Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Desarrollo de un asistente virtual empleando el framework Rasa

Autor:
Alberto Castañeiras Folgueral

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Desarrollo de un asistente virtual empleando el framework Rasa

Autor:
Alberto Castañeiras Folgueral

Tutor:
César González Ferreras

Resumen

El desarrollo de asistentes virtuales y chatbots ha experimentado un crecimiento significativo en la última década, impulsado por avances en inteligencia artificial y comprensión del lenguaje natural (NLU). Este Trabajo de Fin de Grado explora esta evolución mediante el desarrollo de un asistente utilizando el framework Rasa, reconocido por su flexibilidad y robustez en la creación de soluciones de chatbot adaptadas a necesidades específicas. Rasa se destaca en el ámbito de los asistentes virtuales por su capacidad para entrenar modelos de diálogo contextual que pueden gestionar y responder de manera efectiva a interacciones complejas basadas en intenciones y entidades reconocidas.

El enfoque del proyecto ha sido aprovechar las capacidades avanzadas de Rasa para desarrollar un asistente virtual cinéfilo que no solo ofrece recomendaciones personalizadas sobre películas, sino que también facilita conversaciones fluidas y naturales con los usuarios. Integrado mediante la aplicación de mensajería Telegram, el chatbot proporciona a los usuarios acceso a información detallada sobre películas, que incluye tramas, elenco y equipo directivo, gracias a la API de la web TMDB, que es la fuente principal de datos del sistema. Esta interacción, potenciada por la comprensión del lenguaje natural, permite al asistente comprender y responder a consultas complejas, mejorando la accesibilidad y la experiencia del usuario.

Los resultados del proyecto subrayan el impacto transformador de los chatbots, demostrando cómo plataformas como Rasa pueden ser empleadas para superar limitaciones previas en la interacción hombre-máquina, y sugieren un futuro prometedor donde los asistentes virtuales juegan un papel clave en la mejora de la experiencia del cliente en diversos sectores.

Abstract

The development of virtual assistants and chatbots has experienced significant growth in the last decade, driven by advances in artificial intelligence and natural language understanding (NLU). This thesis explores this evolution through the development of an assistant using the Rasa framework, recognized for its flexibility and robustness in the creation of chatbot solutions tailored to specific needs. Rasa stands out in the field of virtual assistants for its ability to train contextual dialogue models that can effectively manage and respond to complex interactions based on intentions and recognized entities.

The focus of the project has been to leverage Rasa's advanced capabilities to develop a cinephile virtual assistant that not only offers personalized movie recommendations, but also facilitates fluid and natural conversations with users. Integrated via the Telegram messaging app, the chatbot provides users with access to detailed movie information, including plots, cast and crew, thanks to the TMDB web API, which is the system's primary source of data. This interaction, powered by natural language understanding, enables the assistant to understand and respond to complex queries, improving accessibility and user experience.

The project results underscore the transformative impact of chatbots, demonstrating how platforms such as Rasa can be employed to overcome previous limitations in human-machine interaction, and suggest a promising future where virtual assistants play a key role in improving the customer experience in various sectors.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Capítulo 1

Introducción

1.1. Contexto

El auge de los chatbots ha venido de la necesidad creciente de eficiencia operativa y mejora de la experiencia del cliente en numerosos sectores. Las empresas buscan constantemente maneras de optimizar la atención al cliente, reducir tiempos de espera y costes, y ofrecer servicios accesibles las 24 horas del día. Los chatbots permiten automatizar respuestas a consultas comunes, manejar transacciones simples y proporcionar asistencia inmediata, lo que puede ser particularmente útil fuera del horario laboral o durante picos de demanda.

No todos los chatbots están equipados con Inteligencia Artificial (IA), pero los chatbots modernos utilizan cada vez más técnicas de IA conversacional como el NLP para comprender las preguntas de los usuarios y automatizar las respuestas.[3]

El Natural Language Processing (Procesamiento del Lenguaje Natural) (NLP) describe la capacidad de una máquina para ingerir lo que se le dice, descomponerlo y comprender cuál es su significado. El objetivo es determinar una acción adecuada. El Natural Language Understanding (Comprensión del Lenguaje Natural) (NLU) es un subconjunto de NLP que trata con un área mucho más específica. Se enfoca en cómo manejar mejor las entradas no estructuradas para convertir estas en un formato estructurado, con técnicas de NLU ayudamos a la máquina a comprender las conversaciones.[1]

La habilidad de los chatbots basados en IA para procesar el lenguaje natural, así como para ofrecer servicios personalizados de manera automatizada, aporta ventajas significativas tanto para las organizaciones como para sus clientes.[3]

- **Disponibilidad:** Operan ininterrumpidamente, 24 horas del día, los 7 días de la semana, garantizando una línea de asistencia constante y sin interrupciones.
- **Eficacia operativa:** Alivian la carga de los agentes humanos gestionando consultas repetitivas, lo que les permite concentrarse en cuestiones más complejas y mejorar la calidad del servicio.

- **Reducción de costes:** Reducen significativamente los gastos asociados con la operación de centros de asistencia y la formación de personal, al minimizar la dependencia de asistencia humana.

- **Integración:** Se integran fácilmente en una amplia variedad de plataformas, incluidas páginas web, y aplicaciones de mensajería como WhatsApp, Telegram, y Facebook Messenger. Esta flexibilidad permite que los chatbots estén accesibles en el entorno habitual de los usuarios.

- **Mejora de la experiencia del cliente:** Ofrecen respuestas rápidas y precisas, mejorando la satisfacción del cliente y reduciendo los tiempos de espera.

- **Escalabilidad:** Facilitan la gestión de grandes volúmenes de interacciones simultáneamente sin comprometer la calidad del servicio.

- **Recolección de datos:** Pueden recoger datos de las interacciones con los usuarios, obteniendo valiosa información sobre productos o servicios, lo que facilita mejoras continuas.

- **Soporte multilingüe:** Ofrecen soporte en múltiples idiomas, eliminando barreras de comunicación en un mercado global.[6]

1.2. Motivación

Los primeros chatbots eran esencialmente programas interactivos de preguntas frecuentes basados en un conjunto limitado de preguntas comunes con respuestas preescritas. Con el tiempo, han evolucionado hacia sistemas complejos capaces de gestionar interacciones detalladas, gracias a los avances en IA y NLP. En este contexto, el framework Rasa se destaca como una solución potente por su flexibilidad y capacidad para crear chatbots que no solo responden preguntas, sino que también comprenden el contexto y las intenciones detrás de las interacciones.

Este Trabajo de Fin de Grado explora las capacidades avanzadas de Rasa desarrollando un asistente virtual especializado en cine, integrado a través de Telegram y alimentado por datos de la API de la web TheMovieDatabase (https://www.themoviedb.org/) (TMDB).

Además, este proyecto ha marcado mi primer paso significativo en el mundo de la inteligencia artificial. Este enfoque práctico no solo ha enriquecido mi formación académica, sino que también ha encendido una chispa de curiosidad y posibilidad, abriéndome los ojos a nuevas formas de aplicar las tecnologías de IA y NLP en situaciones reales.

1.3. Objetivo

A continuación se exponen los principales objetivos que abarca este TFG:

1. **Estudiar los conceptos básicos de los chatbots:** Explorar la evolución de los chatbots desde simples programas de preguntas frecuentes hasta sistemas avanzados que utilizan inteligencia artificial para manejar conversaciones complejas.

2. **Analizar el framework Rasa:** Estudiar las capacidades de Rasa como herramienta para el desarrollo de chatbots. Investigar cómo se integran conceptos de inteligencia artificial como el procesamiento de lenguaje natural dentro del framework.

3. **Desarrollar un asistente virtual cinéfilo utilizando Rasa:**

   - **Diseño de conversaciones:** Crear y optimizar conversaciones que el chatbot será capaz de mantener con los usuarios. Esto incluye definir intenciones, acciones necesarias e historias para asegurar un flujo coherente en la interacción.
   - **Integración con Telegram:** Configurar Rasa para comunicarse con los usuarios a través de la plataforma de mensajería Telegram.
   - **Extracción de información mediante la API de TMDB:** Implementar la conexión y extracción de datos de películas, incluyendo tramas, elenco y dirección, desde la API de TMDB para alimentar las respuestas del chatbot.
   - **Evaluación del chatbot:** Realizar pruebas con usuarios finales para evaluar la efectividad del chatbot. Analizar las conversaciones para identificar mejoras.

1.4. Estructura

La memoria sigue la siguiente estructura:

- **Capítulo 1. Introducción:** Este capítulo expone el contexto, la motivación y los objetivos del trabajo de fin de grado. Se explica la relevancia de los chatbots y la inteligencia artificial en la interacción actual entre humanos y máquinas.
- **Capítulo 2. Rasa Framework:** Aquí se profundiza en el framework Rasa, explicando los conceptos clave, sus productos y la metodología (Conversation-Driven Development (Desarrollo guiado por la conversación) (CDD)). Además, se detallan las herramientas que proporciona Rasa para el desarrollo de chatbots.
- **Capítulo 3. Planificación:** Se describe el análisis, la planificación y la estimación de iteraciones del proyecto, junto con las fechas de entrega previstas. Este capítulo también aborda la metodología de gestión de proyectos utilizada.
- **Capítulo 4. Desarrollo:** Este capítulo detalla el proceso de desarrollo del asistente virtual, incluyendo el diseño de conversaciones, la integración con Telegram y la implementación de la extracción de datos mediante la API de TMDB.
- **Capítulo 5. Conclusiones y trabajo futuro:** Se presentan las conclusiones del proyecto, los resultados obtenidos y las recomendaciones para trabajos futuros. Se discuten las posibilidades de expansión del proyecto y las mejoras tecnológicas propuestas.

Finalmente, se incluyen la bibliografía y los anexos del proyecto que complementan la investigación realizada.

Capítulo 2

Rasa Framework

Rasa Open Source es un framework diseñado para construir asistentes virtuales de alto rendimiento.

La versión de Rasa utilizada para el desarrollo del TFG es Rasa 3.x

2.1. Conceptos

Rasa utiliza ficheros YAML para almacenar sus datos de entrenamiento, incluyendo datos de NLU, historias y reglas. Los datos de entramiento en Rasa están organizados bajo *claves*. Las principales *claves* son: **nlu**, **stories** y **rules** 2.1. Cada archivo YAML puede contener múltiples claves, pero cada clave solo puede aparecer una vez por archivo. Rasa lee todos los ficheros YAML a su alcance y organiza el contenido por *claves*.

2.1.1. NLU

Los datos de entrenamiento NLU en Rasa consisten en expresiones de los usuarios que se categorizan por intents (intenciones), que representan lo que un usuario intenta transmitir o lograr con su mensaje. Por ejemplo, una intención podría ser request_movie"donde el usuario busca que se le recomiende una película. Cada intención se asocia con múltiples ejemplos de posibles expresiones que los usuarios podrían usar para expresar esa intención específica. Esto permite al modelo de Rasa aprender y generalizar a partir de las variaciones en la forma de expresar una misma intención.

Además, Rasa permite la anotación de entities (entidades) dentro de los datos de entrenamiento. Las entidades son información específica que se extrae de los mensajes (intents), como nombres, teléfonos o fechas. Por ejemplo, en la intención request_movie", una entidad podría incluir el género 2.2. La identificación correcta de estas entidades permite al chatbot elaborar respuestas más precisas al usuario.

```yaml
version: "3.1"

nlu:
- intent: start
  examples: |
    - Hola
    - Saludos

- intent: request_movie
  examples: |
    - ¿Qué película me recomiendas?
    - Quiero ver una película

stories:
- story: start y user request movie
  steps:
  - intent: start
  - action: utter_start
  - intent: request_movie
  - action: utter_find_movie

rules:
- rule: start
  steps:
  - intent: start
  - action: utter_start
```

Figura 2.1: Ejemplo de archivo YAML con claves Rasa: nlu, stories y rules

```yaml
- intent: request_movie
  examples: |
    - ¿Qué película me recomiendas?
    - Quiero ver una película
    - Me apetece ver una película de [terror](movie_genre)
    - Me gustan las películas de [indios y vaqueros]{"entity": "movie_genre", "value": "Western"}
```

Figura 2.2: Intent con entity movie_genre

Para facilitar la extración de las entidades se pueden usar:

- **Expresiones regulares (regex):** si tu entidad tiene un estructura que puede ser descrita por una expresión regular, como números de teléfonos, números de tarjetas de crédito, etc.
- **Tablas de búsqueda (lookup table):** Las tablas de búsqueda son listas de palabras que se utilizan para generar patrones de expresiones regulares insensibles a mayúsculas y minúscula. Todos los ejemplo en la tabla de búsqueda se combinan en una gran expresión regular. Esta expresión regular se utiliza para comprobar si cada ejemplo de entrenamiento contiene coincidencias con las entradas de la tabla de búsqueda. 2.3
- **Sinónimos (synonym):** Utilizados para mapear entidades extraídas a un valor que podría ser diferente del texto literal extraído. Esto es útil cuando diferentes entradas del usuario se refieren a la misma cosa bajo diferentes nombres. 2.3

La categorización efectiva de las intenciones y entidades en los datos de entrenamiento son fundamentales para el desempeño del sistema NLU en Rasa, permitiéndole comprender y responder de manera más efectiva a las interacciones del usuario.

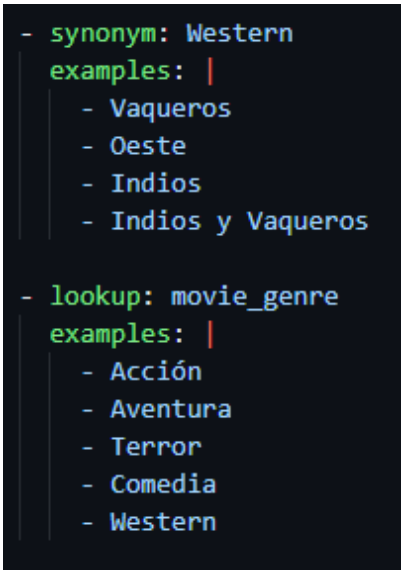

Figura 2.3: Ejemplo de lookup table con synonym

2.1.2. Stories

Las historias (stories) son un tipo de datos de entrenamiento utilizados para entrenar el modelo de gestión del diálogo del asistente. Las stories pueden utilizarse para entrenar modelos capaces de generalizar rutas de conversación desconocidas.[13]

Cada story en Rasa se identifica por un nombre y se compone de una serie de pasos que simulan una interacción entre el usuario y el chatbot. Los pasos incluyen las intenciones del usuario, resultado del sistema NLU integrado en Rasa por lo que no hace falta lidiar con el contenido de los mensajes para construir una story, y acciones del chatbot.

Las acciones a ejecutar por el chatbot se definen en el fichero domain. Existen dos tipos de acciones que el chatbot puede llevar a cabo las responses y las acciones personalizadas:

- **Responses:** Respuestas predefinidas que el bot puede enviar al usuario, como un mensaje de texto. Se suele proporcionar una batería de frases a enviar y el chatbot escoge una aleatoria.
- **Acciones Personalizadas:** Funciones escritas en Python, usando las librerías que ofrece Rasa, que el chatbot puede ejecutar para realizar tareas más complejas, como consultar una base de datos o llamar a una API externa.

Las stories son vitales para construir asistentes virtuales que pueden manejar diálogos complejos y dinámicos.

2.1.3. Rules

Las reglas son un tipo de datos de entrenamiento que se utilizan para entrenar el modelo de gestión de diálogos de su asistente. Las reglas describen fragmentos breves de conversaciones

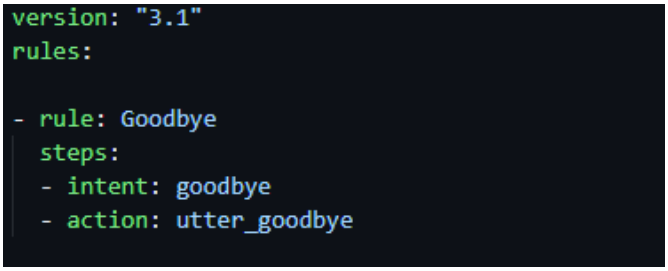

Figura 2.4: Ejemplo de regla: despedirse siempre que el usuario tenga intención de despedirse

que deben seguir siempre el mismo camino [12]. A diferencia de las historias que permiten cierta generalización y aprendizaje de patrones conversacionales más amplios, las reglas son ideales para gestionar patrones de conversación pequeños y específicos.

Para implementar una regla en Rasa, primero debes asegurarte de que la RulePolicy está incluida en la configuración de tu modelo, hablaremos del fichero config más adelante. Esto permite que las reglas sean reconocidas y utilizadas por el asistente durante las conversaciones.

A las reglas se le pueden aplicar condiciones, que son requisitos que deben cumplirse para que la regla sea aplicable.

Es importante no sobrecargar tu asistente con demasiadas reglas, ya que esto puede limitar su capacidad para manejar entradas inesperadas de manera flexible. Las reglas deben ser combinadas con historias para crear un asistente robusto.

Rules vs Stories

Las reglas un tipo de datos de entrenamiento utilizados para manejar trozos de conversaciones que deben seguir siempre el mismo camino [10]. Las reglas pueden ser útiles a la hora de implementar:

- Interacciones de una sola vez: existen mensajes que no necesitan contexto para responderlos. Las reglas son una forma sencilla de asignar intenciones a respuestas, siendo respuestas fijas para estos mensajes. Por ejemplo, Despedirse.
- Fallback: las reglas son útiles para responder a mensajes de usuario de baja confianza con un determinado fallback.
- Formularios: para la activación de un formulario suele seguir una ruta fija.

2.1.4. Domain

El fichero domain (dominio) define el universo en el que opera el asistente. Especifíca los intents, entities, slots, responses, forms y actions que el bot debe conocer. En él también se define una configuración para las sesiones de conversación.

```yaml
slots:
  movie_genre:
    type: text
    mappings:
    - type: from_entity
      entity: movie_genre
```

Figura 2.5: Slot que emplea from_entity para almacenar valores de la entidad movie_genre

2.1.5. Slots

Los slots en Rasa son las variables del chatbot que almacenan información que el usuario ha proporcionado en el transcurso de la conversación, así como información recopilada sobre el mundo exterior.

En Rasa, los slots pueden afectar el flujo de la conversación, dependiendo si están configurados para ello en el dominio del chatbot (se debe incluir la opción influence_conversation).

Existen distintos tipos de slots según la información que almacenan:

- **Texto:** Almacena valores de texto
- **Boolean:** Almacena valores lógicos, verdadero o falso.
- **Categórico:** Almacena un valor de un conjunto limitado de valores.
- **Float:** Almacena números reales
- **List:** Almacena una lista de valores.
- **Any:** Los slots any se utilizan para almacenar diccionarios o pares clave-valor.

Si ninguno de estos tipos predefinidos de slots se ajusta a como debe estar almacenada la información que proporciona el usuario, se pueden crear slots personalizados usando Python y las librerías que proporciona Rasa.

Los "slot mappings"son una característica importante en Rasa que permite especificar cómo se deben llenar los slots con información extraída de los mensajes del usuario. Se pueden configurar los slots para que se llenen automáticamente basándose en entidades extraídas de los mensajes 2.5, valores predeterminados, o incluso a través de acciones personalizadas que ejecuten alguna lógica específica.

2.1.6. Responses

Las responses (respuestas) son mensajes que su asistente envía al usuario. Una respuesta suele ser sólo texto, pero también puede incluir contenido como imágenes y botones. [11]

Las respuestas pueden personalizarse con los slots, encerrados entre llaves {}. Por ejemplo, {movie_genre} en una respuesta será reemplazado por el valor del slot `movie_genre`.

Rasa permite definir múltiples variaciones para una misma respuestas, y se selecciona una aleatoriamente entre esas variaciones para hacer la conversión menos predecible. Además, se pueden crear mensajes específicos para canales de comunicación usando `channel`.

Las respuestas también pueden ser condicionales, basadas en los valores de los slots. Esto permite que el asistente proporcione respuestas que dependen del estado actual de la conversación.

2.1.7. Forms

Los forms (formularios) en Rasa son herramientas esenciales para la recopilación estructurada de información a través de conversaciones, también se conoce como `relleno de slots`. Permiten al asistente solicitar y recopilar varios datos del usuario de manera organizada, necesarios para realizar una tarea específica que solicita el usuario. Por ejemplo, los formularios serían útiles a la hora de reservar una mesa en un restaurante, unos vuelos, etc.

Para implementar un formulario, debes describirlo en el archivo domain, especificando los slots que necesita llenar. Los forms se activan mediante reglas o historias, indicando cuando activarse el form en respuesta a ciertas intenciones del usuario. Un form se desactivará automáticamente una vez que todos los slots estén llenos.

Los forms son personalizables usando código Python y la librería de Rasa. Se puede validar la entrada para cada slot utilizando acciones personalizadas que verifiquen si los datos recogidos cumplen los criterios, consultando APIs o bases de datos externas. Si una entrada no es válida, se puede configurar el form para solicitar nuevamente la información.

2.1.8. Actions

En Rasa, las actions (acciones) permiten al chatbot realizar tareas como responder a preguntas del usuario, interactuar con bases de datos, llamar a APIs externas, o simplemente ejecutar cualquier lógica de programación que sea necesaria durante la conversación.

Tipos de acciones

- **Acciones Predeterminadas:** Rasa incluye algunas acciones predefinidas como `action_listen` (esperar a escuchar que dice el usuario), `action_restart` (reinicia todo el historial de la conversación), etc.
- **Acciones Personalizadas:** Son acciones definidas por el desarrollador para realizar tareas específicas. Estas acciones se programan en Python usando la librería que proporciona Rasa.

```python
class ActionMovie(Action):
    def name(self) -> Text:
        return "action_inform_movie"

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[EventType]:

        movie_id: int = tracker.get_slot("movie_id")

        api = tmdb.TMDB()
        try:
            movie = api.read_movie(movie_id)

            dispatcher.utter_message(image=f"{movie.movie_poster_url()}")
            dispatcher.utter_message(text=f"{movie.name} ({movie.release_date[:4]})")
            dispatcher.utter_message(text=f"Puntuación: {movie.vote_average} ⭐")

        except tmdb.TMDBException as e:
            dispatcher.utter_message(text=f"API: {e}")

        return []
```

Figura 2.6: Ejemplo de acción personalizada

2.2. Configuración

El archivo de configuración (config.yml) define los componentes y las políticas el modelo Rasa utilizará para realizar predicciones basadas en las entradas del usuario 2.7.

Las claves language y pipeline especifican los componentes utilizados por el modelo para realizar predicciones NLU. La clave policies define las políticas utilizadas por el modelo para predecir la siguiente acción.

Lo primero que podemos especificar en el archivo de configuración es que lenguaje en el que el usuarios se va a comunicar con el chatbot.

2.2.1. Pipeline NLU

El pipeline define cómo Rasa procesa el texto de entrada. Incluye componentes para tokenización, extracción de entidades, y clasificación de intenciones. Puedes personalizar el pipeline para adaptarlo a necesidades específicas, agregando o modificando componentes según el lenguaje y las particularidades del dominio de aplicación.

Existen distintos tipos de componentes:

- **Tokenizer:** El primer paso en el pipeline de Rasa es el proceso de tokenización implica dividir el texto en unidades más pequeñas, conocidas como tokens. Que pueden ser palabras, caracteres o subpalabras. Este paso es crucial en el procesamiento del lenguaje natural (NLP).

```yaml
language: en

pipeline:
- name: WhitespaceTokenizer
- name: RegexFeaturizer
- name: LexicalSyntacticFeaturizer
- name: CountVectorsFeaturizer
- name: CountVectorsFeaturizer
  analyzer: char_wb
  min_ngram: 1
  max_ngram: 4
- name: DIETClassifier
  epochs: 100

policies:
- name: MemoizationPolicy
- name: TEDPolicy
  max_history: 5
  epochs: 10
- name: RulePolicy
```

Figura 2.7: Ejemplo de archivo config.yml

Figura 2.8: Ejemplo tokenizer

Existen diferentes maneras de tokenizar. Rasa ofrece por defecto el `WhitespaceTokenizer` que consiste es usar espacios para separar el texto en palabras, aunque también se suelen eliminar signos de puntuación. Otro tokenizador popular es el de la biblioteca spaCy, que puede manejar mejor las peculiaridades de diversos idiomas y que Rasa ofrece opcionalmente.

Esto debe ocurrir antes de que el texto se *featurice* para el aprendizaje automático, por lo que normalmente se incluye un tokenizador al principio de una canalización.2.8 [9]

- **Featurizer:** Una vez segmentado el texto en tokens, estos deben vectorizarse para aplicar técnicas de aprendizaje automático, conocido también como `word embedding`. Eso es lo que hacen los featurizers 2.9. En Rasa, existen dos tipos de featurizer utilizadas para el procesamiento del lenguaje:

  - **Sparse Features (Featurizers Dispersos):** Como por ejemplo CountVectorizer y LexicalSyntacticFeaturizer que son esenciales para procesar el texto en la detección de entidades.
  - **Dense Features (Featurizers Densos):** Modelos de lenguaje pre-entrenados como Spacy mediante SpaCyFeaturizers o Hugging Face mediante LanguageModelFeaturizers.

Rasa aplica estos featurizadores a todos los tokens, pero también genera características para toda la frase. [9]

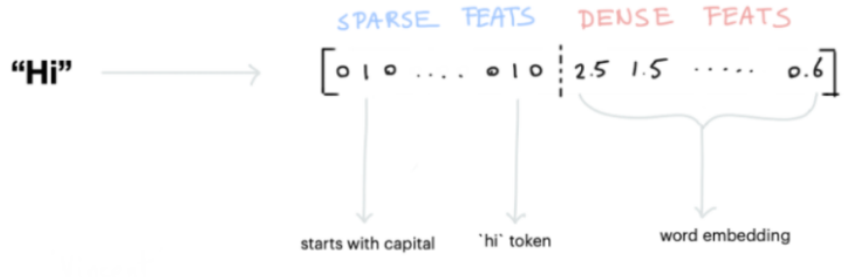

Figura 2.9: Codificación de la palabra Hola

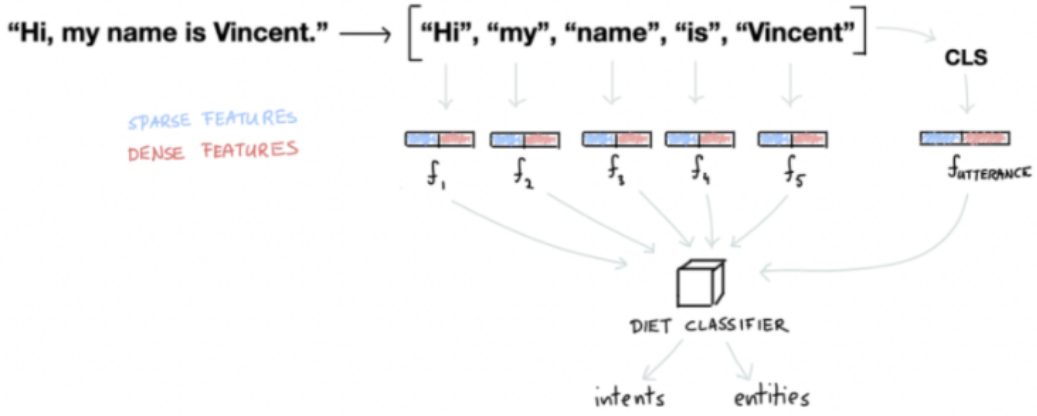

Figura 2.10: Ejemplo intent classifier

- **Intent Classifiers:** Una vez que se han vectorizado los tokens y las frases completas, se pasa al modelo de clasificación de intenciones, el `DIET` de Rasa. Este modelo es maneja tanto la clasificación de intenciones como la extracción de entidades.
- **Entity Extractors:** Aunque el modelo `DIET` de Rasa es capaz de detectar entidades, no siempre es la mejor opción para todas las clases de entidades, especialmente aquellas con patrones estructurados como números de teléfono. Para estas, es más eficiente usar el `RegexEntityExtractor` que utiliza expresiones regulares para identificar entidades de forma precisa y sencilla. Adicionalmente según la necesidad se podría integrar un modelo de spaCy [4] o Duckling entrenado independiente de Rasa en el pipeline los extractores `SpacyEntityExtractor` o `DucklingEntityExtractor`, respectivamente. Para detectar entidades genéricas como nombres propios, ubicaciones o tiempo.

2.2.2. Policies

Cuando Rasa predice una acción, no solo tiene en cuenta las intenciones y entidades actuales, sino que también mira la conversación hasta el momento de predecir. Utiliza políticas para decidir qué acción tomar en cada paso de la conversación:

- **RulePolicy:** Gestiona conversaciones que coinciden con patrones de reglas predefinidos, haciendo predicciones basadas en las reglas especificadas en la clave rules de los datos de entrenamiento NLU.

- **MemoizationPolicy:** Verifica si la conversación actual coincide con alguna de las historias en los datos de entrenamiento y predice la acción siguiente basándose en esas coincidencias.
- **TedPolicy:** Emplea aprendizaje automático para predecir la próxima mejor acción basándose en el contexto de la conversación.

2.3. Metodología CDD

Conversation-Driven Development (CDD) es la metodología recomendada por Rasa para el desarrollo de chatbots que se centra en escuchar a los usuarios y utilizar esa información para mejorar el asistente. El desarrollo de asistentes virtuales con IA es un reto porque los usuarios siempre dirán algo que no habías previsto [8]. Al practicar CDD en cada etapa del desarrollo del chatbot, se orienta al asistente hacia el lenguaje y el comportamiento de usuarios reales.

Las acciones que componen CDD son [7]:

- **Compartir:** Despliega prototipos del asistente a usuarios para obtener retroalimentación lo más pronto posible.
- **Revisar:** Analizar y revisar las conversaciones entre los usuarios y el asistente para identificar mejor las interacciones.
- **Annotar:** Utiliza los datos recogidos para entrenar y mejorar el modelo NLU del asistente.
- **Probar:** Asegúrate de que el asistente se comporta como se espera usando test.
- **Rastrear:** Identificar y medir las conversaciones exitosas para entender qué funciona.
- **Corregir:** Aprender de los fallos y ajusta cómo el asistente maneja situaciones inesperadas o respuestas inadecuadas.

El Desarrollo guiado por la conversación no es un proceso lineal, implica volver repetidamente sobre las mismas acciones a medida que desarrollas y mejorar tu asistente virtual. Este enfoque cíclico es fundamental para adaptar y refinar continuamente el asistente basado en la interacción real y el feedback de los usuarios, garantizando así una mejora constante en su rendimiento.

Capítulo 3

Planificación

3.1. Metodología

En la realización del Trabajo de Fin de Grado opté por adoptar la metodología scrum y su enfoque ágil, debido a la propia naturaleza dinámica del desarrollo del chatbots. Esta metodología me permite realizar ajustes continuos basados en el feedback obtenido de la interacción con usuarios reales.

Además se alinea con la metodología propuesta por Rasa CDD, que como describí anteriormente, que recomienda sacar versiones cuanto antes del chatbot, darlo a probar a usuarios, anotar las mejoras e implementar esas mejoras.

Scrum con su estructura de sprints y revisiones constantes ofrece la experiencia ideal para el desarrollo del chatbot. Para este proyecto se han pensado los siguientes parámetros:

- **Duración sprint:** se establecen 2 semanas por sprint.
- **Número de sprints:** se realizarán 8 sprints.

Al final de cada sprint se revisarán las interacciones que los usuarios han tenido con el chatbot y anotarán todas las mejoras a tener en cuenta en el siguiente sprint.

El desarrollo del proyecto se hace durante el segundo cuatrimestre del curso 2023-2024. Siendo el comienzo del primer sprint el día 26 de febrero. Debido a la compatibilización del proyecto con mi jornada laboral, la dedicación no será completa, estimando unas 2.5 horas en días laborables y entre 5 y 6 horas los fines de semana. Esta distribución de tiempo me permitirá alcanzar más de las 300 horas requeridas para el TFG, finalizando el último sprint el 16 de junio. Aún se tendría unos días de margen para la entrega ordinaria y proporciona varias semanas por si fuese necesario optar por una entrega extraordinaria.

En el próximo capítulo detallaré el proceso de desarrollo en cada sprint. Esta sección proporcionará una visión clara de cómo se ha ido construyendo el proyecto semana a semana, destacando

los logros clave, los desafíos enfrentados y las soluciones implementadas en cada etapa.

3.2. Riesgos

En esta sección abordaré la gestión de riesgos detectados en el proyecto, identificando los riesgos principales evaluando su impacto y probabilidad.

Entendemos por riesgo cualquier evento que pueda ocurrir y que, de hacerlo, tendría un efecto adverso en la consecución de los objetivos del proyecto. La gestión de estos riesgos implica identificarlos tempranamente, evaluar su impacto y probabilidad de ocurrencia y desarrollar estrategias para mitigarlos.

En cada riesgo se detalla una descripción, las estimaciones de su impacto y probabilidad antes y después de implementar las medidas propuestas. Los valores de la probabilidad Tabla 3.1 y los del impacto se evaluarán Tabla 3.2 con los siguientes descriptores cualitativos: bajo, moderado, alto, extremo.

Nivel de probabilidad	**Gama**
Alta	Probabilidad superior al 50 %
Significativa	30-50 % de posibilidades de que ocurra
Moderada	10-29 % de posibilidades de que ocurra
Baja	Menos del 10 % de posibilidades de que ocurra

Tabla 3.1: Niveles de probabilidad

Nivel de probabilidad	**Gama**
Alta	Más del 30 % por encima de lo presupuestado gastos presupuestados
Significativa	Del 20 al 29 % por encima del gasto presupuestado
Moderada	Del 10 al 19 % por encima de los gastos presupuestados
Baja	Dentro del 10 % de los gastos presupuestados

Tabla 3.2: Niveles de probabilidad

A continuación, se enumeran los principales riesgos.

Ri01	**Falta de experiencia**	
Descripción	No se tienen los conocimientos suficientes para un abarcar el desarrollo marcado en los sprints	
Plan de Mitigación	Formación durante las primeras semanas de proyecto. Aprender conceptos de chatbots, NLU y Rasa framework.	
Plan de Contingencia	Incorporar un perfil especializado en la materia de una empresa externa o freelancer que pueda resolver las dudas y ayudar a progresar.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Moderada	Significativa
Después	Baja	Baja

Tabla 3.3: Riesgo 01 - Falta de experiencia

Ri02	**Falta de usuarios**	
Descripción	No hay un número de usuarios significativo provando el chatbot y aportando el feedback.	
Plan de Mitigación	Crear una campaña de comunicación que explique los beneficios y el uso del chatbot. Quizás añadir algún incentivo.	
Plan de Contingencia	Organizar talleres en vivo o webinars en redes sociales.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Moderada	Significativa
Después	Baja	Moderada

Tabla 3.4: Riesgo 02 - Falta de usuarios

Ri03	**Pérdida de código**	
Descripción	Pérdida de código	
Plan de Mitigación	Utilizar al menos dos controles de versión en servidores distintos.	
Plan de Contingencia	Recuperar la versión más reciente y continuar el trabajo desde esa versión.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Moderada	Alta
Después	Baja	Baja

Tabla 3.5: Riesgo 03 - Pérdida de código

Ri04	**Telegram sin servicio**	
Descripción	El principal canal de comunicación con el chatbot se realiza a través de Telegram. Si los servidores de Telegram fallan o son bloquedaos en España nos quedaríamos sin poder comunicarnos al bot.	
Plan de Mitigación	Adaptar el chatbot para funcionar también en otras plataformas de mensajería como WhatsApp o Facebook Messenger, asegurando la continuidad del servicio.	
Plan de Contingencia	Publica el asistente en otra plataforma de mensajería.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Baja	Significativa
Después	Baja	Baja

Tabla 3.6: Riesgo 04 - Telegram sin servicio

Ri05	**Enfermedad**	
Descripción	Enfermedad que obligue al programador a para el trabajo temporalmente.	
Plan de Mitigación	Políticas de trabajo flexible.	
Plan de Contingencia	Reestimar las tareas y ajustar la planificación.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Baja	Significativa
Después	Baja	Moderada

Tabla 3.7: Riesgo 05 - Enfermedad

Ri06	**Complejidad elevada**	
Descripción	Complejidad de los diálogo es demasiado alta.	
Plan de Mitigación	Comenzar con pequeñas mejoras en la conversación con el chatbot en vez de intentar abarcar diálogos extensos y más complejos.	
Plan de Contingencia	Evaluar la tarea y ajustar la planificación.	
Valores Probabilidad/Impacto		
	Probabilidad	Impacto
Antes	Moderada	Significativa
Después	Baja	Moderada

Tabla 3.8: Riesgo 06 - Complejidad elevada

3.2.1. Matriz de riesgos

En la siguiente Tabla 3.9 podemos observar la distribución en la matriz de (impacto/probabilidad) los riesgos identificados.

Probabilidad/Impacto	**Baja**	**Moderada**	**Significativa**	**Alta**
Alta				
Significativa				
Moderada			Ri01,Ri02,Ri06	Ri03
Baja				Ri04,Ri05

Tabla 3.9: Matriz de riesgos

Tras la aplicación de las medidas de mitigación para reducir la probabilidad de ocurrencia del riesgo o reducir el impacto que pueda causar, la matriz de riesgos quedaría de la siguiente forma:

Probabilidad/Impacto	**Baja**	**Moderada**	**Significativa**	**Alta**
Alta				
Significativa				
Moderada		Ri02,Ri05,Ri06		
Baja	Ri01,Ri03,Ri04			

Tabla 3.10: Matriz de riesgos tras los planes de mitigación.

3.3. Distribución temporal

La división por sprints sería la siguiente:

- **Sprint 1 (30 horas):** Se realizó una reunión inicial con el tutor D. César González Ferreras para discutir los objetivos y expectativas del proyecto. Durante esta sesión, se me presentó el framework Rasa el cuál debía estudiar y seleccionar un ámbito en el que aplicarlo para crear un chatbot. Durante los días siguientes, me dediqué a familiarizarme con los conceptos de Rasa y explorar las herramientas que tiene. Tras evaluar las posibilidades de Rasa, finalmente decidí que el chatbot se centraría en proporcionar información sobre películas.

- **Sprint 2 (35 horas):** En el Sprint 2, comencé a explorar el framework Rasa de forma práctica. Configuré el entorno del proyecto e instalé todas las dependencias necesarias para empezar. Durante este periodo revisé los conceptos fundamentales de Rasa para asegurarme de entenderlos bien (ya que era mi introducción a un proyecto relacionado con chatbots y NLP) y simultáneamente probar estos conceptos con el chatbot por defecto que ofrece Rasa.

- **Sprint 3 (63 horas):** En el sprint 3 comencé a desarrollar la primera versión funcional del chatbot. Este prototipo inicial era sólo era capaz de identificar cuando un usuario solicitaba una recomendación. Tras identificar la intención del usuario como recomendación, el bot preguntaba al usuario por el género de película que buscaba y guardaba la respuesta para procesarla adecuadamente. Además, en este punto conseguí explotar e integrar el uso de la API de la web TMDB dentro del proyecto, de donde extraería la información proporcionada al usuario sobre películas. Finalmente, tengo una segunda sesión con el tutor para presentarle los avances y valoramos otras funcionalidades que se pueden aplicar al chatbot. Para facilitar el acceso al chatbot lo conecto a Telegram. Le doy a probar el chatbot a un círculo cercano de amigos y familiares para observar las primeras interacciones hombre-máquina.

- **Sprint 4 (50 horas):** Al comienzo del sprint 4 analicé las interacciones del chatbot para identificar posibles patrones y mejoras al a hora de identificar el género. Detecté un primer problema clave: los géneros solicitados por los usuarios necesitan coincidir exactamente con los géneros definidos en la API de TMDB. Para solucionarlo implementé una tabla lookup con todas las opciones posibles de género y añadí sinónimos a los distintos géneros según la

forma en la que los usuarios se refieren a ese género que surgen de las conversaciones reales, mejorando así la precisión a la hora de identificar el género. También apliqué el patrón formulario para que cuando se le pregunte el género y el usuario no responda correctamente, se le vuelva a preguntar. Vuelvo publicar un segundo prototipo de chatbot.

- **Sprint 5 (48 horas):** En el sprint 5 revisé los resultados de las interacciones en el chatbot con los nuevos cambios, notando una mejoría respecto a la versión anterior. Realizo otra sesión con el tutor para mostrar el avance y plantear nuevas funcionalidades. En este punto investigo más endpoints que ofrece la API para ofrecer más funcionalidad. Comienzo por escribir dos story relacionadas con la película que se le ha recomendado, permitiendo a los usuarios solicitar un resumen de la película, preguntar por los actores principales y quién dirigió la película. A la hora de responder sobre los actores y directores se le muestra una listado limitado. Publiqué una tercera versión del chatbot.

- **Sprint 6 (54 horas):** En el sprint 6 volví a revisar las interacciones con el chatbot tras la nueva actualización. Me di cuenta que en ocasiones el bot se perdía en la conversación, tanto a la hora de responder como de interpretar bien que intención estaba expresando el usuario. Para solucionar este problema añadí más stories de ejemplo para que el modelo conversacional aprendiera de ellos. También implemente alguna funcionalidad extra, integre los endpoints para preguntar a la API sobre información de los actores y directores, y en que otras películas trabajaron. En una primera aproximación quería utilizar una modelo con nombres de personas pre-entrenados, como los que ofrece spaCy pero al pertenecer los nombres a multiples idiomas se complicaba. Al final opté por ofrecer en la respuesta una lista de botones con los nombres de los actores, el usuarios al seleccionar uno se le daba información de ese actor. Publiqué una cuarta versión del chatbot.

- **Sprint 7 (35 horas):** En el sprint 7 volví a analizar las interacciones con el chatbot. En este punto el principal problema es sería saber guiar al chatbot por el flujo de las conversaciones y adaptarse a respuestas inesperadas que no estaban controladas. Esto supone más tiempo de investigación como funcionan los distintos pipelines y policies para ajustar el chatbot. Publico una versión.

- **Sprint 8 (40 horas):** En este último sprint se da por finalizado el chatbot tras aplicar las últimas mejoras en el control de respuestas inesperadas y añadir más ejemplos que se extraen de las interacciones. En este sprint se elabora la memoria del trabajo desarrollado.

El tiempo de todas las interacciones sumaría un total de 355 horas.

3.4. Costes

El Trabajo de Fin de Grado se ha desarrollado en 355 horas. Siendo el rango salarial de un desarrollador Backend con experiencia en Python entre los 20-30K [16]. Si tomamos un salario medio anual dentro del anterior rango, 25.000€ al año en el que se trabajan 1.800 horas anuales,

según convenio colectivo de consultorías [2]. Tenemos que el coste hora serían 13,89 €/hora. El coste total del desorrallador sería de **4.930€**.

Ya que el proyecto se ha realizado desde el domicilio del trabajador podría optar a un plus de trabajo flexible y ayuda comedor, alrededor de unos 120 € mensuales. Para los 4 meses que ha supuesto el desarrollo del proyecto hace un total de 480€.

Coste total: 5.410€.

Capítulo 4

Descripción iteraciones

En este capítulo se va a describir en detalle el desarrollo realizado en cada uno de los sprints. Además, expondrán los desafíos encontrados durante el desarrollo de los diálogos del chatbot y las estrategias empleadas para resolverlos.

4.1. Sprint 1

En el primer sprint del proyecto tuve una reunión inicial con mi tutor D. César González Ferreras, donde discutimos los objetivos y expectativas del Trabajo de Fin de Grado. Durante esta sesión, fui introducido al framework Rasa, que iba a utilizar para desarrollar el chatbot.

Los siguientes días me enfoque en estudiar el framework Rasa y familiarizarme con los conceptos clave, los cuales he descrito en Capítulo 2.1. Mi principal recurso para el aprendizaje fue la documentación oficial proporcionada por Rasa [10], que es muy útil y completa. Fuera de la documentación me costó encontrar recursos tan útiles. También tomé lecciones y webinars que ofrece el equipo de Rasa en su canal de Youtube [14].

Tras evaluar las posibilidades que ofrece Rasa y considerar diferentes aplicaciones posibles, decidí que el chatbot se especializara en el dominio del cine. Elegí este campo debido a su amplio alcance y la disponibilidad de datos a través de diversas APIs, además de ser un tema de gran interés debido a la popularidad de las plataformas de streaming. Esta dirección no solo satisfacía los requisitos del proyecto, sino que también sería de gran utilidad para un público amplio y diverso.

Investigué varias APIs para proporcionara datos sobre películas y series. Finalmente me decidí por la API de TMDB. Esta API destacó por su amplia variedad de endpoints que se adaptan perfectamente al objetivo del chatbot. Ofrece acceso a una rica base de datos de películas y series, incluyendo detalles como sinopsis, calificaciones, elenco y mucho más.

flake.nix

```
{
  description = "A Nix-flake-based Python development environment";

  inputs.nixpkgs.url = "https://flakehub.com/f/NixOS/nixpkgs/0.1.*.tar.gz";

  outputs = { self, nixpkgs }:
    let
      supportedSystems =
        [ "x86_64-linux" "aarch64-linux" "x86_64-darwin" "aarch64-darwin" ];
      forEachSupportedSystem = f:
        nixpkgs.lib.genAttrs supportedSystems
        (system: f { pkgs = import nixpkgs { inherit system; }; });
    in {
      devShells = forEachSupportedSystem ({ pkgs }: {
        default = pkgs.mkShell {
          name = "tfg";
          venvDir = ".venv";
          packages = with pkgs;
            [ python310 ]
            ++ (with pkgs.python310Packages; [ pip venvShellHook ]);
          postShellHook = ''
            export LD_LIBRARY_PATH=${pkgs.stdenv.cc.cc.lib}/lib
          '';
        };
      });
    };
}
```

Figura 4.1: Configuración del entorno virtual con nix y flake

4.2. Sprint 2

En este segundo sprint quería profundizar en los conceptos del framework Rasa de forma práctica. Lo primero que necesitaba era un entorno de trabajo.

Inicié la configuración del entorno utilizando herramientas nix (y su utilidad flake) y direnv, que últimamente aplico a todos mis proyectos. Estas herramientas garantizan la creación de un entorno de desarrollo consistente y reproducible, independientemente de la máquina utilizada, incluyendo todas las dependencias del sistema como la versión de Python y librerías necesarias, sin alterar la configuración global de mi terminal. Los ficheros asociados a estas herramientas son flake.nix 4.1, flake.lock y .envrc.

Para manejar las librerías de Python que necesito, utilicé poetry, herramienta para gestionar las dependencias dentro del entorno virtual. La configuración de poetry para este proyecto se describe en los ficheros pyproject.toml 4.2 y poetry.lock.

Al comienzo en el proyecto utilizaba la última versión del framework Rasa y Python 3.11 pero semanas posteriores al integrar Telegram, para conectar el chatbot al servicio de mensajería, la librería que ofrece Rasa tenía bugs reportados [5]. Con este inconveniente, tuve que reconfigurar las dependencias, utilizando Python 3.10 y la versión de Rasa 3.6.6. Decir que utilizo la versión de Rasa más simple, sin los extras que ofrece para las librerías spaCy, por ejemplo. Otros paquetes de Python que uso en el proyecto es request, que lo utilizo para hacer las llamadas al endpoint de TMDB.

Figura 4.2: Dependencias Python gestionadas por poetry

Una vez con todo configurado, pude iniciar un proyecto Rasa usando el comando `rasa init`. El cli de Rasa ofrece el comando `rasa init` para crear un proyecto con todos los ficheros necesarios, configuraciones por defecto y con datos listos para entrenar el chatbot. De este modo, pude experimentar con un chatbot real, bastante sencillo, todos los conceptos que estuve estudiando en la documentación.

Con los datos del chatbot listos podemos ir probándolo. Para entrenar el modelo ejecutamos el comando `rasa train`. Este comando genera la carpeta models dentro de nuestro proyecto y un fichero .zip con todos los archivos que necesita Rasa para el modelo. Rasa aplica entrenamiento con caché, si existen modelos en el directorio models/, sólo se entrenarán las partes de tu modelo que hayan cambiado. Por ejemplo, si sólo has cambiado alguna rule en los datos de entrenamiento, se generarán los nuevos ficheros para esa rule el resto lo toma de de la caché [10].

Para empezar a charlar con el chatbot debemos ejecutar el comando `rasa shell`, así podremos mantener una conversación con el modelo. Para montar un servicio donde exponer el modelo entrenado y charlar con él se ejecuta `rasa run`, esto habilita el puerto 5005 (puerto por defecto), y conectarlo con otros servicios de terceros: aplicaciones de mensajería, páginas web, etc.

4.3. Sprint 3

En el tercer sprint utilizando como plantilla el chatbot por defecto comienzo a crear los primeros intents, reponses y rules. Para este tercer sprint el objetivo sería diseñar un chatbot capaz de:

- Saludar: El chatbot debe comprender una serie de expresiones como saludo y responder con un saludo.
- Despedirse: El chatbot debe entender cuando el usuario desea despedirse y responder con un mensaje de despedida.
- Recomendar películas: El chatbot debe entender cuando el usuario desea que se le recomiende

una película, preguntarle por el género, entender que se le informa el género e identificarlo y guardarlo en el slot.

4.3.1. Saludar

El chatbot debe comprender una serie de expresiones como saludo y responder con un saludo. Diagrama: 4.3.

Para lograr esta interacción primero configuro el intent `start`, la respuesta `utter_start` y la historia `Start`.

Se podría conseguir el mismo resultado con una rule, pero al final me decidí por utilizar una story ya que no tiene mucho sentido saludar en el transcurso de una conversación, sólo al principio de una conversación.

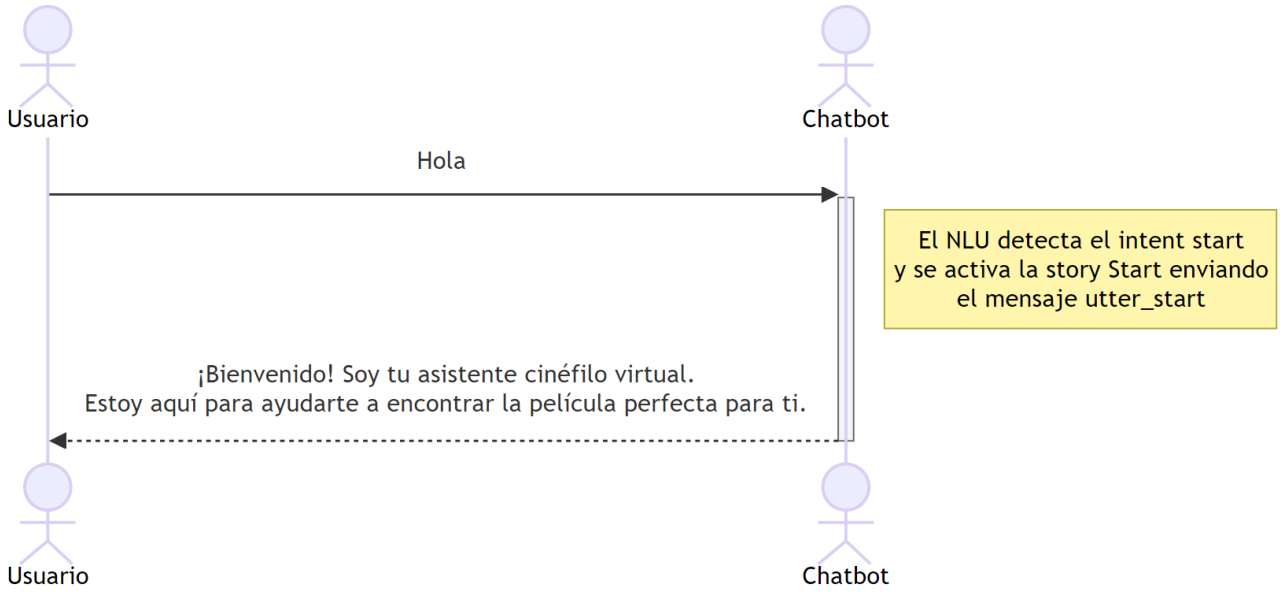

Figura 4.3: Diagrama de secuencia de la historia: Saludar

```
nlu:
- intent: start
  examples: |
    - /start
    - Hola
    - hola
    - Saludos
stories:
- story: Start
  steps:
  - intent: start
  - action: utter_start
responses:
  utter_start:
```

```
15   - text:   Bienvenido  ! Soy tu asistente cinéfilo virtual. Estoy aquí para
       ayudarte a encontrar la película perfecta para ti.
```

Código 4.1: Datos de entrenamiento para Saludar.

4.3.2. Despedirse

El chatbot debe entender cuando el usuario desea despedirse y responder con un mensaje de despedida. Diagrama: 4.4.

Para lograr esta interacción primero configuro el intent `goodbye`, la respuesta `utter_goodbye` y la rule `Goodbye`.

En esta ocasión he optado por una regla ya que el usuario puede tomar la decisión de despedirse en cualquier momento de la conversación.

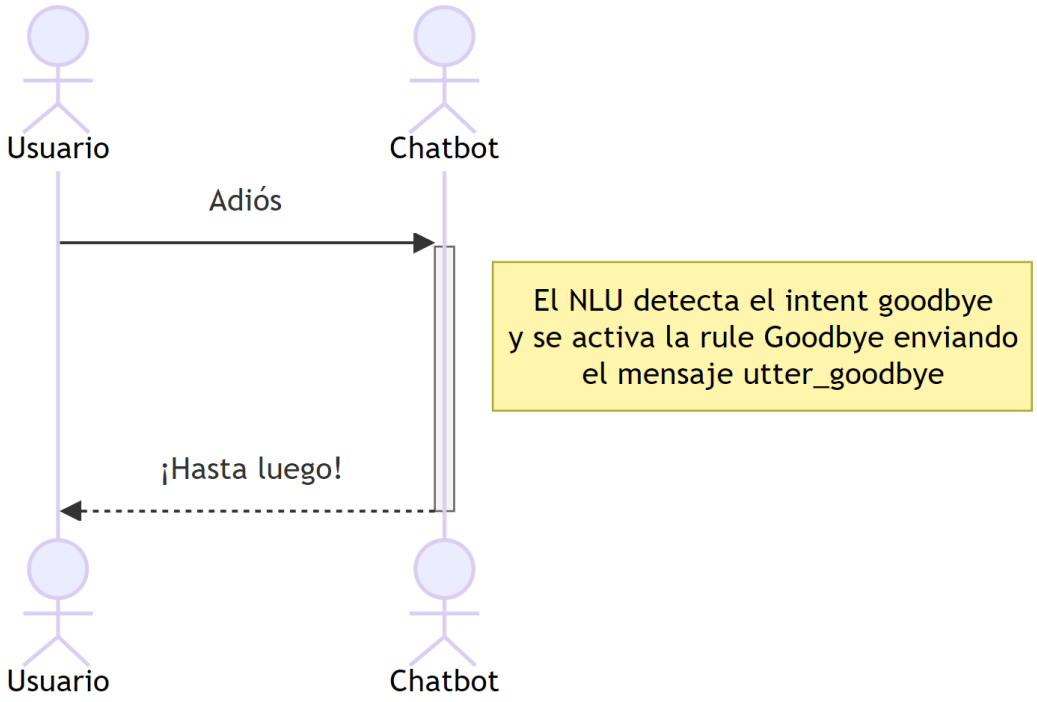

Figura 4.4: Diagrama de secuencia de la regla: Despedirse

```
nlu:
- intent: goodbye
  examples: |
    - Adiós
    - Hasta luego
    - Hasta la próxima
    - Hasta pronto
rules:
- rule: Goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
```

```
responses:
  utter_goodbye:
  - text: Hasta luego
```

Código 4.2: Datos de entrenamiento para Despedirse.

4.3.3. Recomendar películas

El chatbot debe entender cuando el usuario desea que se le recomiende una película, preguntarle por el género, entender que se le informa el género e identificarlo y guardarlo en el slot. Diagrama: 4.5

Primero definimos los dos intents que debemos detectar en la interacción con los usuarios `request_movie` y `inform_genre`. En el primer intent entramos al chatbot con un cuatro ejemplos de frases que diría un usuario para solicitar una película. El segundo intent a parte de aportar expresiones que un usuario usaría para aportar información sobre el género deseado se añade 3 ejemplos del entity `movie_genre`. Por el momento el chatbot solo sería capaz de identificar estos 3 géneros: terror, suspense y drama.

También creo una acción personalizada que se lanzará tras interpretar el género que se encargará llamar al endpoint correspondiente de TMDB. Para facilitar la llamada a los endpoints de API de TMDB diseño y creo una clase en Python, llamada TMDB, que servirá de wrapper para hacer las llamadas a los endpoints. Mi acción personalizada emplea el endpoint */discover/movie* el cuál ofrece la posibilidad de buscar películas utilizando más de 30 filtros. Aplicaré los siguiente filtros en la búsqueda: el género extraído del slot `movie_genre` y una puntuación mayor que 8.

```
nlu:
- intent: request_movie
  examples: |
    - ¿Qué película me recomiendas?
    - Quiero ver una película
    - Recomiéndame una película
    - ¿Podrías sugerirme una película?
- intent: inform_genre
  examples: |
    - Me gustan las películas de [terror](movie_genre)
    - Prefiero las películas de [suspense](movie_genre)
    - Me apetece ver un [drama](movie_genre)
entities:
- movie_genre
slots:
  movie_genre:
    type: text
    mappings:
    - type: from_entity
      entity: movie_genre
stories:
- story: Request movie
  steps:
```

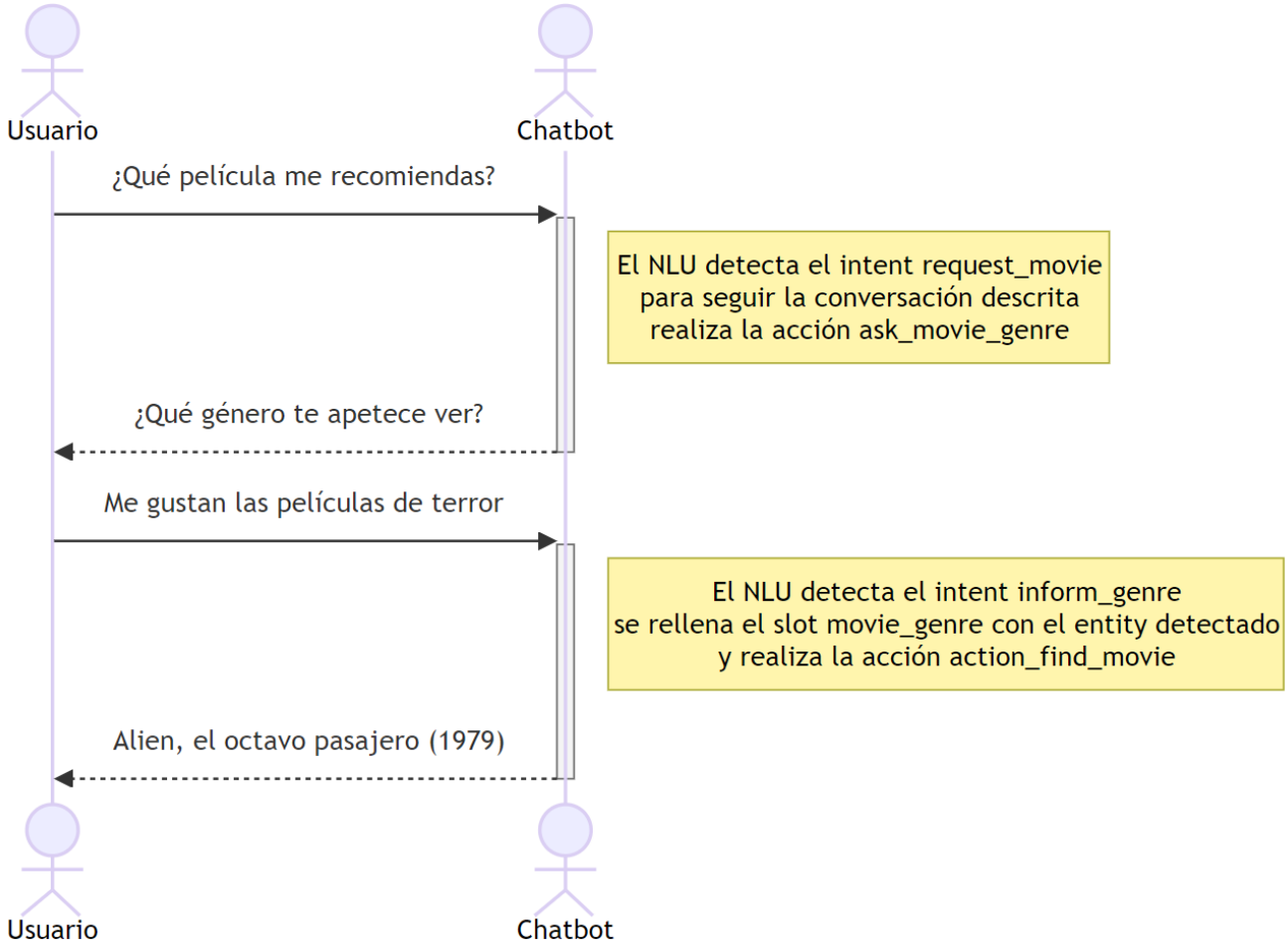

Figura 4.5: Diagrama de secuencia de la historia: Recomendar películas

```
24   - intent: request_movie
25   - action: ask_movie_genre
26   - intent: inform_genre
27   - action: action_find_movie
28 responses:
29   ask_movie_genre:
30   - text: ¿Tienes algún género o tipo de película en mente?
31   - text: ¿Qué género te apetece ver?
```

Código 4.3: Datos de entrenamiento para Despedirse.

4.3.4. Telegram

Para permitir que usuarios reales interactuaran con el chatbot, era esencial integrarlo con un servicio de mensajería. Esta conexión facilitaría la interacción directa con el chatbot en un entorno familiar para los usuarios, aprovechando las plataformas de mensajería ya disponibles. Rasa ofrece varias opciones pero decidí utilizar Telegram como servicio de mensajería.

La integración fue sencilla, siguiendo los pasos descritos en la documentación oficial de Rasa [10]. Para crear un bot en Telegram basta con establecer una conversación con el bot `BotFather` que te asiste en la creación de bots. A mi bot le llamo `@TFGRasaBot`. Tras obtener el token del bot lo copio en el fichero `config.yml` dentro del proyecto Rasa para establecer la comunicación automáticamente.

Por último es necesario establecer una conexión entre los servidores de Telegram y el servicio local de Rasa que ejecuta el modelo entrenado de del chatbot. Utilicé ngrok para esta tarea, en su versión gratuita, que me permite crear un túnel desde internet hacia mi máquina local (que de otro modo no sería accesible desde internet) utilizando una URL.

Aunque existen alternativas como contratar un servidor y un dominio público, opté por usar ngrok por su simplicidad y para minimizar los costes en este proyecto. Esto me permitió concentrarme en desarrollar y mejorar el chatbot sin preocuparme por la infraestructura adicional. Comando usado para levantar el servicio de ngrok:

```
ngrok http --domain=trusted-monitor-uniquely.ngrok-free.app 5005
```

El flujo final sería el siguiente, Diagrama 4.6:

- El usuario envía un mensaje al desde su aplicación de Telegram a TFGRasaBot.
- TFGRasaBot reenvía el mensaje de los servidores de Telegram que
- Los servidores de Telegram a su vez reenvían el mensaje a la url del parámetro `webhook_url`. Esta url pública, me la proporciona ngrok.
- Ngrok redirige el tráfico a mi servicio local de Rasa que ejecuta el modelo entrenado de del chatbot.

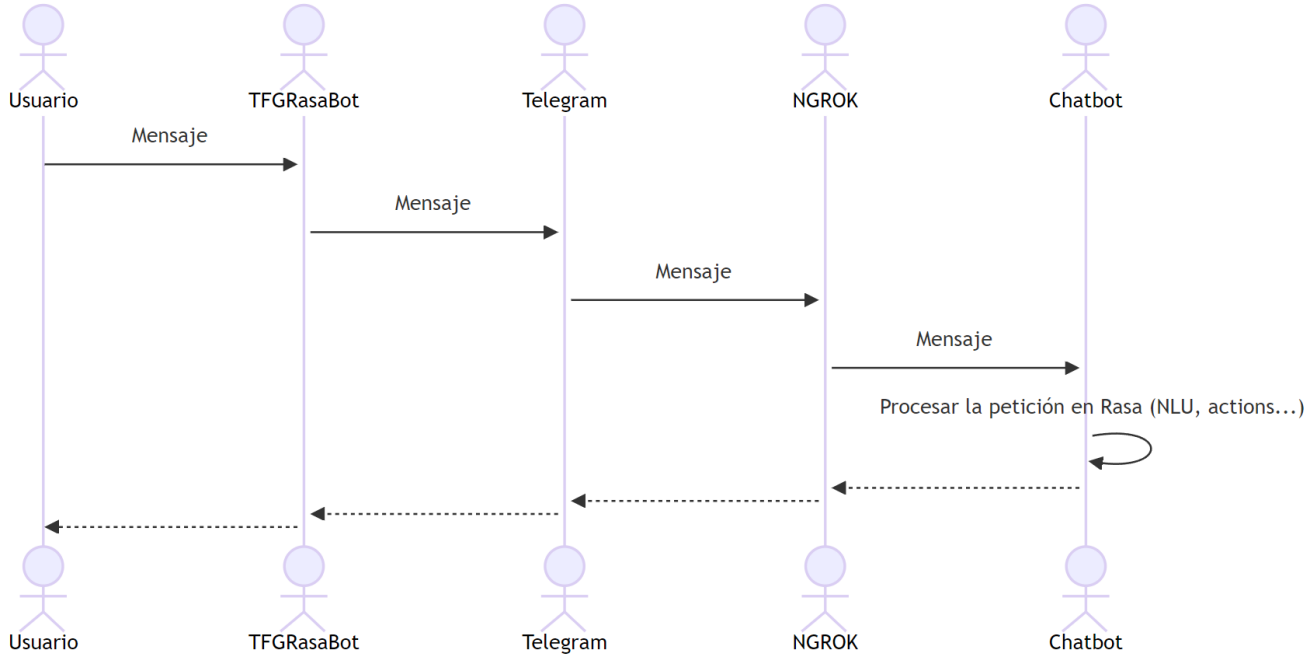

Figura 4.6: Arquitectura durante el desarrollo

- Chatbot procesa el mensaje y ejecuta la acciones necesarias retornando el mensaje de respuestas.

4.3.5. Tracker

Tras publicar el chatbot se hacía necesario almacenar las interacciones que los usuarios tienen con él, para poder analizar las conversaciones más tarde y mejorar los datos de entrenamiento. Rasa ofrece diversas opciones para registrar estas conversaciones. Para mi proyecto, opté por utilizar MongoDB, una base de datos no SQL. Para levantar el servicio de MongoDB utilizo docker. Entre los archivos del proyecto existe un `compose.yml` con dos servicios:

- MongoDB: base de datos No-SQL, encargada de almacenar las conversaciones.
- Mongo Express: es una interfaz web para manejar instancias de MongoDB.

Finalmente lanzar el chatbot, es necesario ejecutar algunos comandos en la terminal. Primero utilizo el comando `rasa run -m models --endpoints endpoints.yml` para poner en marcha el servicio con el modelo Rasa entrenado. Paralelamente, en otro terminal, ejecuto `rasa run actions` que se encargará de exponer las acciones personalizadas en el puerto 5055 para el uso del chatbot.

Después de configurar todo, la primera versión del chatbot ya está lista para ser probada por usuarios reales. Esta fase inicial se trata de identificar y corregir posibles errores.

4.4. Sprint 4

Al comienzo del sprint 4, tras un par de días de pruebas, dediqué tiempo a examinar las conversaciones de usuarios reales guardadas en la base de datos MongoDB. El formato en que Rasa guarda esta información en el tracker puede parecer algo caótico pero al final te haces a la idea. Sin embargo, recomendaría utilizar alguna herramienta de análisis de datos para facilitar y mejorar el análisis.

Las conclusiones tras analizar las conversaciones fueron:

- Mejorar el intent `request_movie`: no siempre que el usuario transmitía su intención de solicitar una película el chatbot no lo interpretaba así.
- Mejorar el intent `inform_genre`: no tengo todos los géneros posibles descritos en los datos de entrenamiento, aunque el intent era identificado por Rasa no podía extraer el entity `movie_genre`.
- Mejorar el flujo: a veces el usuario pedía una recomendación pero en vez de responder el género, el usuario responde otra cosa, y se pierde el flujo de story.

4.4.1. `request_movie`

Para mejorar la capacidad de Rasa para identificar el intent `request_movie` sería necesario añadir más ejemplos de petición. Añado varios ejemplos más sacados de conversaciones reales y otros nuevos.

```
nlu:
- intent: request_movie
  examples: |
    - ¿Qué película me recomiendas?
    - Quiero ver una película
    - Recomiéndame una película
    - ¿Podrías sugerirme una película?
    - ¿Tienes alguna recomendación de película?
    - ¿Qué película me sugieres?
    - Dime una película
    - Me recomiendas una película
    - Me dices una película
    - ¿Qué película puedo ver?
    - Sugiéreme una película
    - Dame una película para ver
    - Ayúdame a encontrar una película
    - Dame una buena película
    - ¿Puedes recomendarme alguna película?
    - Quiero ver una buena película
    - ¿Cuál película me sugieres?
```

Código 4.4: Datos de entrenamiento Petición de película.

4.4.2. `inform_genre`

Para solucionar los problemas a la hora de identificar los géneros decido crear una tabla `lookup` que contenga todos los géneros disponibles en la API de TMDB. Varios géneros se pueden ser informados de varias formas, por ejemplo, en vez de género Romance el usuario podría decir *Una película romántica*. O en vez de Western el usuario podría referirse a este género como *Una del oeste* o *Una película de vaqueros*.

Esta lista se irá mejorando y ampliando con los distintas versiones que usen los usuarios para referirse a un género.

```
nlu:
- synonym: Western
  examples: |
    - Vaqueros
    - Oeste
    - Indios
    - Indios y Vaqueros
- synonym: Acción
  examples: |
    - Accion
- synonym: Crimen
  examples: |
    - Crimenes
- synonym: Romance
  examples: |
    - Romántica
- lookup: movie_genre
  examples: |
    - Acción
    - Aventura
    - Animación
    - Comedia
    - Crimen
    - Documental
    - Drama
    - Familia
    - Fantasía
    - Historia
    - Terror
    - Música
    - Misterio
    - Romance
    - Ciencia ficción
    - Suspense
    - Bélica
    - Western
```

Código 4.5: Datos de entrenamiento. Lista de géneros.

4.4.3. Formulario

Para mejorar el flujo de recopilación de la información implmentaré un `forms` en el chatbot. Los `forms` son una herramienta esencial para estructurar la recogida de datos. Primero defino el formulario con los `slots` necesarios y las `reponses` de cada slot, es decir, la pregunta que hace el chatbot al usuario para que responda con la información necesaria para rellenar el `slot`.

Explicaré este punto más en detalle. Rasa admite varias opciones para informar como debe formularse la pregunta al usuario a la hora de rellenar un slot. Las opciones siguen los siguientes patrones y orden:

1. action_ask_<form_name>_<slot_name>
2. utter_ask_<form_name>_<slot_name>
3. action_ask_<slot_name>
4. utter_ask_<slot_name>

También es necesario añadir un par de reglas para manejar el comportamiento del formulario. Una regla para activar el formulario y otra para indicar que se finaliza el formulario.

Para activar el formulario primero se define la acción `movie_form`, que lleva el nombre del formulario, y luego se activa el bucle del formulario con `active_loop: movie_form`. Esto mantiene la conversación en bucle repitiendo las preguntas de los slots hasta que el usuario haya rellenado todos. El formulario acaba cuando todos los slots están llenos, la variable `active_loop` se pone a null, y la variable donde se indica cuál es el siguiente slot a rellenar, `requested_slot` se pone a null.

Por último, hace falta hacer algunas modificaciones a las acciones personalizadas. Para validar los valores del usuario es necesarios añadir validaciones personalizadas a cada slot que se declara en el formulario. Para validar el género consulto si el valor de `movie_genre` se encuentra en la lista de géneros en el endpoint */genre/movie/list* en la API TMDB, si el género está disponible, almaceno el id en el slot `user_genre` para utilizarlo en futuras consultas a la API. Si el género no es válido se vuelve a poner el slot a null para que el chatbot lance de nuevo la pregunta. Adicionalmente he añadido un mensaje para informar la usuario de que el género no es correcto.

```
entities:
- movie_genre
slots:
  user_genre:
    type: any
    mappings:
    - type: custom
  user_movies_history:
    type: list
    mappings:
    - type: custom
```

```
  movie_genre:
    type: text
    mappings:
    - type: from_entity
      entity: movie_genre
      conditions:
      - active_loop: movie_form
rules:
- rule: Activate form
  steps:
  - intent: request_movie
  - action: movie_form
  - active_loop: movie_form
- rule: Finish form
  condition:
    - active_loop: movie_form
  steps:
  - action: movie_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  - action: action_find_movie
forms:
  movie_form:
    required_slots:
      - movie_genre
responses:
  utter_ask_movie_genre:
  - text: ¿Tienes algún género o tipo de película en mente?
  - text: ¿Qué género te apetece ver?
```

Código 4.6: Datos de entrenamiento. Formulario movie_form

He creado un nuevo slot `user_movies_history` que me servirá para almacenar los ids de las películas recomendadas al usuario para evitar repetir las mismas.

También añadí más información al mensaje de respuesta que enviaba el chatbot al usuario sobre la película recomendada. Ahora envía, una confirmación del género que busca, una imagen con la portada de la película, el título de la película junto con el año de estreno y la puntuación de la película. Conversación: 4.7

Vuelvo publicar el chatbot con todas estas mejoras para que hagan pruebas usuarios reales.

4.5. Sprint 5

Al comienzo de este sprint revisé de nuevo las conversaciones con usuarios reales. Los resultados mejoraron bastante. Añadí algún sinónimo de género más.

Tras mostrarle los avances al tutor D. César González Ferreras, me propone mejorar la funcio-

Figura 4.7: Ejemplo de conversación

nalidad del chatbot incluyendo nuevas funcionalidades. Lo primero que reviso es la API de TMDB para ver que opciones me ofrece para añadir más funcionalidad.

Decido incluir dos nuevas funcionalidades:

- Solicitar resumen: El chatbot debe comprender cuando se le solicita un resumen de la película y mostrar un mensaje de texto con el resumen. Se tomará el id de la última película guardada en el historial.
- Solicitar actores: El chatbot debe comprender cuando se le solicita que actores participan en la película y mostrar un listado de los principales. Se tomará el id de la última película guardada en el historial.
- Solicitar director: El chatbot debe comprender cuando se le solicita que director dirige la película y mostrarlo. Se tomará el id de la última película guardada en el historial.

En todas estas historias se trabaja sobre la premisa que el usuario primero ha solicitado una recomendación de película antes de solicitar un resumen o preguntar sobre los actores o directores. Como el comienzo de las 3 historias sería el mismo, la solicitud de una recomendación, utilizo la opción `checkpoint` para modularizar y simplificar los datos de entrenamiento [10]. Estos puntos de control son muy útiles, pero no hay que abusar de ellos ya que ralentizará el entrenamiento del modelo.

En todas las historias se llama a acciones personalizadas que utilizan endpoints de la API de TMDB. En el caso de que se haya recomendado más de una, estas acciones personalizadas siempre toman el id de la última película guardada en el historial (slot `user_movies_history`).

4.5.1. Solicitar resumen

El chatbot debe comprender cuando se le solicita un resumen de la película y mostrar un mensaje de texto con el resumen.

Para comprender cuando el usuario quiere solicitar un resumen de la película he creado el intent `request_movie_info`.

```
nlu:
- intent: request_movie_info
  examples: |
    - ¿De qué trata?
    - ¿Me haces un resumen?
    - ¿Cuál es la sinopsis?
    - Quiero saber más sobre esta película
    - ¿Qué pasa en la película?
    - ¿Cuál es el argumento de la película?
    - ¿Puedes darme detalles sobre la trama?
    - ¿De qué va la película?
    - ¿Me puedes dar un resumen de la película?
```

```
    - Cuéntame de qué trata la película
stories:
- story: User request movie
  steps:
  - intent: request_movie
  - action: movie_form
  - active_loop: movie_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  - action: action_find_movie
  - checkpoint: ask_about_movie

- story: User request movie info
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_info
  - action: action_summary_movie
```

Código 4.7: Datos de entrenamiento. Solicitar resumen.

4.5.2. Solicitar actores

El chatbot debe comprender cuando se le solicita que actores participan en la película y mostrar un listado de los principales.

Para comprender cuando el usuario quiere solicitar un resumen de la película he creado el intent `request_movie_cast`.

```
nlu:
- intent: request_movie_cast
  examples: |
    - ¿Quiénes son los actores principales?
    - ¿Quién actúa en la película?
    - ¿Que actores salen?
    - ¿Quiénes son los miembros del reparto?
    - Dime el elenco de la película
    - ¿Quiénes forman el reparto?
    - ¿Quiénes son los actores?
stories:
- story: User request movie cast
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_cast
  - action: action_movie_cast

- story: User request movie director
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_director
```

```
22   - action: action_movie_director
```

Código 4.8: Datos de entrenamiento. Solicitar actores.

4.5.3. Solicitar director

El chatbot debe comprender cuando se le solicita que director dirige la película y mostrarlo.

Para comprender cuando el usuario quiere solicitar un resumen de la película he creado el intent `request_movie_director`.

```
1  nlu:
2  - intent: request_movie_director
3    examples: |
4      - ¿Quién dirige la película?
5      - ¿Quién la dirige película?
6      - ¿Quién es el director?
7      - ¿Quién dirigió la película?
8      - ¿Quién dirige la película?
9  stories:
10 - story: User request movie director
11   steps:
12   - checkpoint: ask_about_movie
13   - intent: request_movie_director
14   - action: action_movie_director
```

Código 4.9: Datos de entrenamiento. Solicitar director.

Si el usuario pretende que se le de un resumen de la película, informe los actores o el director sin primero haber solicitado una recomendación el modelo de Rasa activará `action_unlikely_intent`. Esta respuesta aparece cuando el modelo detecta e interpreta el intent del mensaje pero este aparece en un momento de la conversación que no esperaba. Esta acción no mostrará ningún mensaje, es un método de control y se puede personalizar. En versiones siguientes se controlarán este tipo de acciones.

Publico una nueva versión para que la prueben usuarios reales.

4.6. Sprint 6

Comencé el sprint 6 como los anteriores, analizando las conversaciones de los usuarios con el chatbot. Detecté un problema, el comienzo de las conversaciones estaba bastante controlado pero a medida que la conversación seguía y se iban entrelazando las distintas peticiones sobre la película el chatbot dejaba de lanzar las acciones correspondientes.

Hacían falta más datos de entrenamiento con las múltiples historias o ejemplos de historias que se pueden dar. Revisando la documentación la recomendación de Rasa es escribir historias utilizando el aprendizaje interactivo, utilizando el comando `rasa interactive`.

Con el comando `rasa interactive` se entrena el modelo y después se abre una sesión para interectuar con el chatbot donde se podrán corregir las predicciones del asistente mientras se habla con él. El aprendizaje interactivo facilita la escritura de historias hablando con el chatbot y proporcionándole feedback [10]. Así puedo ir enseñándole como se debe comportar. Al finalizar la sesión las historias, entities, intents, etc. aprendidos por el bot se exportan a los ficheros de entrenamiento.

```
stories:
- story: User multiple request 1
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_info
  - action: action_summary_movie
  - intent: request_movie_director
  - action: action_movie_director
  - intent: request_movie_cast
  - action: action_movie_cast

- story: User multiple request 2
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_info
  - action: action_summary_movie
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: request_movie_director
  - action: action_movie_director

- story: User multiple request 3
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: request_movie_info
  - action: action_summary_movie
  - intent: request_movie_director
  - action: action_movie_director

- story: User multiple request 4
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: request_movie_director
  - action: action_movie_director

- story: User multiple request 5
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: request_movie_director
```

```
46   - action: action_movie_director
47   - intent: request_movie_info
48   - action: action_summary_movie
```

Código 4.10: Datos de entrenamiento. Historias complementarias.

Con estos ejemplos de historias conseguí que el chatbot fuera capaz de practicamente todos los flujos posibles de la conversación.

Tras mejorar las historias me planteé añadir más funcionalidad, por ejemplo, dar más datos sobre los actores y directores de las películas, y las películas en las que han participado. También poder preguntar al bot directamente por cualquier actor.

- Solicitar información actor/director: El chatbot debe ser capaz de comprender cuando el usuario solicita un información sobre un actor/director de los proporcionados y mostrar un mensaje de texto junto con una imagen del actor/director.
- Solicitar películas actor/director: El chatbot debe ser capaz de comprender cuando el usuario solicita en qué otras películas participa el actor/director que se informa previamente.
- Buscar actor/director: El chatbot debe comprender cuando el usuario quiere buscar un actor/director por su nombre.

4.6.1. Solicitar información actor/director

El chatbot debe ser capaz de comprender cuando el usuario solicita un información sobre un actor/director de los proporcionados y mostrar un mensaje de texto junto con una imagen del actor/director.

Una vez que el usuario ha solicitado información sobre que actores o directores participan en la película, éste puede pedir información sobre uno de los actores o directores. Para conseguir esta funcionalidad voy a utilizar los botones de Telegram, ya que facilitará la interacción del usuario con el chatbot. Mostraré 5 botones con los 5 principales actores/directores que ofrece la API de TMDB. Estos botones mostrarán el nombre del actor y como payload (mensaje que devuelve) el id del actor correspondiente, algo así person_155256. Ejemplo de convesación: Figura 4.8.

Completo las historias anteriores para incluir un paso más en las posibles interacciones de la historia. Completo con el intent `inform_person_id` que se activará cuando el usuario hace click en uno de los botones. Añado la acción personalizada `action_extract_person_id` que se encarga, mediante expresiones regex, de extraer el id del actor/director del texto que envía telegram, por ejemplo, de person_55190 se obtiene 55190 y lo almacena en el slot `person_id` que se utilizará en el endpoint más adelante para obtener información sobre el actor/director.

Creo las acción personalizada `action_inform_person` para llamar al endpoint de la API correspondiente de obtener la información de una actor/director.

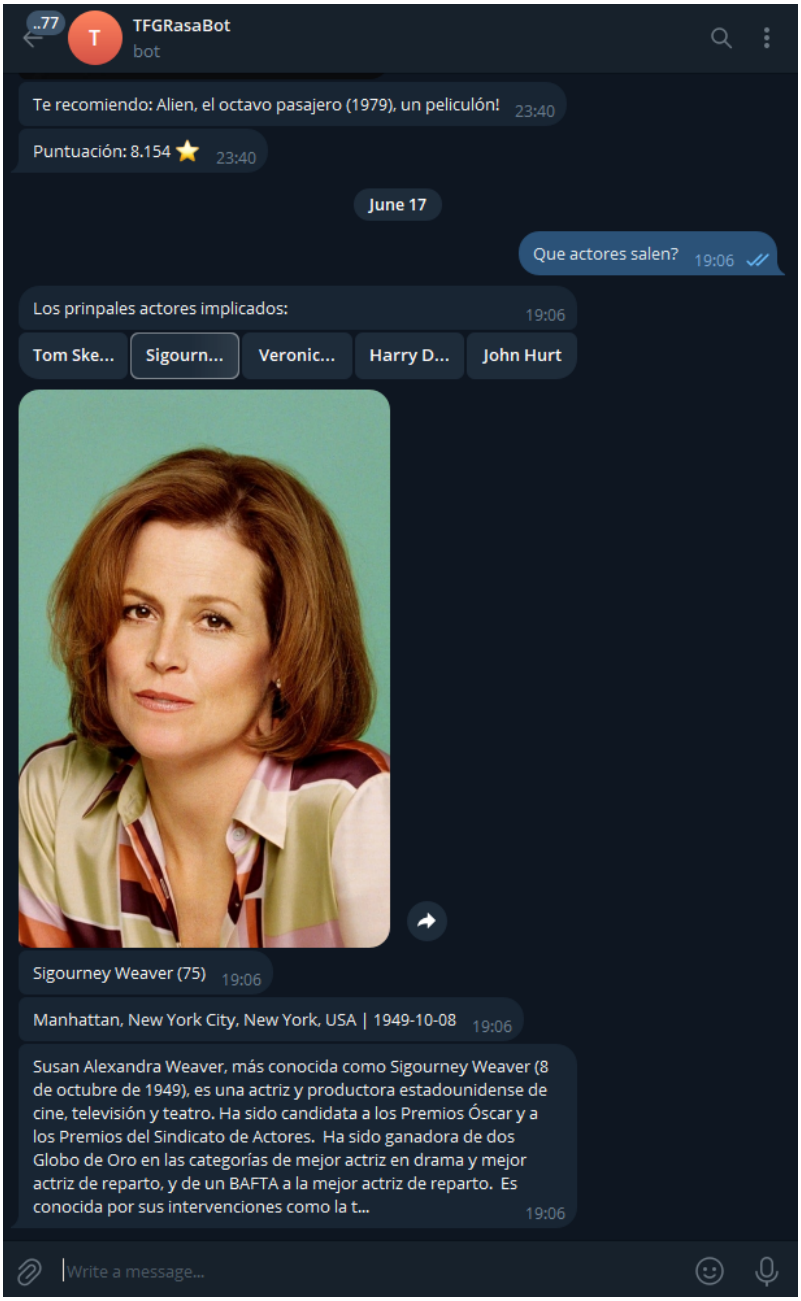

Figura 4.8: Ejemplo de conversación

A parte de modificar las historias para incluir esta nueva interacción creo una regla `Extract person_id` ya que no hace falta ningún contexto previo responder a esta solicitud simplemente extraer el id del botón.

```
nlu:
- intent: inform_person_id
  examples: |
    - person_123
    - person_64151
    - person_55190
    - person_23681
stories:
- story: User multiple request 1
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_info
  - action: action_summary_movie
  - intent: request_movie_recommendation
  - action: action_movie_recommendations
  - intent: request_movie_director
  - action: action_movie_director
  - intent: inform_person_id
  - action: action_extract_person_id
  - action: action_inform_person
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: inform_person_id
  - action: action_extract_person_id
  - action: action_inform_person
rules:
- rule: Extract person_id
  steps:
    - intent: inform_person_id
    - action: action_extract_person_id
    - action: action_inform_person
slots:
  person_id:
    type: text
    mappings:
    - type: custom
      action: action_extract_person_id
```

Código 4.11: Datos de entrenamiento. Información actor/director.

4.6.2. Solicitar películas actor/director

El chatbot debe ser capaz de comprender cuando el usuario solicita en qué otras películas participa el actor/director que se informa previamente.

Al igual que en la anterior funcionalidad al preguntar por las películas en las que ha trabajado o dirigido una persona se le va a mostrar 5 botones con el nombre de las películas y como payload

Figura 4.9: Ejemplo conversación

el id, movie_551662. Primero se muestran las películas en las que ha actuado y luego si ha participado en otras películas como director, productor, etc. También creo una acción personalizada `action_extract_movie_id` de donde se extrae el id de la película empleando expresiones regulares y lo guarda en slot `movie_id`. Ejemplo de conversación: Figura 4.9.

La acción personalizada `action_inform_movie` utiliza el movie_id para extraer información de la película desde el endpoint de la API.

```
nlu:
- intent: request_person_movies
  examples: |
    - ¿En que películas ha trabajado?
    - ¿Qué películas ha hecho?
    - ¿Cuáles son las otras películas en las que ha participado?
```

```
    - Dime películas en las que ha actuado
    - ¿Tiene otras películas?
    - ¿En qué otras películas ha actuado?
    - Muéstrame más películas en las que ha trabajado
    - ¿Cuáles son las otras películas de este actor?
    - ¿Qué otras películas ha dirigido?
    - ¿En qué películas ha trabajado este director?
stories:
- story: User multiple request (with person movies) 1
  steps:
  - checkpoint: ask_about_movie
  - intent: request_movie_cast
  - action: action_movie_cast
  - intent: request_person_movies
  - action: action_inform_person_movies
  - intent: request_movie_info
  - action: action_summary_movie
  - intent: request_movie_recommendation
  - action: action_movie_recommendations
  - intent: request_movie_director
  - action: action_movie_director
  - intent: request_person_movies
  - action: action_inform_person_movies
rules:
- rule: Extract movie_id
  steps:
    - intent: inform_movie_id
    - action: action_extract_movie_id
    - action: action_inform_movie
slots:
  movie_id:
    type: text
    mappings:
    - type: custom
      action: action_extract_movie_id
```

Código 4.12: Datos de entrenamiento. Información películas actor/director.

4.6.3. Buscar actor/director por nombre

Buscando entre la documentación de la API [15] existe un endpoint que busca actores dado un nombre. Necesito un intent que para extraer el nombre del actor que se desea buscar.

Estuve probando a entrenar al chatbot con intents que incluían nombres de actores pero no fue suficiente. El chatbot sólo era capaz de reconocer los nombres que había informado en los datos de entrenamiento, cualquier otro nombre no lo reconocía como entity nombre.

Para superar esta limitación, investigué más en la documentación de Rasa y descubrí que el pipeline del proyecto se puede integrar con spaCy, lo que podría mejorar significativamente la identificación de nombres al utilizar el reconocimiento de entidades con modelos preentrenadas.

Descargue las dependencias y el modelo grande de spaCy en español, y configuré el pipeline con el para incluir el `SpacyEntityExtractor`. Usando las entities PERSON que detecta spaCy conseguí mejorar la identificación de nombres en el intent pero no tanto como esperaba. Esta configuración era capaz de identificar prácticamente todos los nombres pero al combinarlos con el apellido para afinar la búsqueda en la API, no era tan bueno. Además si el usuario quería buscar un actor que no tiene nombre de origen español fallaba ya que estaba usando el modelo español.

Finalmente descarté esta funcionalidad al no ser capaz de hacer funcionar la extracción de nombres del intent como esperaba. Para una futura ampliación del chatbot se podría investigar en profundidad la integración de spaCy y otras librerías para mejorar la extracción de entidades como nombres.

Al finalizar el sprint publico una nueva versión del chatbot.

4.7. Sprint 7

Al inicio del Sprint 7, revisé las interacciones del chatbot almacenadas en MongoDB, como en sprints anteriores. Añadiendo más ejemplos de intents y sinónimos.

Las pruebas con usuarios reales han ido bastante bien y los flujos de las conversaciones diseñadas son claros. El *Happy Path*, el camino ideal que siguen las conversaciones que he diseñado está funcionando bien. Ahora me centraré en controlar esos casos en los que el usuario se desvía del camino descrito en las stories.

Con la funcionalidad básica establecida y la integración de la API completa, doy por concluido la introducción de nuevas features. Me centraré en mejorar como el chatbot maneja mensajes inesperados y en solicitar más información al usuario en el formulario.

4.7.1. Solicitar plataforma streaming

Para afinar las recomendaciones de películas, he decidido añadir una pregunta en el formulario sobre las plataformas de streaming a las que el usuario está suscrito. Esta información permitirá utilizar un filtro en la API que tiene en cuenta las plataformas disponibles para el usuario, mejorando así la relevancia de las películas recomendadas.

Para poder hacer las preguntas he modificado el formulario que tenía para hacerlo dinámico. Con dinámico me refiero a que los slots que el formulario necesita rellenar cambie según las respuestas del usuario. Esto lo consigo modificando los slots del formulario con una acción personalizada desde el fichero `action.py`. 4.10.

En primer lugar, pregunto al usuario si está suscrito a alguna plataforma de streaming, a la cuál debe responder de manera afirmativa o negativa. He creado los intent `affirm` y `deny` para identificar la respuesta del usuario. Si el usuario responde afirmativamente el slot `has_provider`

```python
async def required_slots(
    self,
    domain_slots: List[Text],
    dispatcher: CollectingDispatcher,
    tracker: Tracker,
    domain: DomainDict,
) -> List[Text]:
    required_slots = ["has_provider", "provider", "movie_genre"]

    if tracker.slots.get("has_provider") is False:
        required_slots.remove("provider")

    return required_slots
```

Figura 4.10: Formulario dinámico

guarda el valor True y entonces la siguiente pregunta será a que plataforma está subscrito, aquí deberá responder una entras las válidas por la API. En caso de responder que no tiene ninguna subscripción se guarda en el slot `has_provider` False, y esto provoca que se elimine `provider` de los slots requeridos en el formulario y se elimine la pregunta para saber a que plataforma está subscrito.

Para validar las respuestas a la pregunta a sobre la plataforma a la que está subscrito he utilizado la misma estrategia que con los géneros. He buscado que plataformas utiliza la API y he creado una lista de las más conocidas en una tabla lookup. También he creado algunos sinónimos para identificar plataformas más fácilmente.

```
nlu:
- intent: affirm
  examples: |
    - Si
    - Vale
    - Perfecto
    - Correcto
- intent: deny
  examples: |
    - No
    - Negativo
    - Para nada
    - En absoluto
- synonym: Amazon Prime Video
  examples: |
    - Amazon
    - Amazon Prime
    - Prime Video
    - Prime
- synonym: Disney Plus
  examples: |
    - Disney
    - Disney+
- lookup: provider_type
  examples: |
    - Amazon Prime Video
```

```
    - Netflix
    - Apple TV
    - Disney Plus
    - HBO Max
    - Movistar Plus
    - Atres Player
    - Youtube
- intent: inform_provider
  examples: |
    - Estoy suscrito a [Amazon Prime Video](provider_type)
    - Tengo [Amazon Prime Video](provider_type)
    - Uso [Amazon Prime Video](provider_type)
    - Estoy subscrito a [Netflix](provider_type)
    - Tengo [Netflix](provider_type)
    - Uso [Netflix](provider_type)
forms:
  movie_form:
    required_slots: []
slots:
  user_provider:
    type: any
    mappings:
    - type: custom
  has_provider:
    type: bool
    mappings:
    - type: from_intent
      value: true
      intent: affirm
      conditions:
        - active_loop: movie_form
    - type: from_intent
      value: false
      intent: deny
      conditions:
        - active_loop: movie_form
  provider:
    type: text
    mappings:
    - type: from_entity
      entity: provider_type
      conditions:
      - active_loop: movie_form
responses:
  utter_ask_provider:
    - text: ¿A qué plataforma estás subscrito?
```

Código 4.13: Datos de entrenamiento. Plataformas streaming.

4.7.2. Unhappy Paths

Las Unhappy Paths son todos los casos extremos posibles. Por ejemplo, el usuario se niega a dar la información solicitada, cambia el tema de conversación o corrige algo que ha dicho antes. [10].

En primer lugar voy a controlar los mensajes fuera de lugar, estos son aquellos mensajes que nada tienen que ver con el dominio de mi chatbot, es decir, todos aquellos mensajes que se alejen de lo controlamos en las historias y no existe un intent para identificarlo. Para controlarlo he creado un primer intent general llamado `out_of_scope` y otro más particular para preguntas relacionadas con el clima `out_of_scope_weather`. He creado este segundo grupo de intents ya que muchos usuarios hacían referencia al clima y me pareció buena idea tratarlo por separado. Los ejemplos de estos dos intent se han extraído de las conversaciones con usuarios reales.

Para responder a estos mensajes he creado un par de reglas. La utilización de reglas en este caso es adecuada ya que quiero que siempre se le responda a los mensajes out_of_scope sin importar el contexto de la conversación.

```
nlu:
- intent: out_of_scope_weather
  examples: |
    - ¿Cómo está el clima hoy?
    - ¿Va a llover mañana?
    - ¿Cuál es la temperatura actual?
    - Dime el pronóstico del tiempo
    - ¿Qué tiempo hace?
    - ¿Está soleado?
    - ¿Habrá tormenta?
- intent: out_of_scope
  examples: |
    - ¿Te gustan los perros?
    - Tengo una mascota
    - Me gustan los gatos
    - Tengo un loro
    - Dime un chiste
    - Quiero encargar comida
    - Dime el precio de bitcoin
rules:
- rule: Out of scope
  steps:
    - intent: out_of_scope
    - action: utter_out_of_scope
- rule: Out of scope
  steps:
    - intent: out_of_scope_weather
    - action: utter_out_of_scope_weather
responses:
  utter_out_of_scope:
    - text: Mi propósito es recomendas películas, no tengo información sobre
        esto.
```

```
32  utter_out_of_scope_weather:
33    - text: No tengo disponible el pronóstico del tiempo.
```

Código 4.14: Datos de entrenamiento. Out of scope.

Otro unhappy path a controlar son los mensajes con baja confianza NLU, esto sucede cuando un usuario envía un mensaje ambiguo y el modelo Rasa no es capaz de asegurar el intent con cierta precisión. Cambio la configuración para aumentar el umbral de confianza que quiero que use el chatbot, establezco el umbral en 0.5. Todos los intents que no se alcancen este umbral se va a predecir el intent `nlu_fallback`. Podemos controlar que sucede que pasa cuando se lanza este intent con una regla y mostrar un mensaje del tipo "Lo siento, no te entiendo.".

```
1  pipeline:
2  - name: FallbackClassifier
3      threshold: 0.5
4  rules:
5  - rule: Message with low NLU confidence
6    steps:
7    - intent: nlu_fallback
8    - action: utter_please_rephrase
9  responses:
10   utter_please_rephrase:
11     - text: Lo siento, no te entiendo.
```

Código 4.15: Datos de entrenamiento. NLU fallback.

Por último, voy a mejorar el flujo de conversaciones ya que he notado que en conversaciones con usuarios muy largas el chatbot "se pierde". Con esto merefiero, que en ocasiones el chatbot no sabe que acción aplicar ya que no tiene mucho contexto de en que momento de la conversación está. Para mejorarlo creo una acción personalizadad para controlar la acción que viene por defecto, `action_unlikely_intent`. Esta acción se lanza cuando el chatbot ha detectado un intent correctamente pero no lo esperaba en ese punto de la conversación.

En la acción personaliza puedo consultar cuál fue último intent identificado por el chatbot, y ha provocado el `action_unlikely_intent` y responder en cosecuencia. Si el intent es sobre solicitar información de las películas y en el slot `user_movies_history` contiene datos indica estamos en un estado avanzado de la conversación y no hay problema en lanzar la acción correspondiente al intent. Si por otro lado el slot `user_movies_history` se encuentra vacío indica que el usuario ha empezado la conversación pidiendo un resumen, por ejemplo, de una película que todavía no se le ha recomendado. Para solucionarlo se activa el formulario para recomendarle primero la película y después continuar con las preguntas relacionadas con ella, siguiendo el Happy Path definido. Lo mismo aplico con las preguntas relacionadas con actores/directores, si el slot `user_movies_history` o el slot `person_id` están vacíos se les debe recomendar una película primero.

```python
class ActionUnlikelyIntent(Action):

    def name(self) -> Text:
        return "action_unlikely_intent"

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[EventType]:

        last_intent = tracker.get_intent_of_latest_message()
        user_movies_history: List[int] = tracker.get_slot("user_movies_history")
        person_id: int = tracker.get_slot("person_id")

        if last_intent in [
            "request_movie_recommendation",
            "request_movie_info",
        ]:
            if user_movies_history is None:
                dispatcher.utter_message(
                    text="Parece que has solicitado información sobre una película antes de que te haya recomendado alguna."
                )
                dispatcher.utter_message(text="Primero necesito saber tus gustos.")
                return [
                    ActiveLoop("movie_form"),
                ]

            if last_intent == "request_movie_recommendation":
                return [FollowupAction("action_movie_recommendation")]
            if last_intent == "request_movie_info":
                return [FollowupAction("action_summary_movie")]
```

Figura 4.11: Acción ActionUnlikelyIntent

Con estos Unhappy Paths controlados el chatbot se presenta bastante robusto contra mensajes inesperados.

Por último añado algunos intents y rules que mejoran la experiencia del usuario al dialogar con el chatbot pero que no suponen un cambio en las funcionalidades. El chatbot responderá cuando el usuario indique que le ha gustado la recomendación o cuando exprese que no le ha gustado. Además también interpretará cuando el usuario le da las gracias.

```
nlu:
- intent: dislike_movie
  examples: |
    - No me gusta la película
    - No es lo que esperaba
    - No me gusta
    - Tampoco me gusta
    - Es mala
    - Es muy mala
- intent: like_movie
  examples: |
    - Me gusta
    - Sí, me gusta la película
    - La veré
    - Tiene buena pinta
    - Es interesante
    - Está bien
    - Me encanta
```

```
- intent: thanks
  examples: |
    - Gracias
    - Muchas gracias
    - Te lo agradezco
    - Gracias por tu ayuda
    - Agradezco tu asistencia
    - Gracias, eso es todo
    - Gracias por la información
    - Muy amable
rules:
- rule: User dislike movie
  steps:
  - intent: dislike_movie
  - action: utter_sad
- rule: User like movie
  steps:
  - intent: like_movie
  - action: utter_happy
- rule: User thanks
  steps:
  - intent: thanks
  - action: utter_thanks
responses:
 utter_sad:
  - text: Lamento que no te haya gustado.
  utter_happy:
  - text: Me alegro que te guste
  utter_thanks:
    - text: De nada, encantado de ayudar!
```

Código 4.16: Datos de entrenamiento. Mejoras de diálogo.

4.7.3. Cambiar género y plataforma

Durante el desarrollo me di cuenta que no había forma de cambiar el género que el usuario había escogido al principio de la conversación. Esto suponía un problema ya que no se le iba a poder recomendar películas de otros géneros hasta que no expirase la sesión y el slot con el género quedase vacío.

Cree los intents correspondientes para esta petición de cambio y cree una acción que borra el contenido del slot `movie_genre` y lanza el formulario de nuevo para que el usuario responda la pregunta sobre el género para aprovechar la validación del dato introducido que tengo en ese formulario. Figura: 4.12.

```
nlu:
- intent: change_genre
  examples: |
    - Quiero cambiar de género
    - Cambiar de género
```

```python
class ActionInformGenre(Action):
    def name(self) -> Text:
        return "action_change_genre"

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[EventType]:

        return [SlotSet("movie_genre", None), ActiveLoop("movie_form")]
```

Figura 4.12: Acción cambiar de género

```
    - Me gustaría cambiar el género
    - Quisiera cambiar el género
    - Cambia el género
    - Quiero otro género
rules:
- rule: Change Genre
  steps:
    - intent: change_genre
    - action: action_change_genre
    - action: movie_form
    - active_loop: movie_form
```

Código 4.17: Datos de entrenamiento. Cambiar de género.

También me di cuenta que tendría el mismo problema con la plataforma de streaming a la que el usuario había contestado que estaba subcrito. El usuario no podía buscar películas en otras plataformas o eliminar la que tenía si ya no estaba subscrito. Incluí los intens, rules y acciones necesarias para eliminar los slots.

```
nlu:
- intent: inform_no_provider
  examples: |
    - No uso ninguna plataforma
    - No estoy suscrito
    - Ya no tengo la suscripción
    - Se terminó la suscripción
    - No tengo acceso a ningún servicio de streaming
    - No tengo suscripciones activas
    - No estoy suscrito a ningún servicio
- intent: inform_change_provider
  examples: |
    - Tengo nueva suscripción
    - He cambiado de plataforma
    - Ahora estoy suscrito a otra plataforma
    - He cambiado mi suscripción
    - He actualizado mi suscripción
    - Me he suscrito a un nuevo servicio
rules:
```

```
- rule: Inform no provider
  condition:
    - active_loop: null
  steps:
  - intent: inform_no_provider
  - action: action_no_provider
- rule: Change provider
  condition:
    - active_loop: null
  steps:
  - intent: inform_change_provider
  - action: action_change_provider
```

Código 4.18: Datos de entrenamiento. Cambiar de plataforma de streaming.

Finalizando estas mejoras vuelvo a publicar una nueva versión del chatbot.

4.8. Sprint 8

En este último sprint reviso de nuevo las conversaciones y añado algunos ejemplos y sinónimos a los datos de entrenamiento.

Se da por finalizado el desarrollo del chatbot.

Durante las próximas semanas se elaborara la memoria del Trabajo de Fin de Grado.

Capítulo 5

Estado final

A lo largo de este capítulo se detalla el trabajo de análisis y pruebas con usuarios realizado.

5.1. Análisis

5.1.1. Actores

Actores que intervienen en el proyecto desarrollado:

- **Usuario**: Es la persona que accede a Telegram e inicia una conversación con el chatbot. Este actor interactúa directamente con el chatbot, solicitando información y recibiendo respuestas.

- **API TMDB**: Es un sistema externo que proporciona una fuente de datos para el chatbot. Ofrece endpoints para realizar consultas específicas sobre datos de películas que el chatbot utiliza para responder al usuario.

5.1.2. Casos de Uso

A partir de las reuniones con el tutor sobre la funcionalidad del chatbot y mi investigación sobre las posibilidades del framework Rasa y la API de TMDB, he identificado los siguientes casos de uso:

CU-001	**Solicitar recomendación**	
Versión	1.0	
Precondición	El *usuario* ha iniciado una conversación con el chatbot	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el usuario solicite una recomendación de película.	
Secuencia	Paso	Acción
	1	El *usuario* escribe al chatbot con la intención de que le recomiende una película
	2	El sistema procesa el mensaje e identifica la intención.
	2.1	Si el sistema ya cuenta con todos los datos necesarios para realizar la recomendación se salta al paso 9
	3	El sistema pregunta al usuario si está suscrito a alguna plataforma de streaming.
	4	El *usuario* responde.
	5	El sistema valida la respuesta.
	5.1	Si la respuesta es afirmativa el sistema pregunta a cuál está suscrito.
	5.2	El *usuario* responde.
	5.3	El sistema valida la respuesta.
	6	El sistema pregunta que género de películas le interesa.
	7	El *usuario* responde.
	8	El sistema valida la respuesta.
	9	El sistema consulta la *API TMDB* aplicando los filtros con las respuestas del *usuario*.
	10	La *API TMDB* responde con una lista de películas que se pueden recomendar.
	11	El sistema ejecuta el caso de uso descrito en la tabla 5.4 para reducir el número de películas.
	12	El sistema envía un mensaje al *usuario* con la película recomendada.
	13	El sistema ejecuta el caso de uso descrito en la tabla 5.3.
Postcondición	Se le muestra al usuario un mensaje con el título completo de la película junto con el año de estreno, su portada y puntuación..	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
	5, 5.3, 8	El sistema envía un mensaje indicando que no ha entendido la respuesta del *usuario*.
	5.3	El sistema no encuentra la plataforma de streaming dada en la respuesta en dentro de la *API TMDB*.
	8	El sistema no encuentra el género de película dado en la respuesta en dentro de la *API TMDB*.
Comentarios		

Tabla 5.1: CU-001

CU-002	**Solicitar resumen**	
Versión	1.0	
Precondición	El *usuario* ha solicitado una recomendación o información de una película en el transcurso de la conversación	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite un resumen de película.	
Secuencia	Paso	Acción
	1	El *usuario* escribe al chatbot con la intención de obtener un resumen.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema ejecuta el caso de uso descrito en la tabla 5.5.
	4	El sistema utiliza el id de la película para solicitar más detalles a la *API TMDB*.
	5	*API TMDB* responde con información adicional sobre la película.
	6	El sistema envía un mensaje con la sinopsis de la película.
Postcondición	Se le muestra al usuario un mensaje con el resumen de la película.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
	4	No hay ninguna película guardada en el historial. El sistema dispara el caso de uso descrito en la tabla 5.1
Comentarios		

Tabla 5.2: CU-002

CU-003	**Guardar película**	
Versión	1.0	
Precondición	Se ha establecido una estructura para almacenar las películas que se recomiendan o sobre las que se proporciona información durante la conversación.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el sistema desee guardar el id de película.	
Secuencia	Paso	Acción
	1	El sistema obtiene información de una película que ha solicitado el *usuario* a través de la *API TMDB*.
	2	El sistema almacena el id de la película en la estructura diseñada para almacenar el historial de películas.
Postcondición	La estructura para almacenar las películas aumenta su tamaño en una unidad.	
Excepciones	Paso	Acción
Comentarios		

Tabla 5.3: CU-003

CU-004	**Consultar historial de películas**	
Versión	1.0	
Precondición	Se ha establecido una estructura para almacenar las películas que se recomiendan o sobre las que se proporciona información durante la conversación.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el sistema desee consultar si el id de una película existe en el historial de la conversación.	
Secuencia	Paso	Acción
	1	El sistema obtiene información de una lista de películas que encajan con las preferencias del *usuario* a través de la *API TMDB*.
	2	El sistema elimina de la lista las películas que se encuentren en el historial.
Postcondición	Se reduce el número de películas posibles para recomendar.	
Excepciones	Paso	Acción
Comentarios	Con este comportamiento evitamos recomendar al *usuario* siempre las mismas películas	

Tabla 5.4: CU-004

CU-005	**Extraer última película**	
Versión	1.0	
Precondición	Se ha establecido una estructura para almacenar las películas que se recomiendan o sobre las que se proporciona información durante la conversación.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el sistema desee extraer la última película almacenada en el historial de películas.	
Secuencia	Paso	Acción
	1	El sistema obtiene el historial de películas de la conversación.
	2	El sistema extrae el id de la última película guardada en el historial.
Postcondición	Se obtiene el id de la última película almacenada.	
Excepciones	Paso	Acción
	2	El sistema todavía no cuenta con ninguna película en el historial. El sistema disparará el caso de uso descrito en la tabla 5.1.
Comentarios		

Tabla 5.5: CU-005

CU-006	**Solicitar actores**	
Versión	1.0	
Precondición	El *usuario* ha solicitado una recomendación o información de una película en el transcurso de la conversación	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite conocer los actores que salen.	
Secuencia	Paso	Acción
	1	El *usuario* escribe al chatbot con la intención de obtener que actores aparecen en la película.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema ejecuta el caso de uso descrito en la tabla 5.4.
	4	El sistema utiliza el id de la película para solicitar más detalles sobre los actores a la *API TMDB*.
	5	*API TMDB* responde con información adicional sobre los actores de la película.
	6	El sistema envía en un mensaje un listado con los actores de la película.
Postcondición	Se le muestra al usuario un listado con los nombres de los actores principales.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
	4	No hay ninguna película guardada en el historial. El sistema dispara el caso de uso descrito en la tabla 5.1
Comentarios	No hace falta mostrar todos los actores, un listado corto valdría.	

Tabla 5.6: CU-006

CU-007	**Solicitar directores**	
Versión	1.0	
Precondición	El *usuario* ha solicitado una recomendación o información de una película en el transcurso de la conversación	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite conocer los directores de la película.	
Secuencia	Paso	Acción
	1	El *usuario* escribe al chatbot con la intención de obtener que directores dirigieron la película.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema ejecuta el caso de uso descrito en la tabla 5.4.
	4	El sistema utiliza el id de la película para solicitar más detalles sobre los directores a la *API TMDB*.
	5	*API TMDB* responde con información adicional sobre los directores de la película.
	6	El sistema envía en un mensaje un listado con los directores de la película.
Postcondición	Se le muestra al usuario un listado con los directores.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
	4	No hay ninguna película guardada en el historial. El sistema dispara el caso de uso descrito en la tabla 5.1
Comentarios		

Tabla 5.7: CU-007

CU-008	**Solicitar información actor/director**	
Versión	1.0	
Precondición	El *usuario* ha solicitado conocer que actores/directores participan en una película.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite conocer más detalles de un actor/director.	
Secuencia	Paso	Acción
	1	El *usuario* selecciona un actor/director del listado que se le proporciona.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema utiliza el id del actor/director para solicitar más detalles a la *API TMDB*.
	4	*API TMDB* responde con información adicional sobre el actor/director de la película.
	5	El sistema envía en un mensaje con detalles del actor/director.
Postcondición	Se le muestra al usuario un mensaje con detalles del actor/director: nombre completo, fecha y lugar de nacimiento y una pequeña bibliografía.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
Comentarios		

Tabla 5.8: CU-008

CU-009	**Solicitar películas actor/director**	
Versión	1.0	
Precondición	El *usuario* ha solicitado conocer detalles de un actor/director.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite conocer en que otras películas aparece un actor/director.	
Secuencia	Paso	Acción
	1	El *usuario* envía un mensaje con la intención de conocer otras películas en las que el actor/director ha actuado.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema utiliza el id del último actor/director mencionado para solicitar las películas en las que trabaja a la *API TMDB*.
	4	*API TMDB* responde con información adicional sobre las películas del actor/director.
	5	El sistema envía en un mensaje con el listado de películas del actor/director.
Postcondición	Se le muestra al usuario un mensaje con el listado de las películas en las que ha trabajado el actor/director.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
Comentarios	No hace falta mostrar todas las películas, un listado corto con 5 valdría.	

Tabla 5.9: CU-009

CU-010	**Solicitar película similar**	
Versión	1.0	
Precondición	El *usuario* ha solicitado una recomendación o información de una película en el transcurso de la conversación	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el usuario solicite una película similar a la última película que se ha mencinado en la conversación.	
Secuencia	Paso	Acción
	1	El *usuario* escribe al chatbot con la intención de que le informe de una película similar a la última mencionada.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema ejecuta el caso de uso descrito en la tabla 5.5.
	4	El sistema utiliza el id de la película para solicitar películas similares a la *API TMDB*.
	5	*API TMDB* responde con un listado de películas similares.
	6	El sistema ejecuta el caso de uso descrito en la tabla 5.4 para reducir el número de películas.
	7	El sistema envía un mensaje al *usuario* con la primera película similar de la lista.
	8	El sistema ejecuta el caso de uso descrito en la tabla 5.3.
Postcondición	Se le muestra al usuario un mensaje con el título completo de la película junto con el año de estreno, su portada y puntuación.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
Comentarios		

Tabla 5.10: CU-010

Figura 5.1: Diagrama Casos de Uso

CU-011	**Solicitar información película**	
Versión	1.0	
Precondición	El *usuario* ha solicitado conocer las películas en las que ha trabajado un actor/director.	
Descripción	El sistema deberá comportarse como se describe en el siguiente caso de uso cuando el *usuario* solicite conocer más detalles sobre una película.	
Secuencia	Paso	Acción
	1	El *usuario* envía un mensaje con la intención de conocer más información sobre una de las películas en las que ha trabajado un actor/director.
	2	El sistema procesa el mensaje e identifica la intención.
	3	El sistema utiliza el id de la película seleccionada para extraer información de la *API TMDB*.
	4	*API TMDB* responde con información adicional sobre la película.
	5	El sistema envía en un mensaje información sobre la película.
Postcondición	Se le muestra al usuario un mensaje con el título completo de la película junto con el año de estreno, su portada y puntuación.	
Excepciones	Paso	Acción
	2	El sistema no detecta la intención del *usuario* y le envía un mensaje indicando que pruebe de nuevo.
Comentarios		

Tabla 5.11: CU-011

5.1.3. Requisitos funcionales

RF-001	**Solicitar recomendación**
Dependencias	Ninguna
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita que se le recomiende una película.
Importancia	Alta
Comentarios	Si los usuarios utilizan nuevas expresiones para solicitar recomendaciones de películas, estas deberían integrarse en los datos de entrenamiento para mejorar la capacidad de respuesta del chatbot.

Tabla 5.12: RF-001

RF-002	**Mostrar formulario**
Dependencias	Ninguna
Descripción	El sistema deberá realizar una serie de preguntas al usuario para conocer sus preferencias cinematográficas y las plataformas de streaming que utiliza. Datos obligatorios que se deben obtener: ▪ Confirmación si el usuario está suscrito a alguna plataforma y, en caso afirmativo, cuál. ▪ El género de películas que prefiere ver.
Importancia	Alta
Comentarios	Es necesario que el usuario responda a todas antes de continuar en la conversación.

Tabla 5.13: RF-002

RF-003	**Saludar**
Dependencias	Ninguna
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario le envía un saludo.
Importancia	Alta
Comentarios	El saludo marcará el inicio de la mayoría de las historias.

Tabla 5.14: RF-003

RF-004	**Despedirse**
Dependencias	Ninguna
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario se despide.
Importancia	Baja
Comentarios	

Tabla 5.15: RF-004

RF-005	**Solicitar resumen de película**
Dependencias	5.12
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario le solicita un resumen de la película.
Importancia	Media
Comentarios	Se tomará el id de la última película guardada en el historial.

Tabla 5.16: RF-005

RF-006	**Solicitar actores**
Dependencias	5.12
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita conocer que actores participan en la película. En la respuesta se mostrará un listado de los principales actores. Se limitará el listado a 5 actores.
Importancia	Media
Comentarios	Se tomará el id de la última película guardada en el historial.

Tabla 5.17: RF-006

RF-007	**Solicitar directores**
Dependencias	5.12
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita conocer que directores dirigen la película. En la respuesta se mostrará un listado de los directores.
Importancia	Media
Comentarios	Se tomará el id de la última película guardada en el historial.

Tabla 5.18: RF-007

RF-008	**Solicitar información actor/director**
Dependencias	5.17, 5.18
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita información sobre el actor/director que el usuario indique.
Importancia	Media
Comentarios	

Tabla 5.19: RF-008

RF-009	**Solicitar películas actor/director**
Dependencias	5.12
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita información sobre otras películas en las que aparece el actor/director que se ha comentado recientemente. Se mostrará un listado limitado a 5 películas.
Importancia	Media
Comentarios	Se utiliza el id del último actor/director del que se ha hablado.

Tabla 5.20: RF-009

RF-010	**Solicitar películas similares**
Dependencias	5.12
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario solicita que se le muestre una película similar a la recomendada.
Importancia	Baja
Comentarios	

Tabla 5.21: RF-010

RF-011	**Mostrar información película**
Dependencias	Ninguna
Descripción	El sistema deberá mostrar la siguiente información de las películas: ▪ Título completo ▪ Año de estreno ▪ Puntuación ▪ Imagen con la portada
Importancia	Baja
Comentarios	

Tabla 5.22: RF-011

RF-011	**Solicitar información película**
Dependencias	**??**
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario información sobre una película
Importancia	Baja
Comentarios	

Tabla 5.23: RF-011

RF-012	**Mostrar información actor/director**
Dependencias	Ninguna
Descripción	El sistema deberá mostrar la siguiente información sobre los actore/directores: ▪ Nombre completo ▪ Fecha y lugar de nacimiento ▪ Pequeña biografía
Importancia	Baja
Comentarios	

Tabla 5.24: RF-012

RF-013	**Usuario indica que le gusta la película**
Dependencias	Ninguna
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario indica que le gusta una película.
Importancia	Baja
Comentarios	

Tabla 5.25: RF-013

RF-014	**Usuario indica que no le gusta la película**
Dependencias	Ninguna
Descripción	El sistema deberá ser capaz de reconocer y responder cuando el usuario indica que no le gusta una película.
Importancia	Baja
Comentarios	

Tabla 5.26: RF-014

5.1.4. Requisitos no funcionales

RNF-001	**Utilizar el framework Rasa**
Dependencias	Ninguna
Descripción	El sistema deberá el framework Rasa para construir el chatbot.
Importancia	Alta
Comentarios	

Tabla 5.27: RNF-001

5.1.5. Requisitos de almacenamiento información

RI-001	**Guardar película**
Dependencias	Ninguna
Descripción	El sistema deberá guardar un historial con los ids de las películas recomendadas o mencionadas durante la conversación.
Importancia	Alta
Comentarios	Este historial servirá para no repetir recomendaciones al usuario y para poderle ofrecer información sobre la última película comentada.

Tabla 5.28: RI-001

RI-002	**Guardar actor/director**
Dependencias	Ninguna
Descripción	El sistema deberá guardar el id del último actor/director mencionado durante la conversación.
Importancia	Alta
Comentarios	

Tabla 5.29: RI-002

RI-003	**Guardar género**
Dependencias	Ninguna
Descripción	El sistema deberá guardar el id del género cinematográfico escogido por el usuario.
Importancia	Alta
Comentarios	

Tabla 5.30: RI-003

RI-004	Guardar plataforma streaming
Dependencias	Ninguna
Descripción	El sistema deberá guardar el id de la plataforma de streaming a la que este suscrito el usuario.
Importancia	Alta
Comentarios	

Tabla 5.31: RI-004

5.2. Pruebas con usuarios

La mayor parte de las pruebas, los usuarios interactuaron con el chatbot de manera telemática y sin mi supervisión directa, eligiendo el momento que más le convenía. Sin embargo, si que pude realizar test de usabilidad con amigos y familiares cercanos lo que me permitió observar la interacción directa con los prototipos del chatbot e identificar posibles problemas. A continuación describo los test realizados.

5.2.1. Tests de usabilidad

Durante el test yo hacía los roles de facilitador, guiando a los usuarios en las tareas les proponía, y de observador, prestando atención a el comportamiento del usuario y evaluando el prototipo.

Este primer test (tabla 5.32) lo realizo con dos usuarios distintos al final del sprint 3, la primera versión del prototipo.

Resultados: Ambos usuarios han sabido iniciar una conversación con el chatbot sin problemas y le han podido saludar y despedirse sin dificultad. El mensaje inicial donde el chatbot describe su funcionalidad no ha quedado demasiado claro para el usuario 2. Ambos usuarios no han podido obtener una recomendación de una película debido a 3 razones: utilizaban géneros de películas que no estaban en los datos de entrenamiento, utilizaban expresiones diferentes de los ejemplos en los datos de entrenamiento y respondían con un mensaje que no tenían que ver con la pregunta.

Debo mejorar la forma en la que el usuario introduce la información para ello aplicaré los siguientes medidas:

- Añadir más ejemplos con expresiones sobre como informar el género. Tomaré ejemplos de las trazas de las conversaciones guardadas.
- Ampliar los géneros de películas que el usuario puede responder, para ello utilizaré una tabla `lookup`.
- Usar la estructura formulario para obligar al usuario a responder correctamente.

ID	Diálogo	Observaciones	Puntuación	
			U1	U2
1	Iniciar	El usuario localiza al chatbot dentro de Telegram e inicia una conversación	5	5
		Métrica: Si tarda hasta 15 segundos en iniciar la conversación		
2	Saludar	El usuario envía un saludo al chatbot	5	5
		Métrica: Si introduce alguno de los saludos disponibles en los datos de entrenamiento		
3	Saludar	El usuario comprende la utilidad del chatbot	5	3
		Métrica: Si hace comentarios o necesita aclaración por parte del facilitador		
4	Solicitar recomendación	El usuario le solicita al chatbot una recomendación de película	4	4
		Métrica: Si tarda hasta 10 segundos en encontrar una expresión que el chatbot entienda como intención de solicitar una recomendación		
5	Solicitar recomendación	El usuario comprende la pregunta sobre el género de la película	5	5
		Métrica: Si hace comentarios o necesita aclaración por parte del facilitador		
6	Solicitar recomendación	El usuario es capaz de enviar un género de películas válido	1	1
		Métrica: Si tarda hasta 10 segundos en encontrar un género disponible en los datos de entrenamiento		
7	Despedirse	El usuario se despide del chatbot	5	5
		Métrica: Si introduce alguna de las formas de despedirse disponibles en los datos de entrenamiento		

Tabla 5.32: Test usabilidad

El segundo test lo realizo en una fase más avanzada del prototipo y con más funcionalidades, al final del sprint 7 (tabla 5.33) .

Resultados: Se han mejorado las deficiencias respecto al anterior test, el formulario funciona perfectamente y los ejemplos de géneros y plataformas de streaming cubren todas las necesidades de los usuarios. Las respuestas del chatbot parecen claras y maneja las entradas inesperadas bastante bien, pero necesita alguna mejora. La satisfacción de las películas no es muy alta, tal vez habría que filtrar mejor la información proporcionada por la API TMDB.

Algo que he notado que no estaba como prueba en los test era que en conversaciones muy largas el chatbot perdía el rumbo de la conversación, y fallaba en la detección de algunos intents.

ID	Diálogo	Observaciones	Puntuación	
			U1	U2
1	Iniciar	El usuario localiza al chatbot dentro de Telegram e inicia una conversación	5	5
		Métrica: Si tarda hasta 15 segundos en iniciar la conversación		
2	Saludar	El usuario envía un saludo al chatbot	5	5
		Métrica: Si introduce alguno de los saludos disponibles en los datos de entrenamiento		
3	Saludar	El usuario comprende la utilidad del chatbot	5	5
		Métrica: Si hace comentarios o necesita aclaración por parte del facilitador		
4	Solicitar recomendación	El usuario le solicita al chatbot una recomendación de película	5	5
		Métrica: Si tarda hasta 10 segundos en encontrar una expresión que el chatbot entienda como intención de solicitar una recomendación		
5	Solicitar recomendación	El usuario responde correctamente a las preguntas del formulario	5	5
		Métrica: Si tarda hasta 10 segundos en responder a las preguntas del formulario		
6	Solicitar actores	El usuario formula correctamente la petición para conocer los actores de la película	4	4
		Métrica: Si continua la conversación sin complicaciones		
7	Solicitar recomendación	El usuario está conforme con la recomendación basada en sus preferencias	3	4
		Métrica: Satisfacción del usuario		
8	Solicitar recomendación	El usuario se equivoca en una de las preguntas pero se lo vuelven a preguntar y continua	5	5
		Métrica: Si tarda hasta 10 segundos en responder a las preguntas del formulario		
9	Solicitar resumen	El usuario utiliza expresiones correctas para solicitar un resumen	5	3
		Métrica: Si no hace preguntas sobre que más hablar con el chatbot		
10	Out of context	Observa al usuario introducir mensajes que el chatbot no se espera	4	4
		Métrica: Si continua la conversación sin complicaciones		
11	Claridad de respuestas	El usuario entiende como respuestas claras los mensajes del chatbot	5	5
		Métrica: Clasificación de la claridad		

Tabla 5.33: Test usabilidad 2

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

El desarrollo del chatbot con Rasa alcanzó los objetivos propuestos inicialmente, ofreciendo una solución funcional para mejorar la interacción usuario-máquina en un dominio concreto, en este caso relacionado con el cine. Rasa a demostrado ser un framework eficaz para crear chatbots que entienden y responden a las necesidades de los usuarios, aplicando NLU, demostrando su capacidad para manejar interacciones/conversaciones complejas de manera eficiente. También destacar todas las herramientas que Rasa ofrece para facilitar el desarrollo y la integración con otras herramientas ya reconocidas y probadas.

A lo largo del desarrollo me enfrenté a varios desafíos técnicos. Debido a la falta de experiencia con los conceptos de NLU y los chatbots tuve que afrontar una curva de aprendizaje para entender y sacar partido de estos conceptos más adelante. Este desafío me ha servido para mejorar mis habilidades de programación y el diseño de interacciones usuario-máquina.

El feedback de los usuarios durante todas las fases del proyecto ha sido fundamental para optimizar el desarrollo del chatbot, destacando la importancia de mejorar los datos de entrenamiento y ajustar las funcionalidades para mejorar la precisión del chatbot.

En general, la experiencia de desarrollar este chatbot con Rasa ha sido enormemente satisfactoria y enriquecedora, ofreciéndome una fuerte introducción a nuevos campos en tecnología e inteligencia artificial. También he tenido la oportunidad de aplicar todos los conocimientos adquiridos en las diferentes asignatura de la carrera y aplicarlos a un proyecto real.

6.2. Trabajo futuro

Paso a resumir algunos puntos en los que se puede profundizar para mejorar el proyecto:

- **Explorar modelos pre-entrenados:** Se podría seguir investigando la integración de mo-

delos pre-entrenados de terceros como los ofrecidos por spaCy o Duckling para mejorar la identificación de entidades como nombres, lugares, tiempo, o cualquier nueva funcionalidad que se aproveche de estos modelos. También estaría bien explorar la posibilidad de crear modelos propios que identifiquen entidades específicas del proyecto como géneros de cine y las plataformas de streaming. Crear modelos personalizados ofrece mayor modularidad al proyecto, y estos podrían ser adaptados para su uso en otros chatbots.

- **Testing:** Otro aspecto muy interesante a incorporar es el testing para garantizar la funcionalidad del chatbot. Rasa ofrece herramientas para ejecutar pruebas sobre como debería responder el chatbot a diferentes mensajes del usuario. Además en después de cada test ofrece un análisis de evaluando el modelo NLU. En el fichero resultado de los test `intent_confusion_matrix.png` muestra que intentciones se confunden con otras. Todas las muestras que se hayan predicho incorrectamente se guardan en `erros.json`.

- **Funcionalidad:** Ampliar la funcionalidad del chatbot. Utilizando las opciones que ofrece las API de TMDB, o incluso incorporando otras nuevas, se pueden seguir ampliando la funcionalidad del chatbot.

- **CI/CD:** Configurar una Continuous Integration (Integración Continua) (CI) y un Continuous Deployment (Despliegue Continuo) (CD). La integración continua es la práctica de fusionar con frecuencia y probar automáticamente los cambios a medida que se envían. El despliegue continuo consiste en desplegar automáticamente los cambios integrados en un entorno de ensayo o producción. Esta combinación facilita la prueba y el despliegue eficiente de modificaciones [10].

- **Accesibilidad:** Para aumentar la accesibilidad del chatbot podríamos incorporar capacidades de comunicación por voz. Esto implicaría usar sistemas capaces de extraer el texto de los mensajes de voz del usuario y enviarlos al modelo Rasa para tratar la petición.

- **Feedback:** Se podría implementar un sistema de recogida de feedback de lo usuarios. Este sistema permitiría a los usuarios valorar si las respuestas proporcionadas por el chatbot son útiles, ayudando a mejorar continuamente los datos de entrenamiento.

- **Otras aplicaciones de mensajería:** Diversificar los canales de acceso permite llegar a una audiencia más amplia y garantiza que los usuarios puedan interactuar con el chatbot en el entorno que prefieran.

Apéndices

Apéndice A

Repositorios

Repositorios con ejemplos de proyectos Rasa que me han servido para aprender:

- https://github.com/RasaHQ/rasa/blob/main/examples
- https://github.com/RasaHQ/spaCy-integration-demo
- https://github.com/RasaHQ/conversational-ai-course-3.x
- https://github.com/josejulio/rasa-form-demo

Apéndice B

Manual de instalación

Tecnologías necesarias para la instalación y ejecución del chatbot:

- git
- Python 3.10
- pip
- Docker y Docker Compose
- Nix y flakes (Opcional)
- Poetry (Opcional)
- ngrok (Opcional)

Necesitamos un terminal con la versión 3.10 de Python y la herramienta pip (o pip3) con la que poder instalar las dependencias. Pasos a seguir:

```
git clone https://gitlab.inf.uva.es/albcast/tfg.git

# Acceder al directorio
cd tfg

# Creamos entorno virtual
python -m venv .venv

# Instalar dependencias del proyecto
pip install -r requirements.txt
```

Código B.1: Pasos de instalación

Tras ejecutar estos comandos ya tendríamos todas las dependencias del proyecto listas.

El uso de Nix y Poetry es opcional ya que son herramientas que facilitan el desarrollo y es un requisito para la instalación del chatbot.

B.1. Nix y flakes (Opcional)

Nix es una herramienta que simplifica la instalación de paquetes en nuestra terminal. En particular, la uso para gestionar diferentes versiones de Python que necesito en mis proyectos. Por ejemplo, utilizo Python 3.10 para TFG y Python 3.12 en otro, y Nix me permite manejar ambas versiones simultáneamente sin conflictos, asegurando un entorno de desarrollo organizado y eficiente.

Una vez instalado Nix y su modulo flakes, siguiendo las guías oficiales, es necesario instalar y activar el paquete `direnv`.

Con todo esto instalado (Nix, flakes y direnv) con tan solo entrar al directorio donde hemos clonado el proyecto Nix se encargará de descargar Python 3.10 y crear el entorno virtual. Pasos a seguir:

```
git clone https://gitlab.inf.uva.es/albcast/tfg.git

# Acceder al directorio
cd tfg

# Permitir el uso de direnv
direnv allow

# Instalar dependencias del proyecto
pip install -r requirements.txt
```

Código B.2: Pasos de instalación

B.2. Poetry (Opcional)

Poetry es una herramienta de gestión de dependencias y empaquetamiento para proyectos de Python. Se puede usar como sustituto de pip. Pasos a seguir:

```
git clone https://gitlab.inf.uva.es/albcast/tfg.git

# Acceder al directorio
cd tfg

# Instalar dependencias del proyecto y crear entorno virtual
poetry install
```

Código B.3: Pasos de instalación

B.3. Ngrok (Opcional)

Ngrok también sería una herramienta que me ha ayudado durante el desarrollo y que me facilita exponer la API con el modelo de Rasa al mundo para que pueda ser accesible por los servidores de Telegram. Esta herramienta es externa al proyecto, no hay ningún fichero asociado a ella.

Todo los ficheros necesarios para utilizar las distintas herramientas se encuentran en el repositorio del proyecto.

Apéndice C

Manual de configuración

C.1. Configuración

Antes de poner en funcionamiento el es necesario realizar las configuraciones necesarias para poder utilizar los servicios externos de los que dependemos: Telegram y TMDB.

C.1.1. Telegram

- Token para el bot de Telegram
- Nombre del bot de Telegram
- URL pública
- API Key de la API TMDB

Dentro del fichero `credentials.yml` es necesario especificar las los parámetros necesarios para conectar el chatbot a Telegram.

El primer paso es obtener el token de nuestro bot en Telegram. Para conseguirlo, debes mantener una conversación con `@BotFather` en Telegram, este bot es un asistente que te facilita la creación de otros bots.

1. Busca al usuario `@BotFather` en Telegram.
2. Utiliza el comando `/newbot` para crear un nuevo bot.
3. El `@BotFather` te solicitará un nombre para el bot. Debes proporcionar uno que termine en bot o _bot.
4. Para finalizar el `@BotFather` te entrega el token.

También puedes personalizar el bot añadiendo detalles como una descripción, comandos específicos y una foto de perfil.

El siguiente campo en el fichero `credentials.yml` es verify, en él se debe incluir el nombre que se le ha dado al chatbot en Telegram.

Por último, en el apartado webhook_url se le proporciona la url pública para acceder a la API donde está el modelo de Rasa expuesto. A la url donde esté publicado el chatbot es necesario añadirle el siguiente sufijo al path `/webhooks/telegram/webhook` ya es el webhook correspondiente al servicio de Telegram. Por ejemplo si tenemos nuestro bot bajo el dominio *https://example.com* debemos escribir la siguiente url *https://example.com/webhooks/telegram/webhook*.

```
telegram:
  access_token: "7148448146:AAFZcMQbz8_Akmy3SqmyiJcl7xf-fEWFpaM"
  verify: "tfgrasabot"
  webhook_url: "https://example.com/webhooks/telegram/webhook"
```

Código C.1: Ejemplo de redenciales chatbot para Telegram

El token de Telegram debe ser privado ya que sino cualquiera podría hacer uso del token para montar su propia implementación de bot.

C.1.2. TMDB

Para poder hacer uso de los datos que nos proporciona la API de TMDB es necesario registrarse en su plataforma y obtener un API Key. Este API Key se utiliza en las llamadas a la API para identificarnos, sin él no podremos usar la API.

Dentro del fichero `actions/tmdb.py` existe una variable global llamada API_KEY se debe sustituir esa variable por la API Key proporcionada.

La API Key de TMDB debe ser privada ya que sino cualquiera podría identificarse con nuestra cuenta en la API.

Apéndice D

Manual de despliegue

Una vez tengamos las dependencias instaladas y el proyecto configurado podremos ejecutar el proyecto.

En primer lugar es necesario entrenar el modelo. Los modelos entrenados por Rasa no se guardan en el repositorio del proyecto por lo que la primera vez que vayamos a desplegar el chatbot es necesario entrenar el modelo.

Después lanzamos los contenedores docker con los servicios MongoDB y MongoExpress que nos servirán para almacenar y analizar las conversaciones.

Finalmente lanzamos los procesos con el modelo de Rasa. En un primer terminal levantamos el proceso que levanta la API con el modelo de Rasa. En un segundo terminal levantamos el proceso con la API que contiene las acciones personalizadas.

Asumiendo que ya está el proyecto descargado, instaladas las dependencias y configurado, nos situamos en la carpeta del proyecto y ejecutamos los siguiente pasos:

```
# Entrenar modelo
rasa train

# Contenedores tracker
docker compose up -d

# Lanzar proceso con el modelo Rasa
rasa run -m models --endpoints endpoints.yml

# Lanzar segundo proceso con las acciones
rasa run actions
```

Código D.1: Pasos de despliegue

El proceso que contiene la API con el modelo de Rasa se debe exponer para sea alcanzable por Telegram. Por defecto el puerto que utiliza es el 5005.

Opcionalmente si no se quiere exponer directamente el servidor que aloja el proceso a internet

se puede utilizar ngrok.

```
# Lanzar ngrok con redirección al puerto 5005
.\ngrok http 5005
```

Código D.2: Pasos ngrok

Apéndice E

Manual de usuario

Una vez se tenga el chatbot desplegado y listo para usarse el usuario final debe buscar en Telegram al usuario `@tfgrasabot` como se muestran en E.1.

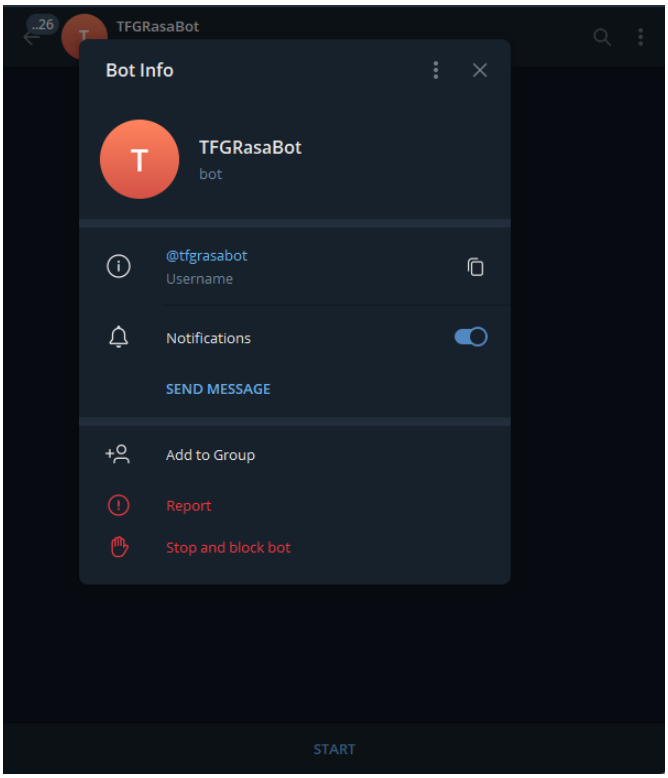

Figura E.1: Nombre del chatbot

Para comenzar la conversación pulsamos el botón start que provocará que se envíe un primer mensaje al chatbot con el texto `/start`. Este texto se tuvo en cuenta en los datos de entrenamiento del chatbot como un ejemplo de saludo.

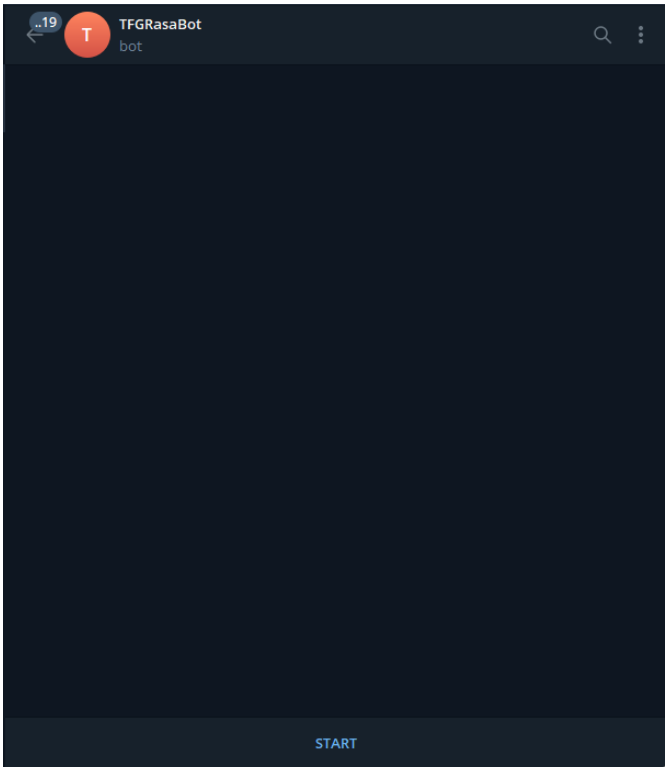

Figura E.2: Comienzo de conversación

Apéndice F

Código

El código con el código desarrollo a lo largo del proyecto se encuentra públicado en el Gitlab de la escuela.

Enlace: gitlab.inf.uva.es/albcast/tfg

Bibliografía

[1] Javier Cepedano. *Diferencias entre NLP, NLU y NLG*. https://www.upbe.ai/blog/diferencias-entre-nlp-nlu-y-nlg/, 2020. (acceso: 17-04-2024).

[2] Juan Gutiérrez. *Convenio colectivo de consultoría*. https://planesdefuturo.mapfre.es/economia-domestica/empleo/convenio-colectivo-consultoria/, 2023. (acceso: 09-06-2024).

[3] IBM. *¿Qué es un chatbot?* https://www.ibm.com/es-es/topics/chatbots. (acceso: 08-05-2024).

[4] ipglobal. *¿Cómo entiende un Bot? NLU en Rasa*. https://www.ipglobal.es/como-entiende-un-bot-nlu-en-rasa/. (acceso: 11-05-2024).

[5] Rasa Jira. *When integrating a Telegram bot with the Rasa framework, I am consistently encountering a 'RuntimeError' stating that the event loop is closed.* https://rasa-open-source.atlassian.net/browse/OSS-764, 2024. (acceso: 07-04-2024).

[6] Gianna Maderis. *Top 22 benefits of chatbots for businesses and customers*. https://www.zendesk.es/blog/5-benefits-using-ai-bots-customer-service/, 2024. (acceso: 08-05-2024).

[7] Alan Nichol. *Rasa CDD Docs*. https://rasa.com/blog/conversation-driven-development-a-better-approach-to-building-ai-assistants/, 2020. (acceso: 10-05-2024).

[8] Rasa. *Rasa CDD Docs*. https://rasa.com/docs/rasa/conversation-driven-development. (acceso: 02-03-2024).

[9] Rasa. *Rasa Config Overview*. https://learning.rasa.com/conversational-ai-with-rasa/pipeline/. (acceso: 10-05-2024).

[10] Rasa. *Rasa Docs*. https://rasa.com/docs/rasa. (acceso: 02-03-2024).

[11] Rasa. *Rasa Responses Docs*. https://rasa.com/docs/rasa/responses. (acceso: 02-03-2024).

[12] Rasa. *Rasa Rules Docs*. https://rasa.com/docs/rasa/rules. (acceso: 02-03-2024).

[13] Rasa. *Rasa Stories Docs*. https://rasa.com/docs/rasa/stories. (acceso: 02-03-2024).

[14] Rasa. *Conversational AI with Rasa Open Source 3.x*. https://youtube.com/playlist?list=PL75e0qA87dlEjGAc9j9v3a5h1mxI2Z9fi&si=fn868ulMSvBJN5tw, 2021. (acceso: 06-03-2024).

[15] TMDB API Reference. *Version 3 of The Movie Database (TMDB) API*. https://developer.themoviedb.org/reference/, 2024. (acceso: 01-04-2024).

[16] Antonio Rodríguez. *Salarios en tecnología 2023 [España]*. https://www.getmanfred.com/blog/salarios-en-tecnologia-2023-espana, 2023. (acceso: 09-06-2024).

Siglas

API Application Programming Interface. 17

CD Continuous Deployment (Despliegue Continuo). 90

CDD Conversation-Driven Development (Desarrollo guiado por la conversación). 18, 29, 30

CI Continuous Integration (Integración Continua). 90

IA Inteligencia Artificial. 16, 17, 29

NLP Natural Language Processing (Procesamiento del Lenguaje Natural). 16, 17

NLU Natural Language Understanding (Comprensión del Lenguaje Natural). 16, 20, 29, 89

TMDB TheMovieDatabase (https://www.themoviedb.org/). 17, 35, 38, 39, 43, 48, 49, 52, 56, 70, 87, 90, 96, 97