre del modelo de lenguaje SpaCy. Esta etapa genera una lista de *tokens* que serán almacenados como `spacy_tokens_normalized_text` tras consumir un texto `text` que contiene el valor de la variable `normalized_text` supuestamente presente en la memoria compartida.

```
- name: SpacyTokenizer
  lang_model: es_core_news_md
  mapping:
    inputs:
      text: normalized_text
    outputs: [spacy_tokens_normalized_text]
```

Código 64: Ejemplo de definición de etapa en un proceso *pipeline*.

Para cargar en memoria un proceso con una arquitectura de *pipeline* utilizamos la función `load_pipeline` que recibe como argumento el *path* del fichero YAML y una lista opcional `custom_services` de clases de servicios que han sido definidos por el usuario y no forman parte de la librería.

```
1 from loader.nlu_dataset_loader import load_dataset
2 from loader.pipeline_config import load_pipeline
3
4 import os
5
6 path = os.path.join("resources", "preprocessing", "config", "pipeline.yml")
7
8 # 1. Inicializacion
9 process = load_pipeline(path, custom_services=None)
10
11 # 2. Entrenamiento (opcional)
12 training_data = {...}
13 process.train(**training_data)
14
15 # 3. Ejecucion
16 output = process.execute(text="Hola ¿Qué tal? Soy Luis")
```

Código 65: Ejemplo de carga de un proceso *pipeline* definido en un fichero YAML, su entrenamiento y ejecución.

6.2. Configuración del proceso de comprensión

La creación de un nuevo proceso cognitivo para un agente conversacional consta de cuatro pasos. El primero de ellos es la elección de una arquitectura del proceso. Como norma general, `PipelineProcess` debería ser suficiente. El segundo paso es elegir una serie de servicios de preprocesamiento o NLU de la *suite* ofrecida por el *framework*.

Tal y como comentamos en la teoría, es necesario aislar las instancias de los servicios en nodos computacionales y definir un *mapping* que especifique su interacción con la memoria compartida a la hora de leer y escribir en ella. El último y cuarto paso es el entrenamiento, si fuera necesario, del proceso, esto es, de los servicios que lo constituyen.

Tenemos dos opciones para definir un proceso *pipeline*: *programmatically* o con un fichero YAML. Sea cual sea el método por el que se decante el desarrollador deberá garantizar que el proceso NLU sea *stateless* y que este incluya una instancia del servicio `CoreMessageBuilder`. Este servicio fue introducido para crear una representación estándar para el proceso *core* que sea independiente de la elección de nombres de variables de salida del proceso para los campos de intención, sentimiento y entidades nombradas.

6.3. Configuración del proceso *core*

Tal y como se ha planteado durante el diseño del *framework*, el proceso *core* puede definirse con dos servicios que proporcionamos: `CorePreprocessor` y `BotLogicService`. El primero de ellos se encarga de aplicar un preprocesamiento al mensaje generado por el proceso de comprensión. El segundo actúa en base al estímulo recibido y el contexto en el que agente se encuentra. Consideramos que estos dos servicios son suficientes para definir proceso *core* de agentes complejos.

La inicialización de un proceso que utilice el servicio `BotLogicService` debe hacerse a través del código y no un fichero YAML. Al inicializar este servicio deberemos asociar a cada

acción del agente una función o *lambda* que la ejecute.

6.4. Creación de un agente conversacional

Una vez que tenemos inicializados los procesos del agente conversacional, debemos encapsularlos de la misma forma que hicimos con los servicios. La clase `BotNode` representa un nodo computacional de *bot* que encapsula un proceso cognitivo. Al igual que con los servicios debemos especificar un *mapping* entre las variables de entrada y salida, y los parámetros y salidas del método `execute`.

Se proporciona una clase para los agentes denominada `Bot`. Su inicialización requiere un nodo computacional de *bot* para el proceso de comprensión y otro para el de *core*. Ambos nodos deben ser previamente entrenados. Para facilitar la depuración del agente se exponen dos métodos `understand(message)` y `react(message)` que ejecutan el proceso de comprensión y *core*, respectivamente. El primero de ellos recibe un mensaje del tipo `BaseIncomingMessage`, mientras que el segundo solo recibe mensajes del tipo `CoreMessage`.

Al recibir una entrada el método `execute` de `Bot` ejecuta el proceso de comprensión con dicha entrada y el resultado es utilizado como entrada para el proceso de *core*. En la sección sobre el caso práctico mostraremos un fragmento de código con la inicialización de los nodos de *bot* y la construcción del agente con la clase `Bot`. Por lo anterior, en esta sección solo mostraremos un ejemplo de creación de nodos de proceso.

```
1 simple_nlu_process = PipelineProcess(  
2     name="Understanding",  
3     stateful=False,  
4     nodes=[  
5         ProcessNode(  
6             service=IntentService(),  
7             input_mapping={"text": "text"},  
8             output_mapping=["text_intent"],  
9             training_input_mapping={"x_texts": "texts", "y_intents": "intents"}  
10        ),  
11        ProcessNode(  
12            service=CoreMessageBuilderService(),  
13            input_mapping={"text_intent": "intent"},  
14            output_mapping=["core_message"],  
15            training_input_mapping={"text_intent": "intent"}  
16        )  
17    ]  
18 )
```

```

16         )
17     ],
18     output_keys=["core_message"]
19 )

```

Código 66: Ejemplo de creación de proceso desde código.

6.4.1. Integración con Telegram

Nuestro *framework* proporciona una implementación de agente conversacional `TelegramBot` que hereda de la clase `Bot` y extiende su funcionalidad para facilitar la integración del agente con la API de Telegram para la creación de *bots* a los que se puede acceder desde los chats de dicha *app*. La librería utilizada es `simple-telegram-bot` en la versión 13.7.

La inicialización de un agente `TelegramBot` incluye dos campos adicionales: un *token* proporcionado por Telegram que representa al *bot* en su plataforma y una lista opcional `commands` de comandos.

Los mensajes de contenido textual son procesados de la misma forma que un agente convencional, pero hay que tratar ciertos casos especiales en los que un mensaje es un comando o un comando con contenido textual. Los comandos son órdenes directas al proceso *core* que no requieren el proceso de comprensión, excepto si el mensaje incluye un contenido textual. En Telegram los comandos son una subsecuencia de caracteres que empieza por '/'. Por ejemplo, '/help' es un comando.

Al recibir un mensaje desde Telegram, un evento es lanzado en el agente y en función de si se trata de un comando o no, se envía un mensaje `TelegramIncomingMessage` o un mensaje de comando `TelegramCommandMessage`. La primera clase hereda de `BaseIncomingMessage` e incluye atributos para representar el contexto de Telegram. La segunda clase hereda de `TelegramIncomingMessage` e introduce un atributo `command_name`.

Tal y como hemos mencionado anteriormente, un comando *c* con texto adicional *t* será procesado por el proceso de comprensión. A la intención del mensaje *t* se le añadirá un prefijo con el nombre del comando y una barra '\'. Por ejemplo, si la intención de *t* es `greeting` y el comando es `/order_food`, la intención del mensaje compuesto será `order_food\greeting`.

6.5. Caso práctico: Bot COVID-19

Nuestro caso práctico consiste en el diseño e implementación de un agente conversacional capaz de aportar información básica sobre el COVID-19, sus síntomas y las vacunas disponibles para desarrollar inmunidad frente al virus. Además de proporcionar información como un *bot* de preguntas frecuentes, dispondrá de flujos para pedir cita para la vacunación y de rastreo de posibles casos que notifique el usuario. La estructura del proyecto es la siguiente:

- Un directorio `data` con un fichero YAML `entities.yml` con las entidades en su forma canónica y sinónimos o expresiones equivalentes, un directorio `models` con modelos entrenados, un fichero YAML `recall.yml` con recuerdos para el modelo de comportamiento basado en recuerdos, un fichero YAML `keras-dataset.yml` con 1600 historias para entrenar el modelo de comportamiento basado en Keras, y los ficheros YAML `generic_nlu.yml`, `flow_nlu-sm.yml` y `faq_nlu.yml` de conjuntos de datos NLU.
- Un directorio `config` en el que almacenamos los ficheros YAML que definen procesos *pipeline*. En el caso que nos ocupa, solo habrá un fichero para el proceso NLU denominado `nlu-cfg-jupyter.yml`.
- Un módulo `process_nlu.py` para cargar el proceso NLU y poder hacer pruebas con el proceso de comprensión.
- Un módulo `actions.py` con las acciones.
- Desde el cuaderno de Jupyter `3. Agentes.ipynb` puede cargar estos procesos, inicializar el agente y poder desplegarlo en Telegram. Asimismo, se proporciona la opción de inicializar el agente como un *bot* general y el método `say(text)` para comunicarse con él. Una cuestión a tener en cuenta es que el agente de Telegram no se puede inicializar más de una vez en el cuaderno por restricciones de la librería de Telegram. Deberá reiniciar el cuaderno al completo para poder inicializarlo de nuevo.

6.6. Preprocesamiento

Este agente conversacional será capaz de mantener conversaciones en inglés. Se podría haber recurrido a otro idioma como el español, pero queremos comprobar el rendimiento del

agente con los mejores modelos disponibles y los modelos de lenguaje más completos a día de hoy son del inglés. El modelo de *token* utilizado será `SpacyToken`, esto es, utilizaremos el servicio de segmentación de `SpacyTokenizer` para segmentar los mensajes en *tokens* de SpaCy. El modelo de lenguaje utilizado por SpaCy para extraer información y segmentar el texto es `'en_core_web_md'`, uno de los modelos de tamaño medio disponibles para el lenguaje natural seleccionado.

A continuación, describimos cada una de las etapas de preprocesamiento introducidas en el agente:

1. **Normalización del texto a nivel de caracter:** El servicio `TextNormalizer` genera una representación textual normalizada a nivel de caracter del texto recibido por el proceso de comprensión. El texto se transforma a minúsculas, se eliminan las elipsis y se normalizan los símbolos de interrogación y exclamación. Asimismo, se reemplazan símbolos no ASCII como las letras acentuadas por símbolos ASCII similares a los originales.
2. **Segmentación del texto original:** Una instancia de `SpacyTokenizer` segmenta el texto original sin normalizar en *tokens* de SpaCy. Esta secuencia será consumida por servicios NLU de extracción de entidades, puesto que no se alteran propiedades como el *letter case* de las letras.
3. **Segmentación del texto normalizado:** Una instancia de `SpacyTokenizer` segmenta el texto normalizado en *tokens* de SpaCy. Esta secuencia será consumida por servicios de codificación, puesto que al normalizar los textos simplificamos el espacio de caracteres y, por ende, el de *tokens*, por lo que mensajes que difieren en el uso de mayúsculas, acentos, etc. serán codificados y clasificados de la misma forma.
4. **Codificador de secuencias de *tokens*:** Hemos recurrido al servicio de codificación `SpacyEncoder`.

6.7. Clasificación de intenciones

Hemos agrupado el espacio de intenciones del agente en tres grupos: intenciones genéricas, de FAQ y de flujo. Las primeras son independientes del dominio del agente y representan actos comunicativos simples o comunes. Estas intenciones se describen en la tabla 11.

Genéricas	
Intención	Descripción
greeting	Saludo del usuario al agente.
farewell	El usuario se despide del agente.
yes	Respuesta afirmativa.
no	Respuesta negativa.
not_sure	El usuario muestra inseguridad en su respuesta o no recuerda un dato que le ha sido solicitado.
thanks	El usuario da gracias al agente por su trabajo.
didnt_understand	El agente tiene una confianza demasiado baja en su estimación de la intención.

Figura 11: Espacio de intenciones genéricas del agente.

Solicitud de información (FAQ)	
Intención	Descripción
help	El usuario solicita ayuda al agente o le pregunta qué puede hacer para ayudarlo.
what_is	Se solicita una breve definición de una enfermedad o vacuna.
tell_symptoms	Se solicita una lista con los síntomas más comunes.
is_a_symptom	Se pregunta al agente si una o más condiciones físicas pueden ser síntomas de COVID-19.
tell_side_effects	Se solicita una lista con los efectos secundarios más comunes de una vacuna.
is_a_side_effect	Se pregunta al agente si una o más condiciones físicas pueden ser efectos secundarios de una vacuna.

Figura 12: Espacio de intenciones de preguntas FAQ del agente.

De flujos	
Intención	Descripción
say_name	El usuario ha proporcionado el nombre completo o no de un individuo.
say_age	El usuario ha proporcionado la edad de un individuo.
say_year_birth	El usuario ha proporcionado el año de nacimiento de un individuo.
say_id	El usuario ha proporcionado el identificador de ciudadano de un individuo.
say_phone_number	El usuario proporciona su teléfono de contacto.
say_symptom	El usuario proporciona uno o más síntomas y opcionalmente, su duración o el número de días que los ha tenido.
ask_is_full_name	El usuario pregunta si el nombre a proporcionar debe ser el nombre completo o no.
ask_location	El usuario pregunta dónde tiene lugar la prueba.

Figura 13: Espacio de intenciones para los flujos del agente.

El segundo grupo está constituido por intenciones propias del dominio que ni inician ningún flujo ni son necesarias para modelar respuestas a preguntas del agente relacionadas con los flujos. Estas intenciones se describen en la tabla 12.

Finalmente, tenemos un tercer grupo de intenciones a las cuales pertenecen mensajes que inician flujos, proporcionan información para el flujo activo o solicitan información dependiente del flujo activo. En la tabla 13 se describen las intenciones de este último grupo.

Disponemos de tres *datasets* elaborados con la ayuda de *scripts* de generación de ejemplos. Estos *datasets* están en el directorio `data` bajo los nombres `'generic_nlu.yml'`, `'faq_nlu.yml'` y `'flow_nlu-sm.yml'`.

Se recurrirá al servicio `IntentClassifier` para efectuar la clasificación de la intención de los mensajes. El modelo de máquina de soporte vectorial resultante será almacenado en `data/models` bajo el nombre de `'intent_cls.pkl'`. El servicio estará inicializado de tal forma que la salida del mismo será una distribución de probabilidad (i.e. `output_mode = probabilities`) en el

Categoría de entidad	Descripción	Ejemplos
LOCATION	Localización.	Home, hospital.
PHSICAL_CONDITION	Síntoma, efecto secundario o condición física.	Fever, continuous cough, chills.
PHONE_NUMBER	Número de teléfono.	+34 123 456 789
NUMBER	Cantidad numérica.	Two, 12.
TIME_UNIT	Unidad temporal.	Days, week.
TIME_EXPR	Expresión de tiempo.	Today, last week.
ID	Identificación de ciudadano. Es una cadena de uno o más dígitos seguidos de una letra mayúscula o minúscula.	1234Z
NAME	Nombre completo o no.	Lola Llamas Fernández, María.
VACCINE	Nombre comercial de una vacuna.	Pfizer.
DISEASE	Nombre de una enfermedad conocida.	Coronavirus, Covid.

Figura 14: Tipos de entidades soportadas.

espacio de intenciones definida en esta sección.

En el caso de que el servicio haga una predicción con una confianza inferior a un 65 %, se clasificará el mensaje con la intención `'didnt_understand'`. Con el objeto de mejorar la calidad del modelo se aplica una búsqueda en un espacio de hiperparámetros durante el entrenamiento.

6.7.1. Entidades

A partir de las intenciones definidas podemos deducir los tipos de entidades necesarios. Las intenciones de solicitud de información requieren al menos dos tipos de entidades: enfermedades y condiciones físicas. Con condiciones físicas abarcamos tanto a los posibles síntomas como a los efectos secundarios de las vacunas.

Los dos flujos del agente deben trabajar con símbolos que representen nombres, edades, años de nacimiento, identificadores, teléfonos, localizaciones y síntomas con su respectiva duración.

El primer paso es definir las clases de entidades a las que se limitará el dominio del agente. En la tabla 14 describimos y aportamos ejemplos de todos los tipos de entidades esperados.

El segundo paso es definir un listado con las formas canónicas de las entidades soportadas y sus respectivas expresiones equivalentes o sinónimos. Por ejemplo, una forma alternativa de referirse a 'fever' puede ser 'high body temperature'. En el proceso de comprensión incluimos dos nodos computacionales con los servicios `NamedEntitiesSpellChecker` y `SynonymsNormalizer` que harán pequeñas correcciones en las entidades mencionadas por el usuario si este envió un mensaje con alguna errata y normalizarán las entidades soportadas.

En `data/entities.yml` introducimos una clave por cada forma canónica y le asociamos una lista posiblemente vacía de alternativas. Aunque esta lista pueda llegar a estar vacía para una forma canónica α , es necesaria incluirla en el fichero para que el servicio de corrección de erratas tenga en cuenta a α en su vocabulario.

El tercer paso es examinar si alguna entidad puede ser detectada a través de una expresión regular a nivel de *token*. En caso afirmativo, debemos añadir dicha expresión como una característica más del servicio *featurizer* de nuestro proceso. En el caso que nos ocupa, hemos definido para el identificador de ciudadano una expresión regular $[0-9]^+[A-Za-z]$. En el caso de los números de teléfono tenemos que percatarnos de que estas expresiones son a nivel de *token* y un usuario puede introducir el número por grupos de dígitos de dos, tres o más dígitos. Delegamos en el servicio `IOBTagger` de extracción de etiquetas de entidades la responsabilidad de deducir el patrón a nivel de secuencia de *tokens*.

El último paso es decidir si es necesario, además del servicio `IOBTagger` para la extracción de entidades, una instancia del servicio de extracción de roles y grupos. La intención 'say_symptom' puede obligarnos a introducir la detección de roles y la separación en grupos de entidades. Supongamos que el usuario ha introducido el siguiente texto:

My son had fever for three days and headaches, but these symptoms went away. He has a cough since last night and chills since Monday.

El proceso *core* debe ser capaz de determinar si un síntoma está todavía presente o no. Otro aspecto a tener en cuenta es que la duración de un síntoma se puede expresar de varias formas.

Un usuario puede especificar un número de días o semanas, o bien utilizar una referencia temporal como el día de ayer, la semana pasada o la noche del día anterior.

Hemos decidido crear 5 roles para ser capaces de disponer de esta información en el proceso *core*. Para determinar si un síntoma persiste o no, etiquetaremos a la entidad del síntoma con el rol `'symptom_now'` o `'symptom_before'`. Para las expresiones temporales definimos tres roles: `'time_value'`, `'time_unit'` y `'time_expr'`. Las dos primeras serán etiquetas de números y palabras que designen una unidad temporal como los días o semanas. La tercera y última será utilizada para etiquetar expresiones no numéricas que definan un periodo de tiempo. Al tratar con un reducido conjunto de formas canónicas, el proceso *core* puede asociar a estas expresiones una fecha.

6.8. Lógica del agente

6.8.1. Modelo de comportamiento basado en reglas

En un fichero `rules.py` definimos listas de reglas para algunas de las intenciones definidas en el modelo de intenciones del agente. El modelo de comportamiento basado en reglas tiene mayor capacidad expresiva que el basado en recuerdos, por lo que hemos recurrido a él para ciertos comportamientos que requieren de más información sobre el estado. Haciendo uso de reglas hemos definido el siguiente comportamiento básico:

1. **Saludos:** Cuando el usuario saluda al agente por primera vez (i.e. la intención es `'greeting'` y es el primer mensaje recibido en la conversación), se debe saludar al usuario y proporcionarle una información básica de ayuda para que sea consciente de qué puede hacer el *bot*. Si no es la primera vez que el usuario se ha comunicado con el agente, solo se le saludará, pero no se repetirá nuevamente la información de ayuda a no ser que el usuario la solicite de nuevo.
2. **Despedidas:** Cuando el usuario se despide (i.e. la intención es `'farewell'`) debemos tener en cuenta si hay un flujo activo o no. En el caso de que lo hubiera, habría que solicitar una confirmación de finalización. En otro caso, el agente se despide del usuario.
3. **Inseguridad en la respuesta:** La intención `'not_sure'` será determinante en el caso de que el usuario se encuentre en un flujo activo. Dado que los campos solicitados por el

agente de nuestro caso práctico no son opcionales, debemos informar al usuario de que se finaliza el flujo por no disponer de la suficiente información. Si no estamos en un flujo, ignoramos el mensaje.

4. **Respuestas de afirmación, negación y mensajes de agradecimiento:** Los mensajes de agradecimiento siempre son ignorados. Las respuestas de afirmación o negación serán ignoradas si tienen lugar fuera de un flujo. Por ejemplo, si al informar de los síntomas de COVID-19, el usuario responde 'Ok, got it', se interpretará como una respuesta afirmativa y será ignorada al no haber un flujo activo.
5. **Inseguridad en la clasificación de intenciones:** Si la intención es 'didnt_understand', informamos al usuario de que no le hemos entendido.
6. **Petición de ayuda:** Mostramos al usuario una breve descripción de lo que puede hacer el agente.
7. **Intenciones de solicitud e información:** Al recibir una petición de información, ejecutamos las acciones que consisten en proporcionar esa información, comprobar si un síntoma está vinculado a una enfermedad o un efecto secundario a una vacuna.
8. **Definiciones:** Disponemos de dos reglas que inspeccionan la categoría de las entidades recibidas para decidir si hay que reaccionar describiendo un virus o una vacuna. Asimismo, se introduce una tercera regla que en el caso de no encontrar ninguna entidad con la categoría DISEASE o VACCINE, notifica al usuario que o ha solicitado información que desconoce el agente, o bien no ha sido capaz de entender su mensaje.
9. **Inicio de flujos:** Las dos intenciones que inician flujos son 'make_appointment' y 'feeling_unwell', por lo que definimos dos reglas para iniciar los flujos correspondientes al recibir un mensaje con estas intenciones. Un caso especial a contemplar es que ya haya un flujo activo. En este caso deberá preguntarse al agente si quiere cancelar el anterior y empezar un nuevo flujo.

6.9. Modelo de comportamiento basado en flujos

En este agente disponemos de un flujo para pedir cita para recibir la vacuna y otro de rastreo para notificar que se tienen síntomas. El flujo para pedir cita se denomina `'make_appointment'` y tiene los siguientes pasos:

1. Se informa al usuario de que se va a iniciar el flujo.
2. Se pregunta al usuario la edad de la persona que va a vacunarse. Introducimos una validación para que no se acepten peticiones de cita a menores de 12 años. Esta condición la hemos impuesto para demostrar cómo funciona la validación de campos.
 - Si el usuario proporciona el año en el que nació, se aplica una transformación previa a la validación para calcular los años del sujeto. Si el usuario responde que es del 88 o 1988, por ejemplo, se aplica la transformación al detectarse que la intención no es `'say_age'` sino `'say_year_birth'`.
 - Si la validación no es superada, se informa al usuario y se finaliza el flujo.
3. Se pregunta al usuario el nombre de la persona que va a vacunarse.
4. Se pregunta al usuario el identificador de ciudadano. En un entorno real se comprobaría la identidad a partir de esta información y la anterior.
5. Se pregunta al usuario si la persona que se va a vacunar es alérgica o no. Utilizamos un *slot* de confirmación. Si el usuario no está seguro, se considera el peor caso, esto es, que sea alérgico.
 - Si se responde afirmativamente, se le informa al final del flujo de que deberá quedarse 30 minutos en el centro de salud tras la vacunación.
6. Se propone un día y hora para la cita. Se utiliza un segundo *slot* de confirmación. Si el usuario no está seguro, se considera que ha rechazado la cita y se le ofrece una la siguiente semana. En el caso de que se acepte, continuamos con el flujo.
7. Se informa al usuario de que la cita ha sido planificada. Finaliza el flujo.

Los *slots* para la edad e identificador de ciudadano son los más simples. Las condiciones para que se rellenen se limitan a que las entidades tengan los tipos NAME y ID, y que las intenciones sean 'entity', o 'say_name' o 'say_id' según sea uno o el otro.

```
1 slot_name = FlowBotSlotDefinition(  
2     name="make_appointment/name",  
3     order=2,  
4     in_messages_with_intents=["entity", "say_name"],  
5     category="NAME",  
6     reaction=["action_ask_name"],  
7 )  
8  
9 slot_id FlowBotSlotDefinition(  
10    name="make_appointment/id",  
11    order=3,  
12    in_messages_with_intents=["entity", "say_id"],  
13    category="ID",  
14    reaction=["action_ask_id"],  
15 )
```

Código 67: Definición de *slots* de flujo para el nombre e identificador de ciudadano en el flujo de pedir cita.

El *slot* para la edad presenta una mayor complejidad al ser necesario introducir una transformación y validación. Tal y como podemos apreciar en el siguiente fragmento de código, se introduce una reacción para el caso en el que el dato no sea válido. A través del atributo `reaction_on_invalid_value` se introduce la funcionalidad de informar al usuario de que no tiene la edad suficiente.

```
1 def transform_age(state: BotInternalState, slot_def: FlowBotSlotDefinition, value: str):  
2     if state.get_intent() == "say_year_birth":  
3         year = int(value)  
4  
5         if len(value) == 2:  
6             year += 1900  
7  
8     return str(datetime.datetime.today().year - year)
```

```

9     else:
10         return str(value)
11
12
13 def check_age_group(state: BotInternalState, slot_def: FlowBotSlotDefinition, value: str):
14     try:
15         age = int(value)
16         return age >= 12 # Mock
17     except ValueError:
18         return False
19
20
21 slot_age = FlowBotSlotDefinition(
22     name="make_appointment/age",
23     order=1,
24     in_messages_with_intents=["entity", "say_age", "say_year_birth", "make_appointment"],
25     category="NUMBER",
26     reaction=["action_ask_age"],
27     validate=check_age_group,
28     transform=transform_age,
29     reaction_on_invalid_value=["action_appointment_invalid_age", "flow_finish"]
30 )

```

Código 68: Definición del *slot* de flujo para la edad con transformación y validación.

Los dos *slots* restantes son de confirmación. El *slot* de confirmación de flujo es similar al de si es alérgico o no, con la diferencia de que se considera relleno solo si se aporta una respuesta afirmativa.

```

1 FlowConfirmationBotSlotDefinition(
2     name="make_appointment/confirmation",
3     order=5,
4     intents_accept=["yes"],
5     intents_reject=["no"],
6     first_reaction=["action_generate_date", "action_suggest_date"],
7     reaction_accept=["action_appointment_ok"],
8     reaction_reject=[
9         "action_appointment_not_ok",

```



```

10     "action_generate_date",
11     "action_suggest_date"
12 ],
13 enabled=lambda state: state.get_current_flow_empty_slots() == \
14 ["make_appointment/confirmation"],
15 filling_mode=FlowConfirmationBotSlotFillingMode.COMPLETED_IF_ACCEPTED
16 )

```

Código 69: Definición del *slot* de flujo para la confirmación de la fecha propuesta para la cita.

El flujo de rastreo de casos es similar al anterior, puesto que solicita la edad, el nombre y el identificador de ciudadano. Además de estos campos, solicita el número de teléfono y una lista de síntomas. Este último campo se define como un `FlowComplexBotSlotDefinition` por la naturaleza de la información que vamos a rellenar para ese *slot*. A través de la función de transformación se filtran los grupos de entidades que no tienen una entidad de categoría `PHYSICAL_CONDITION`. El validador simplemente comprueba que hay al menos un grupo de entidades de este tipo en el mensaje.

```

1 FlowComplexBotSlotDefinition(
2     name=feeling_unwell_slots["symptoms"],
3     order=5,
4     in_messages_with_intents=["entity", "say_symptom"],
5     transform=transform_symptoms,
6     validate=check_symptoms,
7     reaction_on_invalid_value=["action_invalid_symptoms"],
8     reaction=["action_ask_symptoms"]
9 )

```

Código 70: Definición del *slot* de flujo para la lista de síntomas en el flujo de rastreo de casos.

6.10. Modelos de comportamiento basado en modelos estadísticos y recuerdos

Si bien la mayor parte de la lógica del agente la hemos podido modelar con modelos de comportamiento de carácter determinista, es necesario introducir un modelo capaz de lidiar

con los casos que no esperamos o que por su complejidad son más fáciles de implementar a través de aprendizaje automático.

Para utilizar el modelo de comportamiento basado en modelos estadísticos, hemos definido dos *slots* globales para las vacunas y las enfermedades, y hemos configurado el *featurizer* del contexto de la conversación como sigue:

```
1 featurizer = StateFeaturizerService(  
2     intents=[  
3         "greeting", "farewell", "yes", "no", "not_sure", "thanks",  
4         "help", "what_is", "tell_symptoms", "is_a_symptom",  
5         "tell_side_effects", "is_a_side_effect",  
6         "make_appointment", "feeling_unwell", "say_name",  
7         "say_age", "say_year_birth", "say_id", "say_phone_number",  
8         "ask_is_full_name", "ask_location", "say_symptom",  
9         "entity"  
10    ],  
11    slots=["vaccine", "disease"],  
12    actions=actions # definido anteriormente (lo omitimos para reducir la extension)  
13 )
```

Código 71: Inicialización del *featurizer* del contexto de la conversación.

Los *slots* globales podrán ser sobrescritos en cada intervención. Los hemos definido como sigue:

```
1 vaccine_slot = BotSlotDefinition(  
2     name="vaccine",  
3     category="VACCINE",  
4     in_messages_with_intents=[  
5         "what_is", "entity", "tell_side_effects", "is_a_side_effect"  
6     ]  
7 )  
8  
9 disease_slot = BotSlotDefinition(  
10    name="disease",  
11    category="DISEASE",  
12    in_messages_with_intents=[  
13        "what_is", "entity", "tell_symptoms", "is_a_symptom"
```

```
14 ]
15 )
```

Código 72: Definición de *slots* globales.

El fichero `'data/keras-dataset.yml'` es un conjunto de datos de experiencias con las que se entrenará el modelo predeterminado de Keras, es decir, una red neuronal con una capa de *masking* para ignorar vectores de estado de *padding*, una capa LSTM de 100 unidades y una capa de salida con una función de activación *softmax*. Limitaremos la capacidad del histórico del agente a 5 entradas.

Una de las capacidades de las que queremos dotar al agente es la de deducir que un usuario está repitiendo la pregunta anterior, pero cambiando el objeto sobre el que recae la pregunta. Por ejemplo, si un usuario ha preguntado si la fiebre es un síntoma de COVID y, tras la respuesta afirmativa del agente, el usuario dice 'and what about cough?', el agente debe ser capaz de deducir que el mensaje con la intención `'entity'` debe ser interpretado como un mensaje con la intención `'is_a_symptom'`. De hecho, debería ser posible aplicar el mismo tratamiento a un tercer mensaje 'ok and chills?'. Si bien podríamos modelar este comportamiento con reglas, es más sencillo proporcionar experiencias donde tiene lugar este fenómeno y entrenar el modelo de comportamiento Keras.

Además de este modelo hemos recurrido a un modelo basado en recuerdos que, al igual que el de Keras, inspeccionará el contexto y proporcionará predicciones más fiables en aquellos casos en los que el modelo de Keras no haga predicciones correctas.

6.11. Acciones

En el fichero `actions.py` hemos definido las funciones asociadas a los símbolos de acciones. Hemos utilizado el mismo identificador para facilitar el desarrollo. Las respuestas a los mensajes enviados desde Telegram son enviados exclusivamente por el canal de salida al *chat* de Telegram. El resto de mensajes son enviados en *broadcast*.

6.12. Construcción del *bot*

Una vez definidos los procesos de comprensión y *core*, debemos construir los nodos computacionales de cada proceso. A través de la función `load_nlu` del módulo `process_nlu.py` cargaremos la instancia entrenada del proceso NLU.

El proceso de comprensión espera recibir un texto denominado `'text'` y el mensaje original `'original_message'`. En el caso que nos ocupa, hemos elegido los mismos identificadores, por lo que el *mapping* es la identidad. En el proceso *core* hemos tomado la misma decisión. El mensaje *core* será generado en el proceso de comprensión con el identificador `'message'`, por lo que, al coincidir con el nombre esperado por el proceso *core* para este mensaje, el *mapping* es nuevamente la identidad.

```
1 understanding_node = BotNode(  
2     process=load_nlu(  
3         project_path=os.path.join(os.pardir, "chatbots", "covid"),  
4         config_file_name="nlu-cfg-jupyter.yml",  
5         train=False,  
6         log_level=logging.DEBUG  
7     ),  
8     input_mapping={"text": "text", "original_message": "original_message"},  
9     training_input_mapping={"X_texts": "x_texts", "Y_intents": "y_intents"}  
10 )  
11  
12 core = PipelineProcess(  
13     nodes=[  
14         ProcessNode(  
15             service=core_pp,  
16             input_mapping={  
17                 "message": "message",  
18                 "state": "state"  
19             },  
20             output_mapping=["state"]  
21         ),  
22         ProcessNode(  
23             service=logic_service,  
24             input_mapping={"state": "state"},  
25             output_mapping=[]
```

```

26     )
27 ],
28     name="Core",
29     stateful=True
30 )
31
32 core_node = BotNode(
33     process=core,
34     input_mapping={"message": "message"}
35 )

```

Código 73: Creación de nodos computacionales para los procesos cognitivos del agente.

El agente se puede construir como una instancia de `Bot` con los dos nodos computacionales anteriores.

```

1 bot = Bot(
2     name="BOT-COVID",
3     understanding_node=understanding_node,
4     core_node=core_node,
5     log_level=logging.DEBUG
6 )

```

Código 74: Creación de nodos computacionales para los procesos cognitivos del agente.

Finalmente, para un proyecto de consola podemos crear un bucle infinito de recepción de mensaje y ejecución del agente con la información que espera recibir el proceso de comprensión, esto es, el texto y el mensaje original.

```

1 while True:
2     text = input("User: ")
3     text = text.strip()
4     if text == "":
5         continue
6
7     message = BaseIncomingMessage(
8         channel_id="CONSOLE",

```

```

9         text=text,
10        timestamp=datetime.datetime.now()
11    )
12    bot.execute(text=message.text, original_message=message)

```

Código 75: Código principal de un agente conversacional de consola.

Si queremos desplegar el agente en Telegram, debemos crear el *bot* en esta aplicación y con el *token* que se nos proporciona inicializamos una instancia de `TelegramBot`.

```

1  from smartbot.arch.bot.telegram_bot import TelegramBot
2
3  bot = TelegramBot(
4      token="<your_token_here>"
5      name="BOT-COVID",
6      understanding_node=understanding_node,
7      core_node=core_node,
8      log_level=logging.DEBUG,
9      updater=updater,
10     commands=["start", "help", "make_appointment", "feeling_unwell"]
11 )
12
13 bot.run() # polling

```

Código 76: Inicialización y ejecución de un agente conversacional de Telegram.

Si hemos definido una lista de comandos al crear el *bot* en Telegram, deberemos especificar el nombre de estos comandos a través del parámetro `commands`. Finalmente, para ejecutar este tipo de agentes debemos llamar al método `run()` para hacer *polling* de mensajes.

6.13. Ejemplo de conversación completa

Para concluir este capítulo mostramos una conversación completa con el agente dividida en dos partes. En la segunda parte, el usuario es el mismo que en la primera y el agente es consciente de que no es la primera vez que interactúan. En el resto de archivos adjuntos del trabajo se incluyen más ejemplos y cuadernos de Jupyter con ejemplos de uso de servicios, procesos y este *bot*.

USUARIO: Hii!! Is anyone here??

CHATBOT: Hello!

CHATBOT: I was designed to help you by providing you all the basic information you need. If you want to book your vaccination or tell me one of your relatives or you do not feel well, here I am.

USUARIO: could you put coronavirus in layman's terms?

CHATBOT: Coronavirus disease (COVID-19) is an infectious disease caused by the SARS-CoV-2 virus.

Most people infected with the virus will experience mild to moderate respiratory illness and recover without requiring special treatment. However, some will become seriously ill and require medical attention. Older people and those with underlying medical conditions like cardiovascular disease, diabetes, chronic respiratory disease, or cancer are more likely to develop serious illness. Anyone can get sick with COVID-19 and become seriously ill or die at any age.

Source: WHO (https://www.who.int/health-topics/coronavirustab=tab_1)

USUARIO: it sounds interesting

CHATBOT: [*Ignora el mensaje*]

USUARIO: ok what about pfizer? can you tell me more about this vaccine?

CHATBOT: The Pfizer vaccine (Comirnaty) is an mRNA vaccine that contains the genetic code for an important part of the SARS-CoV-2 (COVID-19) virus called the 'spike protein'. Spike proteins are the little projections on the surface of the virus. Both doses of the Pfizer vaccine (Comirnaty) are the same. The second dose increases your protection – giving you better and likely longer-lasting immunity than the first dose alone.

Source: <https://www.health.govt.nz/our-work/diseases-and-conditions/covid-19-novel-coronavirus/covid-19-vaccines/covid-19-how-vaccine-works>

USUARIO: they were talking about astrazeneca on the tv. what is it?

CHATBOT: [*Vaxzevria es el nombre correcto de la vacuna de esta compañía*] Vaxzevria is a vaccine for preventing coronavirus disease 2019 (COVID-19) in people aged 18 years and older. COVID-19 is caused by SARS-CoV-2 virus. Vaxzevria is made up of another virus (of the adenovirus family) that has been modified to contain the gene for

making a protein from SARS-CoV-2.

Source: <https://www.ema.europa.eu/en/medicines/human/EPAR/vaxzevria-previously-covid-19-vaccine-astrazeneca>

USUARIO: i was wondering if you have information about the russian vaccine sputnik

CHATBOT: Sorry, we do not have information about this

USUARIO: let's assume i will receive moderna, what are the potential side effects of this vaccine?

CHATBOT: Common side effects can include:

- Pain in the arm
- Redness in the arm
- Swelling in the arm
- Fever
- Tiredness
- Headache
- Muscle pain
- Chills
- Nausea

Source: WHO

USUARIO: alright

CHATBOT: [*Ignora el mensaje*]

USUARIO: rash, joint pain and fever are side effects?

CHATBOT: Yes, fever is one of the common side effects. We do not have information about the rest of physical conditions you mentioned

USUARIO: hmm i think it's time to book an appointment for my vaccination

CHATBOT: [*Inicia flujo*] Great, let's book an appointment for vaccination. How old is the person who will receive the vaccine?

USUARIO: I was born on March 8, 1990

CHATBOT: 31 years old. What is his/her name?

USUARIO: wait do you mean my full name??

CHATBOT: We will ask you for the ID later, so you can give us just the first name if you want.

USUARIO: ok my full name is Manuel Lopez Reviriego.

CHATBOT: What is his/her citizen id?

USUARIO: 123456Z

CHATBOT: Is he/she allergic?

USUARIO: yes

CHATBOT: The appointment would be scheduled for 21/09/2021, 13:34. Is it good for you?

USUARIO: sorry i will be so busy that day.

CHATBOT: Alright, I will suggest you another date for your vaccination. The appointment would be scheduled for 28/09/2021, 13:34. Is it good for you?

USUARIO: yeah good for me

CHATBOT: [*Finaliza flujo*] The appointment was scheduled for 28/09/2021, 13:34. Because of being allergic, it is required to stay at the hospital for at least 30 minutes.

USUARIO: ok thanks you were so helpful see you later!

CHATBOT: Bye!

[*El usuario finaliza la conversación, pero vuelve más tarde a comunicarse con el agente.*]

USUARIO: hello again...

CHATBOT: Hello!

USUARIO: my father is sick as a dog, i think it is covid-19...

CHATBOT: [*Inicia flujo*] We are sorry to hear that. We will need some information to track the case. How old is he/she?

USUARIO: he is 56 years old

CHATBOT: OK, 56 years old. What is his/her name?

USUARIO: Manuel Lopez

CHATBOT: What is his/her citizen id?

USUARIO: 1333A

CHATBOT: Can you provide us a contact number?

USUARIO: yeah my phone number is 999 999 999

CHATBOT: Which symptoms?

USUARIO: my father had headache but now he has continuous cough since Tuesday

CHATBOT: [*Finaliza flujo*] Ok, thank you. We will contact you by the phone number you provided: 999 999 999.

For now, Manuel Lopez should stay at home for at least 15 days, because cough is a sign of COVID-19.

[*El usuario finaliza la conversación, pero vuelve más tarde a comunicarse con el agente desde Telegram.*]

USUARIO: hello again

CHATBOT: Hello!

USUARIO: /feeling_unwell

CHATBOT: [*Inicia flujo*] We are sorry to hear that. We will need some information to track the case. How old is he/she?

USUARIO: well first I want an appointment for my son

CHATBOT: Do you want to finish the current flow?

USUARIO: yes, go ahead

CHATBOT: How old is the person who will receive the vaccine?

USUARIO: He is 8 years old

CHATBOT: You cannot book any appointments right now. You will have to wait for 3 weeks

USUARIO: Oh, I didn't know that. Thank you

CHATBOT: [*Ignora el mensaje*]

USUARIO: That was all, see you later

CHATBOT: Bye!

7

Conclusiones y Líneas Futuras

7.1. Conclusiones

A modo de cierre de este trabajo, podemos concluir que la teoría de agentes conversacionales introducida en el capítulo 3 es lo suficientemente completa para describir, utilizando un mismo marco teórico, los distintos agentes conversacionales que están disponibles en el mercado. Prueba de ello es que podemos definir e implementar una amplia selección de comportamientos complejos mediante el *framework* en Python que hemos desarrollado a partir de nuestra teoría.

Otro aspecto a destacar del sustrato teórico de nuestro *framework* es que las hipótesis y suposiciones que hicimos en las primeras etapas del trabajo no solo no han supuesto una reducción ostensible de las capacidades cognitivas de los agentes desarrollados, sino que además han facilitado el desarrollo de los agentes. Por lo anterior, podemos concluir empíricamente la validez y utilidad de nuestras suposiciones e hipótesis.

En cuanto a los procesos de comprensión, queremos señalar que tras realizar un estudio minucioso de las técnicas utilizadas a día de hoy para comprender el lenguaje natural, llegamos a la conclusión de que este problema está parcialmente resuelto. No obstante, debemos mencionar que todavía existe una diferencia notable en el volumen de datos y modelos pre-entrenados disponibles para el inglés y el español. Por ello, es previsible que algunos sistemas NLU para el inglés tengan un mejor rendimiento que otros sistemas NLU entrenados para el castellano.

En lo que al comportamiento respecta, hemos de decir que si bien la complejidad aparente del proceso de comprensión es superior a la de la lógica de un agente conversacional,

debemos señalar que durante el desarrollo del proyecto nos percatamos de que definir un comportamiento correcto requiere una definición más compleja en la que intervienen factores culturales, sociales y el grado de exposición del agente al mundo exterior. Cuanto más abarque el dominio del agente, más palpable será la complejidad inherente de la tarea de diseñar un comportamiento racional esperado ante los estímulos presentes.

Un aspecto que consideramos conveniente mencionar es que la arquitectura de servicios y procesos cognitivos utilizada, así como la *suite* de servicios cognitivos ofrecidos por nuestro *framework*, permite la construcción de agentes conversacionales con un grado de personalización elevado en un periodo de tiempo de desarrollo breve. No obstante, debemos señalar que, al igual que con otros *frameworks*, el grado de personalización elevado conlleva una curva de aprendizaje más pronunciada. Uno de los objetivos futuros de nuestra librería es proporcionar la misma flexibilidad, reduciendo o simplificando la configuración de los componentes.

7.2. Líneas Futuras

7.2.1. Intenciones múltiples

Una de las características que sería interesante añadir en un futuro sería la posibilidad de detectar un conjunto de subintenciones en un mensaje, en lugar de una intención general.

En los *frameworks* que hemos consultado no se suele incluir la posibilidad de entrenar los servicios de comprensión para extraer más de una intención. Dado que las soluciones empresariales de agentes conversacionales no suelen crear agentes de dominio general, sino agentes basados en tareas, es comprensible que se opte por extraer una intención general.

7.2.2. Diseño antropomórfico y vínculo con el usuario

A día de hoy los usuarios pueden ejecutar tareas u obtener respuestas a preguntas a través de un agente conversacional que tiene una personalidad e identificación compartida entre todos los consumidores del servicio. Es probable que en un futuro próximo los usuarios quieran tener una experiencia más única con su asistente personal. En otras palabras, se debe tener como objetivo que el trato sea lo más personalizado posible para que el usuario no perciba que está siendo tratado como otro cliente más.

El diseño antropomórfico tiene por objeto incrementar la sensación de presencia psicoló-

gica, es decir, tiene como objetivo que el usuario sienta, al menos de forma subconsciente, que no está comunicándose con un dispositivo que le da un trato automatizado. Algunas de las técnicas que habría que estudiar si tienen un impacto positivo en la experiencia del usuario:

- **Generación de una identidad para el agente:** Una primera técnica fácil de implementar sería la posibilidad de generar o elegir parámetros como el nombre y género del agente. En agentes conversacionales de propósito general es posible definir más parámetros que den lugar a una personalidad más compleja. Por ejemplo, el *bot* PARRY que mencionamos en el capítulo del estado del arte tiene una personalidad paranoide y cada vez que el usuario menciona palabras como 'mafia' o 'policía' se puede percibir un cambio en el tono del agente.
- **Parametrización del proceso de decisión:** A ciertos rasgos de personalidad se le puede dar una interpretación probabilística. Por ejemplo, un agente puede optar de forma aleatoria por explotar el conocimiento que tiene de un proceso o explorar el espacio de acciones en búsqueda de mejores estrategias. Podemos introducir un parámetro $r \in [0, 1]$ que se interprete como la probabilidad de utilizar la estrategia de exploración. A mayor valor de r , más arriesgará el agente en sus decisiones. A menor valor de r , más conservador será y aprovechará más el conocimiento del que ya dispone.
- **Simulación de sistema emocional:** Otra posible técnica también utilizada por PARRY es la de introducir en el estado interno del agente un vector de estado emocional. Estas variables pueden influir en el proceso de decisión, de generación de lenguaje natural e incluso en el motor de síntesis de voz. Por ejemplo, una versión moderna de PARRY podría incrementar la velocidad en la que el sintetizador de voz emite el sonido, si el valor de la variable ANXIETY de su estado emocional es elevado.

Además del diseño antropomórfico, habría que estudiar un segundo conjunto de técnicas basadas en modelos estadísticos del *agente* que modelan el mundo. En este trabajo hemos descrito el comportamiento de los agentes como una reacción a un estímulo. Una cuestión que deberíamos plantearnos es el impacto que tendría en el rendimiento de un agente el hecho de que disponga información sobre la probabilidad de que reciba un estímulo con ciertas características.

Supongamos, por ejemplo, que un agente tiene una distribución de probabilidad para el sentimiento de los mensajes con un determinado usuario. Al recibir un estímulo, el agente puede usar esta distribución de probabilidad para etiquetar un estímulo como inesperado o no. Si el usuario siempre se comunica con mensajes con un sentimiento neutral o positivo, y empieza a comunicarse con el *bot* con mensajes con un sentimiento negativo, el agente puede suponer que algo le ha ocurrido al usuario, puede preguntarle y si está en lo cierto, se establecerá un vínculo más cercano entre el usuario y el *bot*.

7.2.3. Abstracción de estímulos

Otra funcionalidad que sería interesante abordar sería la creación de símbolos a partir de estímulos. Los símbolos serían estructuras mentales que consumirían servicios de alto nivel. Por ejemplo, un servicio de planificación de alto nivel, en contraste con algunos de los modelos de comportamientos expuestos en el capítulo del proceso *core*, utilizaría representaciones simbólicas del contexto del diálogo para decidir una reacción.

A través de un mecanismo de abstracción, un estímulo constituido por k campos puede generar un conjunto Ω de símbolos que pueden ser utilizados por árboles de decisión, algoritmos de inteligencia artificial simbólica, etc. Este mecanismo en combinación con la distribución de probabilidad sobre los estímulos, nos permitiría aplicar una selección de campos relevantes (i.e. atención selectiva) previa a la generación de símbolos.

Habría que plantearse en un futuro el estudio de las propiedades de un sistema como el descrito en esta sección. Podría ser de utilidad para crear modelos de comportamientos basados en motores de inferencia lógica.

7.2.4. Estímulos complejos e internos

Nuestro trabajo se ha centrado en el desarrollo de agentes conversacionales que se comunican principalmente por mensajes en lenguaje natural. En un futuro próximo debería incluirse la posibilidad de combinar información visual con información textual. Por ejemplo, si un usuario pregunta cuántas personas salen en una foto, el agente debe ser capaz de aplicar un procesamiento en paralelo a la imagen y al texto.

Otro aspecto que trataremos será la posibilidad de que un agente se envíe estímulos a sí mismo. Desde un punto de vista teórico sería interesante estudiar los mecanismos que permi-

ten el diálogo interno.

Referencias

- Abbassi, E., Blanchette, I., Ansaldo, A. I., Ghassemzadeh, H., y Joannette, Y. (2015). Emotional words can be embodied or disembodied: the role of superficial vs. deep types of processing. *Frontiers in Psychology*, 6. doi: 10.3389/fpsyg.2015.00975
- Acemoglu, D., y Restrepo, P. (2018, January). *Artificial intelligence, automation and work* (Working Paper n.º 24196). National Bureau of Economic Research. Descargado de <http://www.nber.org/papers/w24196> doi: 10.3386/w24196
- Acuña, E. (2011). Preprocessing in data mining. En M. Lovric (Ed.), *International encyclopedia of statistical science* (pp. 1083–1085). Berlin, Heidelberg: Springer Berlin Heidelberg. Descargado de https://doi.org/10.1007/978-3-642-04898-2_51 doi: 10.1007/978-3-642-04898-2_51
- Adankon, M. M., y Cheriet, M. (2009). Support vector machine. En S. Z. Li y A. Jain (Eds.), *Encyclopedia of biometrics* (pp. 1303–1308). Boston, MA: Springer US. Descargado de https://doi.org/10.1007/978-0-387-73003-5_299 doi: 10.1007/978-0-387-73003-5_299
- Akhundov, A., Trautmann, D., y Groh, G. (2018, 08). *Sequence labeling: A practical approach. Alexa voice interaction models*. (s.f.). <https://developer.amazon.com/en-US/docs/alexa/ask-overviews/voice-interaction-models.html>, note = Accessed: 2021-08-21.
- Amazon alexa voice ai*. (s.f.). <https://developer.amazon.com/en-US/alexa>, note = Accessed: 2021-08-21.
- Amazon skills kit*. (s.f.). <https://developer.amazon.com/en-US/docs/alexa/ask-overviews/what-is-the-alexa-skills-kit.html>, note = Accessed: 2021-08-21.
- Ayvaz, S., y Shiha, M. (2017, 01). The effects of emoji in sentiment analysis. *International Journal of Computer and Electrical Engineering*, 9, 360-369. doi: 10.17706/IJCEE.2017.9.1.360-369
- Bedekar, M., Saini, T., y Zahoor, S. (2016, 08). Analysing human feelings by affective computing - a survey.. doi: 10.1109/ICCUBEA.2016.7860017
- Belz, A. (2007, abril). Probabilistic generation of weather forecast texts. En *Human language technologies 2007: The conference of the north American chapter of the association for computational linguistics; proceedings of the main conference* (pp. 164–171). Roches-

- ter, New York: Association for Computational Linguistics. Descargado de <https://www.aclweb.org/anthology/N07-1021>
- Bird, S., Klein, E., y Loper, E. (2009). *Natural language processing with python*. O'Reilly.
- Bobrow, D. (1964, 01). Natural language input for a computer problem solving system.
- Bohnet, B., y Dale, R. (2005, 01). Viewing referring expression generation as search. En (p. 1004-1009).
- Bolton, T., Dargahi, T., Belguith, S., Al-Rakhami, M. S., y Sodhro, A. H. (2021). On the security and privacy challenges of virtual assistants. *Sensors*, 21(7), 2312. doi: 10.3390/s21072312
- Bot framework composer concepts*. (s.f.). <https://docs.microsoft.com/en-us/composer/concept-dialog?tabs=v2x>, note = Accessed: 2021-08-21.
- Bot framework composer dialogs*. (s.f.). <https://docs.microsoft.com/en-us/composer/concept-dialog?tabs=v2x>, note = Accessed: 2021-08-21.
- Brady, M. C., y Berwick, R. C. (1989). *Computational models of discourse*. The MIT Press.
- Burgin, M., y Dodig-Crnkovic, G. (2009). A systematic approach to artificial agents. *CoRR*, abs/0902.3513. Descargado de <http://arxiv.org/abs/0902.3513>
- Clark, A., Fox, C., y Lappin, S. (2013). *The handbook of computational linguistics and natural language processing*. Wiley-Blackwell.
- Collins. (s.f.). *Lexicon*. Descargado 2021-05-12, de <https://www.collinsdictionary.com/es/diccionario/ingles/lexicon>
- Conversation*. (s.f.). Descargado 2021-05-17, de <https://www.macmillandictionary.com/dictionary/british/conversation>
- Cucerzan, S. (2007, junio). Large-scale named entity disambiguation based on Wikipedia data. En *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)* (pp. 708–716). Prague, Czech Republic: Association for Computational Linguistics. Descargado de <https://www.aclweb.org/anthology/D07-1074>
- Daft, R., y Lengel, R. (1986, 05). Organizational information requirements, media richness and structural design. *Management Science*, 32, 554-571. doi: 10.1287/mnsc.32.5.554
- Deutsch, J. (1966). Theories of choice and the stimulus-response framework. En E. Nagel, P. Suppes, y A. Tarski (Eds.), *Logic, methodology and philosophy of science* (Vol. 44, p. 413-423). Elsevier. Descargado de <https://www.sciencedirect.com/science/article/>

[pii/S0049237X09706081](https://doi.org/10.1016/S0049-237X(09)70608-1) doi: [https://doi.org/10.1016/S0049-237X\(09\)70608-1](https://doi.org/10.1016/S0049-237X(09)70608-1)

- Devlin, J., Chang, M.-W., Lee, K., y Toutanova, K. (2019). *Bert: Pre-training of deep bidirectional transformers for language understanding*.
- Di Pellegrino, G., Fadiga, L., Fogassi, L., Gallese, V., y Rizzolatti, G. (1992, 02). Understanding motor events: A neurophysiological study. *Experimental brain research. Experimentelle Hirnforschung. Expérimentation cérébrale*, 91, 176-80. doi: 10.1007/BF00230027
- Fogassi, L., y Ferrari, P. F. (2007). Mirror neurons and the evolution of embodied language. *Current Directions in Psychological Science*, 16(3), 136-141. Descargado de <https://doi.org/10.1111/j.1467-8721.2007.00491.x> doi: 10.1111/j.1467-8721.2007.00491.x
- Fremdling, R. (1996, 02). Industrial revolution and scientific and technological progress.
- Fromkin, V., Rodman, R., Hyams, N., Amberber, M., Cox, F., y Thornton, R. (2018). *An introduction to language* (9th ed.). Cengage Learning.
- Gatt, A., y Krahmer, E. (2017). Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *CoRR*, abs/1703.09902. Descargado de <http://arxiv.org/abs/1703.09902>
- Guyon, I., y Elisseeff, A. (2003, 01). An introduction of variable and feature selection. *J. Machine Learning Research Special Issue on Variable and Feature Selection*, 3, 1157 - 1182. doi: 10.1162/153244303322753616
- Güngör, T. (2010). Part-of-speech tagging. En *Handbook of natural language processing* (p. 205-235). Chapman Hall/CRC.
- Haight, F. A. (1981). *Applied probability*. Plenum Publishing Corporation.
- Harris, Z. S. (1954). Distributional structure. *WORD*, 10(2-3), 146-162. doi: 10.1080/00437956.1954.11659520
- Haspelmath, M. (2001, 12). Word classes and parts of speech. En (p. 16538-16545). doi: 10.1016/B0-08-043076-7/02959-4
- Hosmer, D. W., Lemeshow, S., y Sturdivant, R. X. (2013). *Applied logistic regression*. John Wiley Sons.
- Hoy, M. (2018, 01). Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly*, 37, 81-88. doi: 10.1080/02763869.2018.1404391
- Hsu, C.-W., y Lin, C.-J. (2002). A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2), 415-425. doi: 10.1109/72.991427

- Huang, M., Zhu, X., y Gao, J. (2019). Challenges in building intelligent open-domain dialog systems. *CoRR*, *abs/1905.05709*. Descargado de <http://arxiv.org/abs/1905.05709>
- Hull, D., y Grefenstette, G. (1996, 03). A detailed analysis of english stemming algorithms.
- Joachims, T. (1998, 01). Text categorization with support vector machines. *Proc. European Conf. Machine Learning (ECML'98)*. doi: 10.17877/DE290R-5097
- Jones, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, *28*, 11–21.
- Jurafsky, D., y Martin, J. H. (2020). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*.
- Khalid, M. A., Jijkoun, V., y de Rijke, M. (2008). The impact of named entity normalization on information retrieval for question answering. En C. Macdonald, I. Ounis, V. Plachouras, I. Ruthven, y R. W. White (Eds.), *Advances in information retrieval , 30th european conference on IR research, ECIR 2008, glasgow, uk, march 30-april 3, 2008. proceedings* (Vol. 4956, pp. 705–710). Springer. Descargado de https://doi.org/10.1007/978-3-540-78646-7_83 doi: 10.1007/978-3-540-78646-7_83
- Kripke, S. A. (2001). *Naming and necessity*. Harvard University Press.
- Krishnan, V., y Ganapathy, V. (2005). Named entity recognition..
- Kroeger, P. (2015). *Analyzing grammar: an introduction*. Cambridge University Press.
- Kuhn, T. (2014, marzo). A survey and classification of controlled natural languages. *Computational Linguistics*, *40*(1), 121–170. Descargado de <https://www.aclweb.org/anthology/J14-1005> doi: 10.1162/COLI_a_00168
- Kumari, A., y Shashi, M. (2019, 08). Vectorization of text documents for identifying unifiable news articles. *International Journal of Advanced Computer Science and Applications*, *10*, 305. doi: 10.14569/IJACSA.2019.0100742
- Ladani, D., y Desai, N. (2020, 03). Stopword identification and removal techniques on tc and ir applications: A survey. En (p. 466-472). doi: 10.1109/ICACCS48705.2020.9074166
- Lafferty, J. D., McCallum, A., y Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. En *Proceedings of the eighteenth international conference on machine learning* (p. 282–289). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- A language for stemming algorithms*. (s.f.). Descargado de <http://snowball.tartarus.org/>

[texts/introduction.html](#)

- LaPorte, J. (2016, Feb). *Rigid designators*. Stanford University. Descargado de <https://plato.stanford.edu/entries/rigid-designators/>
- Le-Hong, P., Phan, X.-H., y Tran, T.-T. (2013). On the effect of the label bias problem in part-of-speech tagging. En *The 2013 rivf international conference on computing communication technologies - research, innovation, and vision for future (rivf)* (p. 103-108). doi: 10.1109/RIVF.2013.6719875
- Linguistic features · spacy usage documentation*. (s.f.). Descargado de <https://spacy.io/usage/linguistic-features#how-tokenizer-works>
- Lita, L., Ittycheriah, A., Roukos, S., y Kambhatla, N. (2003, 06). truecasing. doi: 10.3115/1075096.1075116
- Liu, Q., Kusner, M. J., y Blunsom, P. (2020). A survey on contextual embeddings. *CoRR, abs/2003.07278*. Descargado de <https://arxiv.org/abs/2003.07278>
- Lo, R. T., He, B., y Ounis, I. (2005). Automatically building a stopword list for an information retrieval system. *J. Digit. Inf. Manag.*, 3, 3-8.
- Long, N., Kuhl, B., y Chun, M. (2018, 01). Memory and attention. En (p. 1-37). doi: 10.1002/9781119170174.epcn109
- Lu, H. (2020). Quasi-orthonormal encoding for machine learning applications. *CoRR, abs/2006.00038*. Descargado de <https://arxiv.org/abs/2006.00038>
- López, Diaz, Quesada, y Guerrero. (2019). User experience comparison of intelligent personal assistants: Alexa, google assistant, siri and cortana. *Proceedings*, 31(1), 51. doi: 10.3390/proceedings2019031051
- Malle, B., y Knobe, J. (1997, 03). The folk concept of intentionality. *Journal of Experimental Social Psychology*, 33, 101–121. doi: 10.1006/jesp.1996.1314
- Manning, C. D., Raghavan, P., y Schütze, H. (2008). *Introduction to information retrieval*. USA: Cambridge University Press.
- McCallum, A., Freitag, D., y Pereira, F. C. N. (2000). Maximum entropy markov models for information extraction and segmentation. En *Proceedings of the seventeenth international conference on machine learning* (p. 591–598). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- McTear, M. F. (2004). *Spoken dialogue technology: Toward the conversational user interface*.

- Springer London.
- Merriam-Webster. (s.f.). *Contraction*. Descargado 2021-05-17, de <https://www.merriam-webster.com/dictionary/contraction>
- Microsoft docs. (s.f.). <https://docs.microsoft.com/es-es/azure/bot-service/bot-service-overview-introduction?view=azure-bot-service-4.0>, note = Accessed: 2021-08-21.
- Microsoft luis docs. (s.f.). <https://docs.microsoft.com/es-es/azure/cognitive-services/luis/>, note = Accessed: 2021-08-21.
- Mikolov, T., Chen, K., Corrado, G. S., y Dean, J. (2013). *Efficient estimation of word representations in vector space*. Descargado de <http://arxiv.org/abs/1301.3781>
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., y Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *CoRR*, *abs/1310.4546*. Descargado de <http://arxiv.org/abs/1310.4546>
- Moens, M.-F. (2006). *Information extraction: algorithms and prospects in a retrieval context*. Springer.
- Mushtaq, S. (2019, Jun). *Data preprocessing in detail*. Descargado de <https://developer.ibm.com/technologies/data-science/articles/data-preprocessing-in-detail/>
- Nadeau, D., y Sekine, S. (2007, 08). A survey of named entity recognition and classification. *Linguisticae Investigationes*, 30. doi: 10.1075/li.30.1.03nad
- Nair, S., Srinivasan, M., y Meylan, S. C. (2020). Contextualized word embeddings encode aspects of human-like word sense knowledge. *CoRR*, *abs/2010.13057*. Descargado de <https://arxiv.org/abs/2010.13057>
- Naseem, U., Razzak, I., Khan, S. K., y Prasad, M. (2020). *A comprehensive survey on word representation models: From classical to state-of-the-art word representation language models*.
- Nguyen, N., y Guo, Y. (2007). Comparisons of sequence labeling algorithms and extensions. *Proceedings of the 24th international conference on Machine learning - ICML 07*. doi: 10.1145/1273496.1273582
- Nlu training data*. (2021, May). Descargado de <https://rasa.com/docs/rasa/nlu-training-data/#entities-roles-and-groups>
- Palmer, D. (2010). Text preprocessing. En *Handbook of natural language processing* (p. 9-30). Chapman Hall/CRC.
- Palmer, D. D., y Hearst, M. A. (1997). Adaptive multilingual sentence boundary disambigua-

- tion. *Computational Linguistics*, 23(2), 241–267. Descargado de <https://www.aclweb.org/anthology/J97-2002>
- Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., y Zettlemoyer, L. (2018). Deep contextualized word representations. *CoRR*, *abs/1802.05365*. Descargado de <http://arxiv.org/abs/1802.05365>
- Picard, R. W. (1995). *Affective computing* (Inf. Téc. n.º M.I.T Media Laboratory Perceptual Computing Section Technical Report No. 321).
- A practical guide to sentiment analysis. (2017). *Socio-Affective Computing*. doi: 10.1007/978-3-319-55394-8
- RAE. (s.f.). *Locución*. Descargado 2021-05-17, de <https://dle.rae.es/locuci%C3%B3n>
- Rasa actions*. (s.f.). <https://rasa.com/docs/rasa/actions>. (Accessed: 2021-08-22)
- Rasa architecture*. (s.f.). <https://rasa.com/docs/rasa/arch-overview>. (Accessed: 2021-08-22)
- Rasa componentes*. (s.f.). <https://rasa.com/docs/rasa/components>. (Accessed: 2021-08-22)
- Rasa docs*. (s.f.-a). <https://rasa.com/docs/rasa/>. (Accessed: 2021-08-22)
- Rasa docs*. (s.f.-b). <https://rasa.com/docs/rasa/glossary>. (Accessed: 2021-08-22)
- Rasa domain*. (s.f.). <https://rasa.com/docs/rasa/domain>. (Accessed: 2021-08-22)
- Rasa forms*. (s.f.). <https://rasa.com/docs/rasa/forms>. (Accessed: 2021-08-22)
- Rasa policies*. (s.f.). <https://rasa.com/docs/rasa/policies>. (Accessed: 2021-08-22)
- Rasa training data format*. (s.f.). <https://rasa.com/docs/rasa/training-data-format>. (Accessed: 2021-08-22)
- Reiter, E., y Dale, R. (2000). Building natural language generation systems. doi: 10.1017/cbo9780511519857
- Reiter, E., y Dale, R. (2002, 03). Building applied natural language generation systems. *Natural Language Engineering*, 3. doi: 10.1017/S1351324997001502
- Rescorla, M. (2019, May). *The language of thought hypothesis*. Stanford University. Descargado de <https://plato.stanford.edu/entries/language-thought/>
- Rish, I. (2001). *An empirical study of the naive bayes classifier* (Inf. Téc.).
- Ross, P., y Maynard, K. (2021). Towards a 4th industrial revolution. *Intelligent Buildings International*, 13(3), 159-161. Descargado de <https://doi.org/10.1080/17508975.2021.1873625> doi: 10.1080/17508975.2021.1873625
- Russell, J. A. (1980). A circumplex model of affect. *Journal of Personality and Social Psychology*,

- 39(6), 1161–1178. doi: 10.1037/h0077714
- Sahlgren, M. (2008). The distributional hypothesis. *Italian Journal of Linguistics*, 20(1), 33–54.
- Sarawagi, S., y Cohen, W. W. (2004). Semi-markov conditional random fields for information extraction. En *Nips*.
- Savoy, J., y Gaussier, E. (2010). Information retrieval. En *Handbook of natural language processing* (p. 455-484). Chapman Hall/CRC.
- Schwab, S. (2015, 11). Accent mark and visual word recognition in spanish. *Loquens*, 2, e018. doi: 10.3989/loquens.2015.018
- Shannon, C. E., y Weaver, W. (1964). *The mathematical theory of communication*. Univ. of Illinois Pr.
- Stern, R., Sagot, B., y Béchet, F. (2012, abril). A joint named entity recognition and entity linking system. En *Proceedings of the workshop on innovative hybrid approaches to the processing of textual data* (pp. 52–60). Avignon, France: Association for Computational Linguistics. Descargado de <https://www.aclweb.org/anthology/W12-0508>
- Turing, A. M. (1950, 10). I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236), 433-460. Descargado de <https://doi.org/10.1093/mind/LIX.236.433> doi: 10.1093/mind/LIX.236.433
- van Deemter, K., Theune, M., y Krahmer, E. (2005, 03). Real versus template-based natural language generation: A false opposition? *Computational Linguistics*, 31, 15-24. doi: 10.1162/0891201053630291
- Vlasov, V., Mosig, J. E. M., y Nichol, A. (2019). *Dialogue transformers*.
- Weizenbaum, J. (1966, enero). Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1), 36–45. Descargado de <https://doi.org/10.1145/365153.365168> doi: 10.1145/365153.365168
- Willett, P. (2006, 07). The porter stemming algorithm: Then and now. *Program electronic library and information systems*, 40. doi: 10.1108/00330330610681295
- Wit.ai*. (s.f.). <https://wit.ai/>, note = Accessed: 2021-08-23.
- Wit.ai recipes*. (s.f.). <https://wit.ai/docs/recipes>, note = Accessed: 2021-08-23.
- Xu, H., Caramanis, C., y Mannor, S. (2008, 03). Robustness and regularization of support vector machines. *Journal of Machine Learning Research*, 10.
- Zeng, Y. (2019). *Context aware machine learning* (Inf. Téc.). Google Inc.

Zhang, Z., Takanobu, R., Huang, M., y Zhu, X. (2020). Recent advances and challenges in task-oriented dialog system. *CoRR*, *abs/2003.07490*. Descargado de <https://arxiv.org/abs/2003.07490>



UNIVERSIDAD
DE MÁLAGA

| uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA