Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Proyecto de fin de Grado en Ingeniería en Tecnologías de la Información

Chatbot para atención ciudadana en la administración pública con apoyo de datos abiertos

Alejandro Ahumada Pérez

Dirigido por: Fernando López Ostenero

Curso 2023/2024, convocatoria diciembre

Chatbot para atención ciudadana en la administración pública con apoyo de datos abiertos

Proyecto de fin de Grado en Ingeniería en Tecnologías de la Información

de modalidad específica

Realizado por: Alejandro Ahumada Pérez

Dirigido por: Fernando López Ostenero

Fecha de lectura y defensa: 15 de diciembre de 2023

Agradecimientos

A Ro, por dejarme ganar al final haciendo trampas.

A Ellie, por portarse estupendamente en las últimas semanas.

A la Carmela y el Pacote, por apoyarme cuando dejé la informática por el diseño y viceversa.

A Fernando López, por aceptar este reto y acompañarme en el camino con enorme profesionalidad y paciencia.

A Fran Delgado, por todo lo que aprendí(mos) trabajando codo a codo, su apoyo en el germen de este proyecto y por escuchar mis "levanteras tech" incluso cuando nuestra vida profesional tomó caminos separados.

Resumen

Este documento detalla el desarrollo de un chatbot sobre la plataforma Rasa Open Source, complementada con Generación de Lenguaje Natural mediante GPT-4, para ayudar a la ciudadanía a acceder a la información publicada en portales de datos abiertos de la administración pública.

Busca permitir al público general acceder y comprender información relevante sobre, por ejemplo, contratación, programas, o presupuestos, fomentando así la transparencia y la rendición de cuentas. De esta forma, empoderamos a la ciudadanía para que ejerza un control social activo, monitoreando la gestión, denunciando irregularidades y participando en procesos de auditoría y fiscalización.

Estos objetivos permiten establecer una relación más dinámica y colaborativa entre la administración pública y la ciudadanía, promoviendo una gobernanza más inclusiva, responsable y efectiva.

Además, los datos abiertos pueden desempeñar un papel crucial en el avance de los Objetivos de Desarrollo Sostenible (ODS) al proporcionar información basada en hechos transparente, accesible y reutilizable.

Abstract

This document outlines the creation process of a chatbot powered by Rasa Open Source, assisted with Natural Language Generation using GPT-4. Its main goal is to support citizens in accessing information published on open data portals of the public administration.

It aims to enable the general public to access and comprehend relevant information on topics such as procurement, programs, or budgets, promoting transparency and accountability. We empower citizens to exercise active social control, monitoring public management, reporting irregularities, and participating in audit and oversight processes.

These objectives facilitate a more dynamic and collaborative relationship between public administration and citizens, promoting more inclusive, responsible, and effective governance.

Moreover, open data can play a crucial role in advancing the Sustainable Development Goals (SDGs) by providing transparent, accessible, and reusable fact-based information.

Palabras clave

Aprendizaje Automático, Aprendizaje Profundo (Deep Learning), Chatbot, Comprensión del Lenguaje Natural (CLN), conjunto de datos, Datos Abiertos, Generación de Lenguaje Natural (GLN), GPT-4, Inteligencia Artificial Conversacional, Modelo Grande de Lenguaje (LLM), Procesamiento del Lenguaje Natural (PLN), Rasa Open Source.

Keywords

Chatbot, Conversational AI, Dataset, Deep Learning, GPT-4, Machine Learning, Natural Language Processing (NLP), Natural Language Understanding (NLU), Natural Language Generation (NLG), Large Language Model (LLM), Open Data, Rasa Open Source.

Índice

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

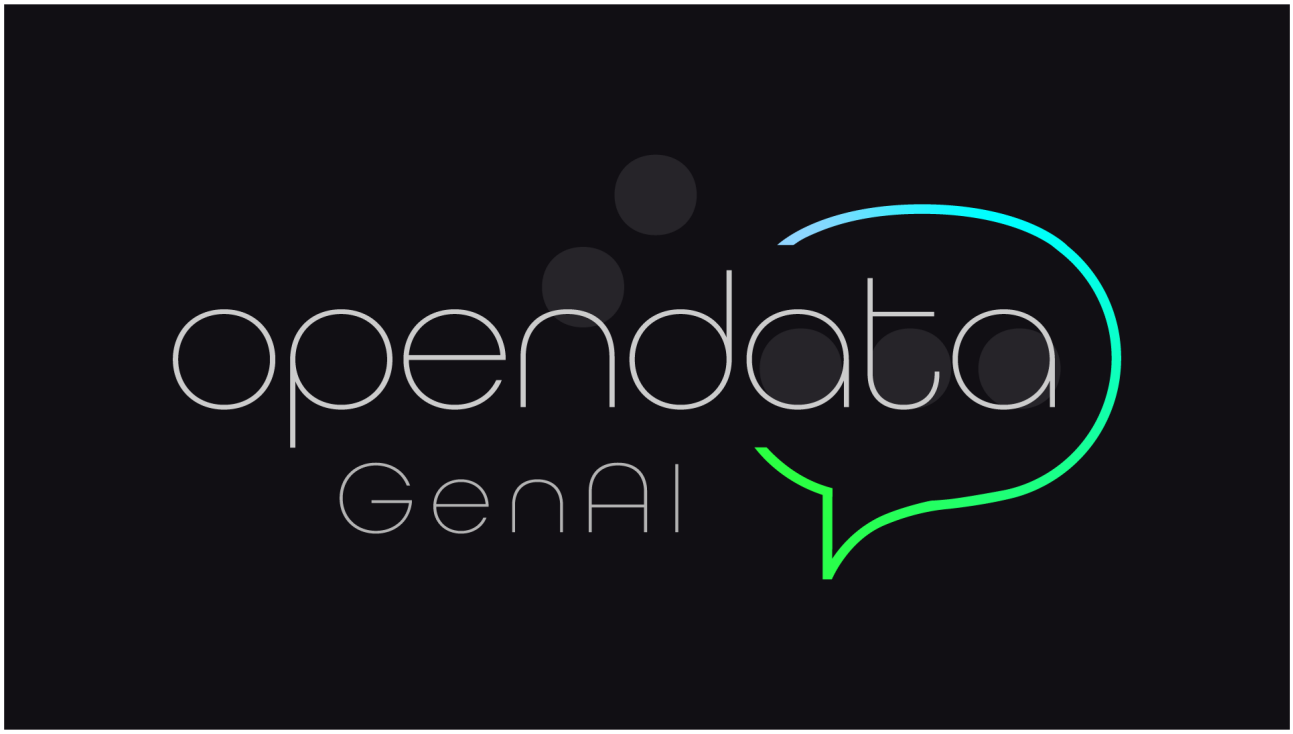


Figura 1.1: Logo de OpenData GenAI Bot

1.1. Motivación

Este proyecto surge de una necesidad detectada durante el desarrollo y puesta en funcionamiento del Portal de Datos Abiertos del Ayuntamiento de Cádiz.

Pese a la buena acogida que tuvo el proyecto[1] entre organizaciones y profesionales del sector, así como su uso para investigación, especialmente de ámbito académico; detectamos que la ciudadanía tenía dificultades para acceder y comprender la información publicada.

Por ello, a raíz de la publicación de un artículo[2] en el blog de datos.gob.es se planteó como solución la creación de un chatbot que facilitase la utilización y comprensión de los datos.

[1] https://www.diariodecadiz.es/cadiz/Ayuntamiento-Cadiz-felicitacion-Aporta-portal-datos-abiertos_0_1721828956.html

[2] https://datos.gob.es/es/blog/chatbots-o-asistentes-virtuales-en-las-aapp-para-democratizar-el-uso-de-datos-abiertos

A nivel personal me pareció un reto muy estimulante, ya que me permitía investigar áreas a mi parecer muy interesantes como son el Procesamiento de Lenguaje Naural, Large Language Models e Inteligencia Artificial Generativa, así como la integración con APIs o arquitecturas cloud. Estas tareas se salen de las que suelo realizar habitualmente en mi desempeño profesional, permitiéndome a la vez afianzar conceptos adquiridos durante el grado y profundizar en áreas de conocimiento que no me son tan familiares.

1.2. Objetivos

El objetivo del proyecto es crear un chatbot que ayude a la ciudadanía a comprender los conjuntos de datos alojados en un portal de datos abiertos, con una interfaz sencilla que permita interacción mediante lenguaje natural con mensajes de texto.

Dicha solución debe entender las peticiones del usuario, permitiendo realizar búsquedas de conjuntos de datos con lenguaje natural, así como descargar los archivos de dichos conjuntos de datos en los diferentes formatos disponibles. Adicionalmente, el software debe ser capaz de generar gráficas con los datos.

Todo esto se debe complementar con capacidades generativas, generando descripciones de forma dinámica, extrayendo datos estadísticos y atendiendo a consultas personalizadas de los usuarios realizadas en texto.

Con esto se busca generar un impacto positivo en la sociedad:

- Avanzando en la digitalización de la Administración Pública.
- Ayudando a la ciudadanía a acceder y comprender la información publicada en portales de datos abiertos.
- Empoderando a la ciudadanía para que ejerza un control social activo, monitoreando la gestión, denunciando irregularidades y participando en procesos de auditoría y fiscalización.
- Estableciendo una relación más dinámica y colaborativa entre la administración pública y la ciudadanía, promoviendo una gobernanza más inclusiva, responsable y efectiva.
- Mejorando la calidad de los conjuntos de datos, generando descripciones de forma dinámica.
- Colaborando en el avance de los Objetivos de Desarrollo Sostenible (ODS) al proporcionar información basada en hechos transparente, accesible y reutilizable. Especialmente relevante el ODS 16. Paz, justicia e instituciones sólidas.

1.3. Estructura del documento

Esta memoria está dividida en siete capítulos y cuatro anexos:

Capítulo 1 Introducción: motivación y objetivos.

Capítulo 2 Estado del arte: conceptos esenciales, análisis de plataformas para IA conversacional, Large Language Models, chatbots y aplicaciones de mensajería.

Capítulo 3 Propuesta: descripción general de la propuesta a realizar.

Capítulo 4 Diseño: recomendaciones para diseñar un sistema conversacional, requisitos del sistema, IA conversacional con Rasa, IA generativa con OpenAI y arquitectura.

Capítulo 5 Desarrollo: configuración de Rasa, definición de dominio, datos para el entrenamiento e integración son servicios de terceros.

Capítulo 6 Pruebas: validación de datos de entrenamiento, pruebas end-to-end en Rasa y pruebas con usuarios.

Capítulo 7 Conclusiones: dificultades, objetivos alcanzados, estudio de costes de mantenimiento y hoja de ruta con trabajos futuros.

Anexo 1 DKAN: descripción y justificación de uso de la tecnología.

Anexo 2 Entorno de desarrollo local. requisitos de software, entornos virtuales en Python, instalación de dependencias, ejecución de servidor local mediante herramienta CLI y Docker.

Anexo 3 Puesta en producción: Alta en servicios de terceros, configuración de NGINX y deploy con Docker Compose.

Anexo 4 Manual de usuario: guía breve para usuarios finales.

Capítulo 2

Estado del arte

Este capítulo pretende ofrecer una visión general de los conceptos, tecnologías y herramientas que serán fundamentales para el correcto desarrollo del proyecto.

2.1. Conceptos esenciales

En esta sección definimos de forma breve algunos conceptos esenciales para seguimiento del proyecto, pertenecientes a las diferentes áreas de estudio implicadas en el desarrollo del mismo.

Aprendizaje Automático (Machine Learning)

El Aprendizaje Automático es una rama de la inteligencia artificial que capacita a las computadoras para aprender de los datos y mejorar su rendimiento en tareas sin ser programadas explícitamente.

Aprendizaje Profundo (Deep Learning)

El Aprendizaje Profundo es un subconjunto del aprendizaje automático que implica redes neuronales con múltiples capas, permitiendo que el modelo aprenda automáticamente representaciones jerárquicas de los datos.

Procesamiento del Lenguaje Natural (PLN)

En inglés Natural Language Processing (NLP). Es una rama de la Inteligencia Artificial que estudia el desarrollo y aplicación de modelos y algoritmos computacionales para permitir que las computadoras comprendan, interpreten y generen el lenguaje humano. Integra las disciplinas Comprensión del Lenguaje Natural (CLN) y Generación de Lenguaje Natural (GNL).

Comprensión del Lenguaje Natural (CLN)

En inglés Natural Language Understanding (NLU). Es subconjunto del PLN que se centra en la capacidad de las máquinas para comprender e interpretar el lenguaje humano, permitiéndoles

extraer significado, contexto e intención a partir de entradas textuales o habladas.

Intent

El término *"intent"* en el contexto de PLN, se refiere al objetivo expresado en la entrada de un usuario. Identificar la intención ayuda al sistema a comprender lo que el usuario desea lograr o qué acción está solicitando. El reconocimiento de la intención es crucial en aplicaciones como chatbots, asistentes virtuales y sistemas activados por voz.

Entity

En PLN, una *"entity"*, o entidad, representa una pieza específica de información dentro de la entrada del usuario que el sistema necesita extraer. Las entidades son las variables o puntos de datos que proporcionan contexto a la intención. Pueden clasificarse en diferentes tipos, como fechas, ubicaciones, cantidades, etc. Extraer entidades es crucial para que los sistemas recopilen los detalles necesarios para cumplir con precisión la solicitud del usuario.

Generación de Lenguaje Natural (GLN)

En inglés Natural Language Generation (NLG), es un subconjunto del PLN que se ocupa de la creación de texto o habla similar al humano a partir de patrones de lenguaje, facilitando la generación automatizada de un lenguaje coherente y contextualmente relevante.

Modelo Grande de Lenguaje (LLM)

Los Modelos Grandes de Lenguaje (o Modelos de Lenguaje Grandes) son potentes arquitecturas capaces de aprender patrones de lenguaje complejos y generar texto contextualmente relevante. Se utilizan en GLN.

Prompt

Se refiere a la entrada o consulta proporcionada al modelo para generar una respuesta. Cuando interactúas con un modelo de lenguaje, introduces un *"prompt"* que el modelo utiliza para generar texto o responder preguntas. El prompt puede ser una pregunta, una afirmación o cualquier forma de entrada que instruya al modelo sobre qué tipo de información o respuesta se espera. Se pueden diferenciar dos categorías: *prompts* de sistema y *propmts* de usuario.

Token

Es la unidad de texto que el modelo procesa. Pueden variar en tamaño y son básicamente fragmentos de texto que el modelo lee y genera durante sus operaciones. La tokenización es el proceso de descomponer una secuencia de texto en tokens individuales, y es un paso fundamental en la preparación de datos de texto para la entrada en modelos de lenguaje. Un token se equivale aproximadamente a 4 caracteres (en inglés).

Inteligencia Artificial Conversacional

Se refiere al uso de tecnologías de Inteligencia Artificial para facilitar una comunicación natural e interactiva entre humanos y computadoras mediante voz o texto.

Chatbot

Un chatbot es un software programado para interactuar con los usuarios de forma conversacional, facilitando información, respondiendo a consultas o realizando tareas.

Datos Abiertos

Son datos que están disponibles de forma libre para que cualquier persona pueda acceder, usar, modificar y compartir. Promueven la transparencia, rendición de cuentas y se basan en el concepto de que cierto tipo de información, especialmente en el ámbito gubernamental, deben ser accesibles para el público general sin restricciones.

Conjunto de datos

Es una colección estructurada de datos. Normalmente incluye metadatos que facilitan la búsqueda y procesado automático de los mismos. Pueden variar en tamaño, formato o tipo de información. Se pueden encontrar, por ejemplo, en Portales de Datos Abiertos.

2.2. Plataformas para IA Conversacional

Realizaremos un análisis de diferentes plataformas de IA conversacional, que serán la base del proyecto y darán solución los requisitos de Comprensión de Lenguaje Natural del proyecto.

Amazon Lex

- Desarrollado por Amazon[1].
- Basado en la misma tecnología que Amazon utiliza para su asistente inteligente Alexa.
- Integración con la plataforma AWS, que permite conectar de forma fácil con otros servicios de AWS como S3, Lambda o DynamoDB, entre otros.
- Tiene capacidades de reconocimiento de voz (ASR) y entendimiento del lenguaje natural (NLU).
- Facilidad para crear bots de voz y texto.
- Cuenta con un editor visual para construir diálogos.
- No es de código abierto y puede resultar costoso a gran escala.

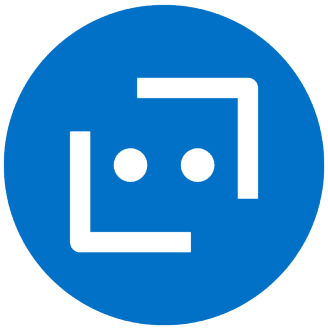
Google Dialogflow

- Desarrollado por Google[2].
- Gran comprensión del lenguaje natural por el respaldo de la tecnología de Google.
- Disponible en dos ediciones: ES (Standard Edition) y CX (Customer Experience Edition), esta última para casos de uso más complejos y con capacidad de manejar conversaciones a mayor escala.
- Ofrece funcionalidades de texto y voz a través de la integración con la API de Google Speech.
- Posee capacidades de machine learning integradas para un mejor reconocimiento de intenciones y entidades.

[1] https://aws.amazon.com/es/lex/
[2] https://cloud.google.com/dialogflow/

- Cuenta con un editor visual para construir diálogos.
- Fácil integración con muchas plataformas de mensajería y asistentes.
- No es de código abierto, pero tiene una capa gratuita que puede ser suficiente para muchos proyectos pequeños a medianos.

Azure Bot Service



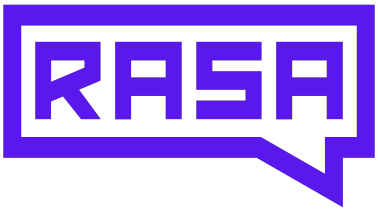
- Desarrollado por Microsoft[3]
- Forma parte de la plataforma de servicios en la nube Azure, por lo que se integra con otros servicios como Azure Functions, Azure Cognitive Services.
- Ofrece capacidades avanzadas de procesamiento de lenguaje natural (NLU) y reconocimiento de voz a través de la integración con Azure Cognitive Services.
- Permite la creación de bots de voz y texto con soporte para diversos canales de mensajería y plataformas.
- Proporciona un entorno de desarrollo visual a través de Azure Bot Composer para diseñar conversaciones y flujos de trabajo de bots.
- Escalabilidad y opciones de despliegue en la nube, ofreciendo flexibilidad en la infraestructura.
- No es de código abierto.
- Puede resultar rentable para empresas que ya utilizan la suite de servicios de Azure debido a la integración estrecha y ventajas en los precios.

[3] https://azure.microsoft.com/es-es/products/ai-services/ai-bot-service/

Wit.ai

- Creado en 2014, fue adquirido por Meta en 2015[4].
- Ofrece funcionalidades de procesamiento de lenguaje natural principalmente para desarrolladores de Meta Messenger, aunque puede usarse independientemente.
- Su punto fuerte es la interpretación del lenguaje y la extracción de entidades.
- Proporciona una manera fácil de entrenar el modelo y mejorarlo con más interacciones.
- Salvo su SDK, no es de código abierto, pero su uso es gratuito.
- Tiene un límite de entre 60-600 peticiones por minuto por usuario según el endpoint, lo cual puede ser bastante limitado en escenarios de alta demanda.
- La comunidad detrás no es tan grande como la de Dialogflow o Lex, por lo que puede haber menos recursos de aprendizaje o soporte comunitario.
- Funcionalidades más limitadas que el resto de opciones.
- Desarrollo poco activo.

Rasa

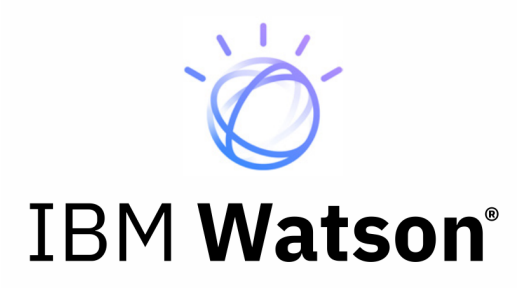


- Ofrece un conjunto de herramientas para construir chatbots más sofisticados y con control de código total[5].

[4] https://wit.ai/
[5] https://rasa.com/

- Es de código abierto y se puede personalizar completamente.
- No cuenta con un editor visual para construir diálogos por defecto, pero está disponible en Rasa X, su versión de pago.
- Es más complejo de configurar que los servicios anteriores y requiere más conocimiento técnico.
- Tiene tres componentes principales: Rasa NLU para el entendimiento del lenguaje, Rasa Core para la gestión de diálogos y Rasa SDK Action Server para la ejecución de acciones.
- Permite un entrenamiento más elaborado para entender mejor las intenciones y contextos de los usuarios.
- No depende de una plataforma cloud específica y se puede desplegar en cualquier servidor o incluso de manera local.
- Permite definir flujos de CI/CD mediante servicios de terceros, por ejemplo, Github Actions.

IBM Watson Assistant



- Desarrollado por IBM[6].
- Ofrece capacidades avanzadas de procesamiento de lenguaje natural (NLP), comprensión del lenguaje (NLU) y aprendizaje automático.
- Permite la creación de chatbots y asistentes virtuales que pueden ser desplegados en una diferentes canales, incluyendo web, móviles, Slack, Facebook Messenger, etc.
- Cuenta con un editor visual para construir diálogos y se puede entrenar fácilmente mediante ejemplos de conversación.
- Puede manejar diálogos complejos y tiene funcionalidades para gestionar el contexto de la conversación a lo largo del tiempo.
- Ofrece una capa gratuita limitada y opciones de pago basadas en las instancias de llamada a la API y otras métricas.
- Se enfoca en facilidad de uso y en la integración con el ecosistema de IBM Cloud.

[6] https://www.ibm.com/es-es/products/watsonx-assistant

- No es de código abierto.

- Soporta la implementación tanto en el cloud de IBM como on-premise, para los clientes que requieren mantener sus datos dentro de su propia infraestructura.

2.3. Modelos Grandes de Lenguaje (LLMs)

Los LLMs se presentan como una de las grandes revoluciones tecnológicas de la última década. Como vimos al inicio del capítulo, son potentes arquitecturas capaces de aprender patrones de lenguaje complejos y generar texto contextualmente relevante. Se utilizan normalmente en tareas de generación de texto, pero también pueden realizar otro tipo de tareas, como traducciones o funcionar como un clasificador.

En esta sección analizaremos los principales LLM, tanto comerciales como open source, tecnología que nos servirá para plantear una solución a nuestras necesidades de Generación de Lenguaje Natural.

2.3.1. Características de un LLM

- **Tamaño:** se entrenan con billones o trillones de parámetros (en su definición inglesa un billón corresponde a 10^9, mientras que un trillón corresponde a 10^{12}).

- **Ventana de contexto:** indica el número máximo de tokens de entrada y de salida que permiten.

- **Proceso de entrenamiento:** Se entrenan en conjuntos de datos enormes provenientes de libros, sitios web y artículos para extraer patrones de lenguaje.

- **Pre-entrenamiento y Fine-tuning:** normalmente son pre-entrenados con un conjunto de datos generalista, para después realizar un proceso de fine-tuning para tareas específicas mediante otros datasets más concretos.

2.3.2. LLMs comerciales

En el último año, varias organizaciones han desarrollado LLM con fines comerciales, que ponen a disposición de sus usuarios mediante pago por uso a través de una API.

Suelen disponer de un SDK que facilita su integración con otras herramientas y su principal ventaja es que eliminan la complejidad de disponer de una infraestructura propia para el uso de un LLM.

GPT-4 Turbo

Modelo de lenguaje desarrollado por OpenAI[7], que se encuentra actualmente en acceso previo. Es la evolución de los conocidos modelos GPT-3.5 y GPT-4, utilizados por ChatGPT.

Entre sus características más destacadas tenemos una ventana de contexto amplia, de 128K tokens, lo que equivale a unas 300 páginas de texto, permite generar texto más completo y coherente.

Es capaz de realizar tareas más complejas que los modelos anteriores y funciona especialmente bien con código y datos estructurados o semi-estructurados.

Dispone de API[8] pública y un SDK[9] para Python y Node.js con un desarrollo muy activo. Adicionalmente, existen librerias[10] disponibles en varios lenguajes y mantenidas por la comunidad. Además, la API comienza a tener capacidades multimodales, complementando la generación de texto mediante prompts, dispondremos de funciones *image-to-text* y de generación de imágenes.

Su precio ese s un 30 % más barato que GPT-4 para tokens de entrada y un 20 % más barato para tokens de salida.

PaLM 2

Desarrollado por Google[11].

Presentado el 10 de mayo de 2023. En la última versión se ha aplicado un escalado de los parámetros para optimizar la computación, que hace que sea mucho más potente que su predecesor pese a tener menor número de parámetros. También se ha mejorado la diversidad de los datos utilizados durante su entrenamiento, así como su arquitectura general.

Las principales mejoras respecto a su primera versión son la inclusión de capacidades multiidioma (ha sido entrenado con textos en más de 100 idiomas), tanto para la comprensión como para la generación; capacidad de razonamiento mejorada gracias a la diversidad de los conjuntos de datos utilizados, especialmente en tareas que impliquen habilidades lógicas y matemáticas; y por último, mejoras en el entendimiento y generación de código en diferentes lenguajes de programación.

Google lo utiliza en Bard[12], y también está disponible para su integración con otras apliaciones mediante una API[13].

[7] https://openai.com/
[8] https://platform.openai.com/docs/api-reference
[9] https://platform.openai.com/docs/libraries/python-library
[10] https://platform.openai.com/docs/libraries/community-libraries
[11] https://ai.google/discover/palm2
[12] https://bard.google.com
[13] https://developers.generativeai.google/products/palm

Claude 2

Desarrollado por Anthropic[14].

Cabe destacar que en su última versión Claude 2.1, presenta una ventana de contexto de 200k tokens, por lo que supera incluso al reciente GPT-4 Turbo en esta característica.

Dispone de API[15], pero lamentablemente en el momento de publicar este documento, no está disponible de forma pública y requiere de la aprobación de una solicitud de acceso anticipado. Además la Unión Europea no se encuentra dentro de las regiones disponibles[16], lo que hace su uso inviable para este proyecto por el momento.

El acceso a la API se facilita mediante un SDK[17] disponible en Python y TypeScript. En su última versión, se introducen *system propmts* similares a los disponibles en la familia GPT, para dar instrucciones y optimizar el rendimiento.

Cohere

Desarrollado por **Cohere**[18], empresa fundada por Aidan Gómez, uno de los autores del artículo que introdujo la arquitectura **transformer** [1].

Sus modelos destacan por ser muy eficientes y están optimizados para realizar tareas como: búsqueda semántica, moderación de contenidos, extracción de entidades o clasificación de textos.

Dispone de API[19] propia para el uso de su modelo, además de permitir el uso de sus modelos en soluciones de terceros como Amazon SageMaker[20] y Amazon Bedrock[21]. Además, ofrece un SDK[22] para facilitar su integración en proyectos Python.

Inflection-2

Desarrollado por Inflection[23], destaca por ser uno de los modelos de lenguaje con una mejor relación coste-eficiencia. Es usado por Pi, el chatbot desarrollado por Inflection.

Actualmente, no dispone de API para integración en otros proyectos. Además, Inflection tiene la peculiaridad de no trabajar activamente en la investigación sobre AGI, planteando una cultura empresarial que fomente una visión ética y segura[24] sobre los avances en IA que contrasta con las prioridades del resto de organizaciones.

[14] https://www.anthropic.com/
[15] https://docs.anthropic.com/claude/reference
[16] https://www.anthropic.com/claude-ai-locations
[17] https://docs.anthropic.com/claude/reference/client-sdks
[18] https://cohere.com/
[19] https://dashboard.cohere.com/
[20] https://docs.aws.amazon.com/sagemaker/
[21] https://us-east-1.console.aws.amazon.com/bedrock/home
[22] https://docs.cohere.com/reference/about
[23] https://inflection.ai
[24] https://inflection.ai/safety

2.3.3. LLMs Open Source

Aunque la complejidad añadida de utilizar un LLM de código abierto hace que nos inclinemos por adoptar una de las soluciones comerciales presentadas, parece idóneo hacer un acercamiento a los modelos de código abierto que han aparecido en los últimos años.

Conviene señalar que el creciente interés por los modelos de código abierto en plataformas como Hugging Face[25], una de las principales plataformas colaborativas sobre IA, ML y NLP, ha hecho florecer una comunidad que comparte modelos, datasets, y aplicaciones, obteniendo avances significativos en áreas que antes sólo estaban al alcance de grandes empresas y universidades.

Llama 2

Desarrollado por Meta, **Llama 2**[26] es el principal LLM de código abierto desarrollado por una *"Big Tech"*, junto a FLAN-T5 (desarrollado por Google).

Presenta tres tamaños de modelo, con 7B, 13B y 70B parámetros respectivamente, y una ventana de contexto de 1024 tokens. Permite realizar tareas de generación de texto, respuesta a preguntas y traducción.

Tiene una licencia personalizada de Meta y se permite su uso en proyectos comerciales.

Alpaca

Alpaca[27] ha sido desarrollado por investigadores de la Universidad de Stanford.

Tomando como partida el modelo **LLaMa 7B** y apicando ténicas de *fine-tuning*, se obtienen resultados similares al modelo text-davinci-003 de OpenAI.

No se permite su uso en proyectos comerciales.

Falcon

Falcon[28] ha sido desarrollado por Technology Innovation Institute (TII).

Disponible en cuatro tamaños diferentes: 1,3B 7,5B, 40B y 180B. Tiene un tamaño de modelo grande en comparación con otras propuestas analizadas, y ha sido entrenado con un conjunto de datos amplio de gran calidad.

Tiene licencia Apache 2.0 y se permite su uso en proyectos comerciales.

[25] https://huggingface.co/
[26] https://ai.meta.com/llama/
[27] https://crfm.stanford.edu/2023/03/13/alpaca.html
[28] https://falconllm.tii.ae/

BLOOM

BLOOM[29] es un modelo basado en la arquitectura transformer. Desarrollado principalemte por el equipo BigScience[30] de la organización Hugging Face, tiene 178 mil millones de parámetros y ofrece resultados similares a GPT-3 de OpenAI.

Tiene licencia Apache 2.0 y se permite su uso en proyectos comerciales.

FLAN-T5

Desarollado por Google. Es la evolución del modelo T5, FLAN-T5[31] y está disponible en cinco tamaños y publicado en Hugging Face[32].

Tiene licencia Apache 2.0 y se permite su uso en proyectos comerciales.

2.3.4. Comparativa y benchmarking

La tabla 2.1 se muestra una comparativa de las principales características de diferentes LLMs. Esta tabla se ha elaborado con información proporcionada por las propias organizaciones que desarrollan los modelos de lenguaje. Para nuestro proyecto podemos analizar características como el número de tokens de contexto, esencial para disponer de muchos tokens de entrada que permitan el análisis de conjuntos de datos extensos.

En la tabla 2.2 podemos ver los resultados del benchmark MMLU elaborado por Inflection, sobre tareas de ámbito amplio (con niveles desde educación secundaria a desempeño profesional).

La tabla 2.3, se muestra un benchmarking, también elaborado por Inflection, en el que se evalúa el desempeño de diferentes LLMs para tareas de cálculo matemático y desarrollo de código, que serán primordiales en nuestro caso de uso para el análisis automático de los diferentes conjuntos de datos.

La comunidad de desarrollo mantiene en Hugging Face un leaderboard[33] público en el que evalúan una gran cantidad de modelos.

2.4. Chatbots

Como vimos anteriormente, un chatbot es un software programado para interactuar con los usuarios de forma conversacional, facilitando información, respondiendo a consultas o realizando tareas. Pueden hacer uso de un LLM u otras técnicas, como las reglas.

[29] https://arxiv.org/abs/2211.05100
[30] https://huggingface.co/bigscience/bloom
[31] https://arxiv.org/pdf/2210.11416.pdf
[32] https://huggingface.co/google/flan-t5-base
[33] https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

Modelo	Organización	Nº de parámetros	Tokens (max)	API/Uso público
GPT-4 Turbo	OpenAI	Desconocido	128k	API
GPT-4	OpenAI	Desconocido	8K	API
GPT-3.5 Turbo	OpenAI	175B	16K	API
GPT-3.5	OpenAI	175B	4K	API
PaLM 2	Google	340B	8196/1024 (i/o)	API
PaLM 1	Google	540B	512	No
Llama 2	Meta	70B	1024	Uso público
LLaMA-1	Meta	65B	256	Uso público
Claude 2	Anthropic	130B	200k	API, por invitación
Claude 1	Anthropic	52B	256	API, por invitación
Cohere	Cohere	137B	1024	API
Inflection-2	Inflection	175B	1024 (estimado)	No
Grok	X	33B	Desconocido	No

Tabla 2.1: Comparativa de LLMs. Fuente: elaboración propia.

Modelo	LLaMA 2	Grok	PaLM 2 L	Inflection-2	Claude 2	GPT-3.5	GPT-4
Puntuación	68,9	73,0	78,3	79,6	78,5	70.0	86,4

Tabla 2.2: Benchmark MMLU de principales LLMs comerciales.
Fuente: Inflection.

Modelo	MBPP@1	HumanEval@1	GSM8k(8)	MATH
LLaMA 2 (70B)	45,0	29,9	56,8	13,5
Grok	-	63,2	62,9	23,9
PaLM 2 L	-	-	80,7	34,3
GPT-4	-	67,0	92,0	-
Inflection-1	43,8	35,4	62,9	16,7
Inflection-2	53,0	44,5	81,4	34,8

Tabla 2.3: Benchmark en tareas matemáticas y código de principales LLMs comerciales.
Fuente: Inflection.

- **Chatbots basados en reglas:** estos chatbots utilizan un conjunto de reglas predefinidas para responder a las entradas del usuario. Las reglas pueden ser simples, como responder a una pregunta con una respuesta predefinida, o más complejas, que permiten una

conversación coherente más cercana al lenguaje humano.

- **Chatbots basados en aprendizaje automático:** utilizan algoritmos de aprendizaje automático para aprender a responder a las entradas del usuario. Los algoritmos de aprendizaje automático son entrenados en un conjunto de datos de texto y código, lo que permite a los chatbots aprender a generar texto, traducir idiomas, escribir diferentes tipos de contenido creativo y responder a preguntas de forma informativa. Esta categoría incluye a los LLM.

A continuación incluimos algunos ejemplos de chatbots destacados.

ChatGPT

Desarrollado por OpenAI[34] y publicado en noviembre 2022, tiene la capacidad de mantener conversaciones complejas y naturales, aprendiendo y adaptándose a la conversación. Entre sus funciones destacan: responder a preguntas de forma informativa, generar diferentes formatos de texto creativo o realizar tareas de traducción. Se encuentra disponible a través de una aplicación web y aplicación móvil para dispositivos iOS.

Hace uso de la familia GPT de modelos LLM, en concreto GPT-3.5 para su versión gratuita y GPT-4 en su versión de pago. La versión premium incluye capacidades multimodales, integrando DALL-E para generación de imágenes, búsqueda en web e importación de archivos.

En la figura 2.1 vemos un ejemplo de consulta en ChatGPT.

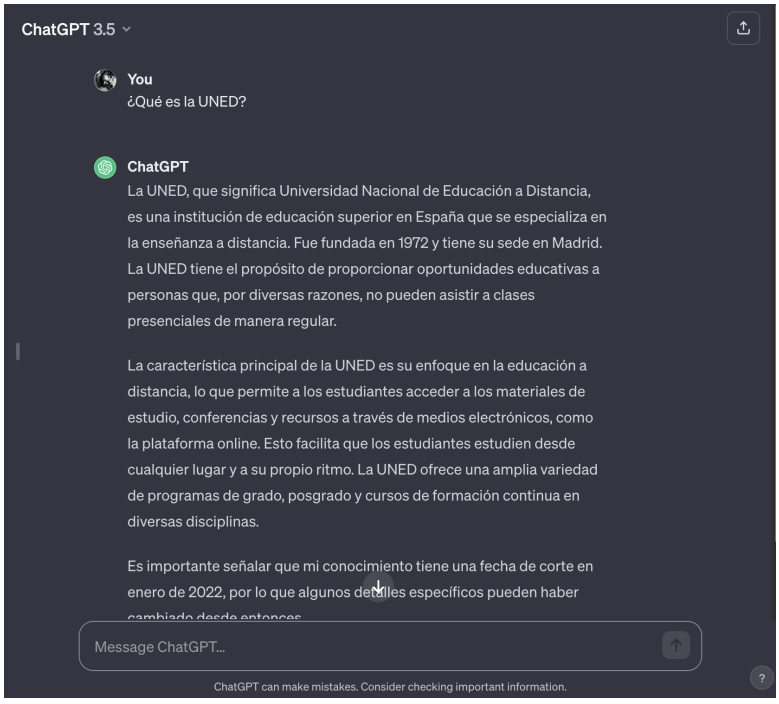


Figura 2.1: Ejemplo de consulta en ChatGPT.

[34] https://chat.openai.com/

Bard

De forma análoga al ejemplo anterior, Bard[35] es la propuesta de chatbot generalista de Google, que hace uso de su modelo más avanzado, PaLM 2. Está disponible a través de una aplicación web.

Tiene funcionalidades similares a ChatGPT, por lo que se puede catalogar como un chatbot generalista. Es completamente gratuito, disponiendo de búsqueda, pero por el momento no dispone de características multimodales. En la figura 2.2 podemos ver una consulta en Bard.

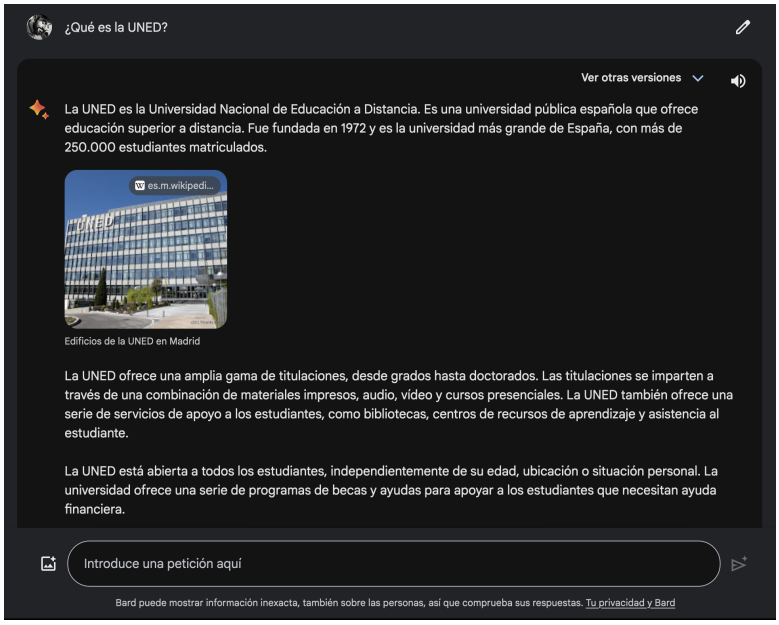


Figura 2.2: Ejemplo de consulta en Google Bard.

Pi

Pi[36] es un chatbot desarrollado por Inflection. Actualmente hace uso de su modelo Inflection-1, aunque esta organización dispone de un modelo más avanzado, Inflection-2.

Se puede utilizar desde su aplicación web y también dispone de aplicación para dispositivos móviles iOS[37] e integración con aplicaciones de mensajería (WhatsApp[38], Instagram[39] y Facebook[40]).

También es de tipo generalista, entrenado en un conjunto de datos amplio, pero es algo más limitado que sus competidores. Destaca por dar respuestas concisas y con una personalidad agradable y comprensiva.

[35] https://bard.google.com
[36] https://pi.ai/talk
[37] https://pi.ai/ios
[38] https://pi.ai/wa
[39] https://pi.ai/ig
[40] https://pi.ai/fb

En la figura 2.3 podemos ver un ejemplo de consulta en el chatbot Pi, en la que podemos apreciar que, pese a comprender la consulta en español, únicamente genera respuestas en inglés.

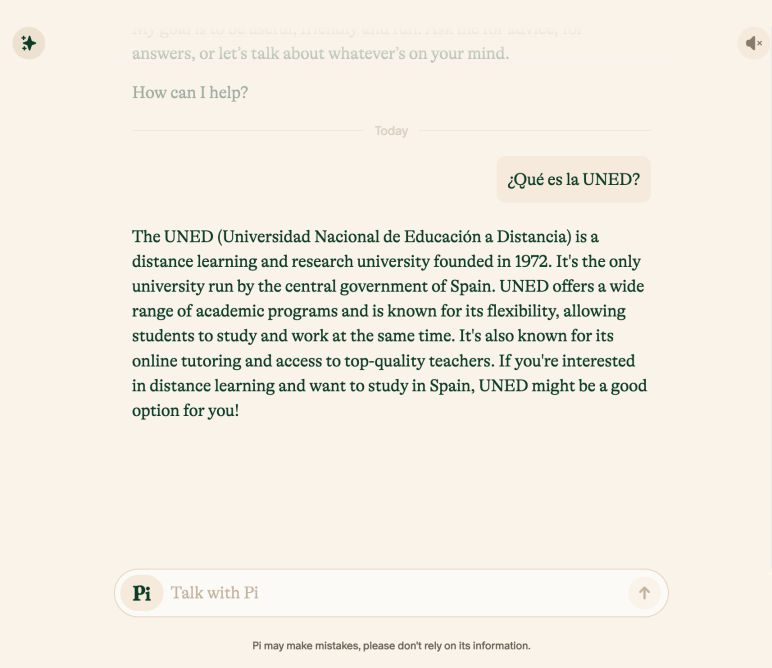


Figura 2.3: Ejemplo de consulta en Pi.

Siri

Encontramos otra categoría, la de asistentes virtuales integrados en el Sistema Operativo, entre los que podemos encontrar Siri, Google Assistant, Amazon Alexa o Samsung Bixby. En concreto, Siri es un asistente virtual desarrollado por Apple y disponible es dispositivos iOS, iPadOS y macOS.

Este tipo de asistentes utilizan reconocimiento de voz y permiten realizar diferentes tareas como: responder preguntas mediante búsquedas en la web, reproducir contenido en audio, establecer alarmas y recordatorios, dar información sobre el clima o controlar dispositivos domésticos inteligentes.

ChatPDF

Además de chatbots de ámbito generalista, ya sean aplicaciones independientes o integrados en el sistema, como los que hemos reseñado, podemos definir otra categoría, chatbots especializados en una tarea concreta.

En esta categoría podemos encontrar varios ejemplo, siendo uno de los más interesantes para nuestro estudio ChatPDF[41], que tiene como cometido el análisis automatizado de documentos en formato PDF, sobre los que podemos hacer consultas en lenguaje natural.

[41] https://www.chatpdf.com

Este tipo de herramientas se asemeja a la solución que buscamos, un chatbot que nos sirva de apoyo a la hora de analizar y comprender datos e información contenida en otro documento.

En la figura 2.4 podemos ver un ejemplo de análisis de una versión previa de la memoria del Proyecto Fin de Grado que nos ocupa.

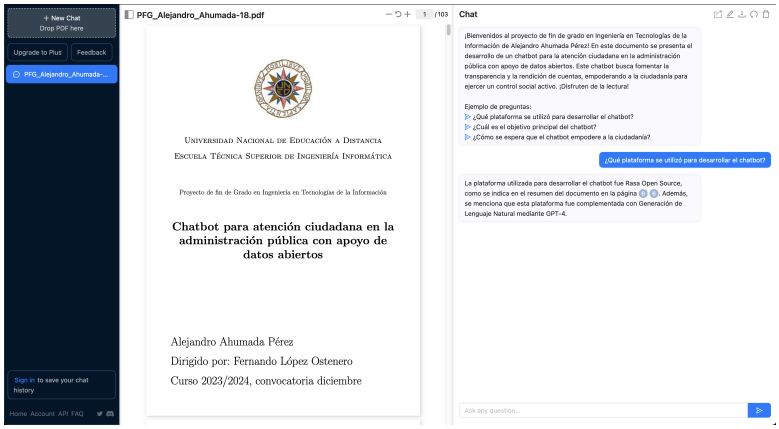


Figura 2.4: Analizando el presente documento en ChatPDF.

2.5. Aplicaciones de mensajería

Una de las razones de que las interfaces conversacionales resulten tan atractivas para el público general, es el uso habitual de aplicaciones de mensajería, que nos hace normalizar la interacción en lenguaje natural mediante chats.

Además, la mayoría de estas aplicaciones permiten integrar bots, por lo que pueden usarse como canales desde las plataformas de IA conversacional comentadas con anterioridad, sin que sea necesario desarrollar una aplicación web específica.

Telegram

Telegram[42] es una aplicación de mensajería instantánea que ofrece una amplia gama de funciones, incluidas mensajes, llamadas, grupos, canales, stories, **bots** y chats secretos. Telegram también ofrece almacenamiento en la nube ilimitado para los mensajes y archivos enviados.

Destaca por su Bot API[43] y porque no es necesario disponer de número de teléfono para crear una cuenta.

Se puede utilizar mediante aplicación web, móvil y de escritorio en Android, iOS, macOS, Windows y Linux.

[42] https://telegram.org/
[43] https://core.telegram.org/bots

WhatsApp

WhatsApp es una aplicación de mensajería instantánea que ofrece una interfaz sencilla y fácil de usar que ofrece mensajes, llamadas, grupos y estados. Su uso es muy popular en España.

Tiene la particularidad de que requiere del uso de un número de teléfono para crear una cuenta. Se puede utilizar mediante aplicación web, móvil y de escritorio en Android, iOS, macOS, Windows y Linux.

Slack

Slack[44] es una aplicación de mensajería instantánea diseñada para uso empresarial. Ofrece una amplia gama de funciones, incluidas mensajes, llamadas, canales, **bots,** integración con aplicaciones externas y API.

Se puede utilizar mediante aplicación web, móvil y de escritorio en Android, iOS, macOS, Windows y Linux.

Discord

Discord[45] es una aplicación de mensajería instantánea diseñada especialmente para el sector *gaming*. Dispone de un servidor de voz, lo que permite a los jugadores hablar entre sí mientras juegan. Discord también ofrece canales, **bots** e integración con juegos.

Está disponible como aplicación web, móvil y de escritorio en Android, iOS, macOS, Windows y Linux.

2.6. Otros proyectos similares

Aunque ya hemos citado un proyecto similar en 2.4, de forma adicional podemos destacar dos proyectos que, sin entrar en la categoría de chatbots, tienen semejanzas con algunas de las funcionalidades u objetivos de nuestro proyecto.

PandasAI

Durante el desarrollo de este Proyecto Fin de Grado, se publicó la herramienta **PandasAI**[46], que permite el análisis de conjuntos de datos mediante SmartDataframes, derivados de los

[44] https://slack.com
[45] https://discord.com/
[46] https://pandas-ai.com/

DataFrames presentes en la conocida librería de análisis de datos **Pandas**[47].

Como podemos ver, tiene objetivos similares a nuestro proyecto, aunque está orientado a su uso por parte de profesionales en tareas de ciencia de datos, por lo que requiere de conocimientos avanzados para integrar en otros proyectos y no está destinado a su uso por parte del público general.

LlamaIndex

LlamaIndex[48] es un framework de orquestación de datos para aplicaciones de grandes modelos lingüísticos (LLM).

Es un conjunto de herramientas que permite a conectar fuentes de datos para almacenar, indexar y consultar datos de forma segura y eficiente, con el fin de aumentar las capacidades de aplicaciones que hagan uso de un LLM mediante un corpus de conocimiento personalizado.

2.7. Retos

Finalmente, se plantearán diversas cuestiones a tener en cuenta durante el desarrollo del proyecto, como ética o privacidad.

Sesgo

Los LLMs se entrenan con grandes datasets de texto, esto hace que puedan reproducir estereotipos y sesgos presentes en dichos textos, lo que puede dar lugar a la generación de contenido sesgado que puede resultar dañino para la sociedad.

Privacidad

El uso de LLMs de terceros requiere que proporcionemos datos que pueden ocasionar problemas de seguridad y/o privacidad. En nuestro caso de uso, esto no resulta tan preocupante ya que los conjuntos de datos utilizados ya son (o deberían ser) públicos.

Legislación y regulación

En un contexto de creciente preocupación en torno a la necesidad de regulación de la IA, para asegurar su uso ético y responsable, existe cierta incertidumbre sobre cambios legales que,

[47] https://pandas.pydata.org/
[48] https://www.llamaindex.ai/

aunque necesarios, puedan afectar al desarrollo y viabilidad del proyecto o de las organizaciones que desarrollan los modelos utilizado.

Gobernanza

Existen dudas razonables sobre la dependencia y concentración de poder en torno a las "Big Tech" causada por los recientes LLMs. A los cambios legales antes mencionados, debemos añadir la existencia cierta incertidumbre legal, o al menos baja armonización, sobre la pertenencia de los derechos sobre los contenidos generados o sobre la responsabilidad sobre la información errónea que puedan generar.

Capítulo 3

Propuesta

Tras obtener un contexto general del ámbito de estudio y las herramientas disponibles en el capítulo 2, podemos comenzar a definir nuestra propuesta.

3.1. Funcionalidades deseadas

Nuestro fin es desarrollar un chatbot que permita a la ciudadanía en general comprender los conjuntos de datos publicados en un portal de datos abiertos, haciendo la información accesible para usuarios fuera del ámbito profesional o académico.

Para ello, nuestro chatbot debe ofrecer ciertas funcionalidades:

- **Búsqueda de conjuntos de datos.** El chatbot debe permitir consultar los datasets publicados en el portal de datos abiertos haciendo uso de su API.
- **Descarga de archivos.** Nuestro bot debe tener la capadidad de descargar y enviar al usuario los archivos (distribuciones de datos) publicados.
- **Generación de gráficas.** Una vez se disponga de un conjunto de datos, el chatbot deberá poder acceder a datos estructurados para poder generar una gráfica, guardarla en un archivo de imagen y enviarla al usuario.
- **Análisis de conjuntos de datos.** El chatbot deberá ayudar a comprender el conjunto de datos, ofreciendo diversa iformación como, por ejemplo, una descipción dinámica, datos estadísticos o, idealmente, permitir al usuario hacer una pregunta específica sobre los datos.

3.2. Componentes del chatbot

Como podemos intuir tras el análisis del capítulo anterior, nuestro chatbot precisará de diversos componentes.

3.2.1. Plataforma para IA conversacional

En primer lugar, necesitaremos una plataforma que sirva como base para la creación de nuestro chatbot.

En nuestro caso necesitaremos hacer uso de una plataforma con capacidad de NLU, en lugar de un framework para creación de bots más sencillo como pueden ser Telegraf[1] o python-telegram-bot[2].

De esta forma, podremos utilizar técnicas de NLU como la detección de intenciones o extracción de entidades que nos permitan mayor flexibilidad y la capacidad de generar conversaciones cercanas al lenguaje natural.

3.2.2. LLM

Será necesario hacer uso de un Large Language Model para realizar un análisis automático de los conjuntos de datos y tareas de generación de lenguaje natural.

Dicho LLM deberá contar con una ventana de contexto amplia, para permitir el análisis de conjuntos de datos grandes con datos estructurados en formato JSON. Además, deberá tener un buen desempeño en tareas matemáticas y de análisis de código.

3.2.3. Canal

Se propone usar una aplicación de mensajería en lugar de integrar el chatbot en el frontend de nuestro portal de datos abiertos.

Esto se realiza por varios motivos:

- El uso de este tipo de aplicaciones está muy extendido y cuentan con una gran base de usuarios.
- Están disponibles en la mayoría de dispositivos y sistemas operativos.
- La mayoría de administraciones ya utilizan estas aplicaciones para comunicarse con la ciudadanía.
- Los usuarios interactúan con este tipo de aplicaciones de forma natural. Este aprendizaje y familiaridad se trasladan de forma directa a nuestro chatbot.
- Por último, la mayoría de plataformas de IA conversacional tienen la capacidad de integrar fácilmente este tipo de aplicaciones, lo que permite ahorrar costes de desarrollo.

3.3. Planificación y estimación de costes

En el diagrama de Gantt de la figura 3.1 podemos ver el cronograma del proyecto.

[1] https://telegraf.js.org/
[2] https://docs.python-telegram-bot.org

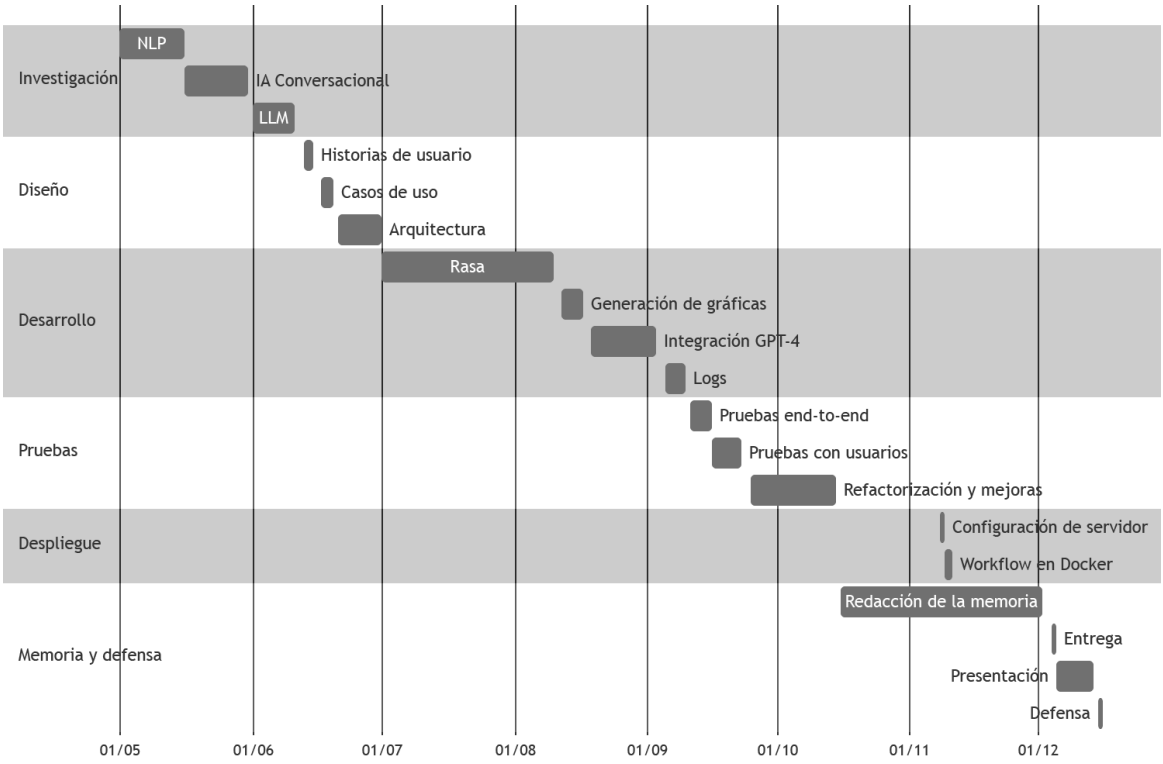


Figura 3.1: Planificación del proyecto mediante diagrama Gantt.

Una vez realizada la planificación, podemos estimar los costes de ejecución del proyecto.

En primer lugar, debemos tener en cuenta los costes de salario. Para ello, podemos calcular el coste empresa a través de la tabla salarial disponible en el convenio colectivo del sector, que se encuentra publicado en el BOE mediante la *"Resolución de 13 de julio de 2023, de la Dirección General de Trabajo, por la que se registra y publica el XVIII Convenio colectivo estatal de empresas de consultoría, tecnologías de la información y estudios de mercado y de la opinión pública"* [3].

Si revisamos la información del área 3, que compete al "Desarrollo de software, Programación y Explotación de Sistemas" para un nivel B2, que correspondería a un perfil similar al necesario para el desarrollo de este proyecto, se obtiene un salario bruto anual para 2023 de 26.614,83€. Importante destacar que el convenio plantea una subida hasta los 27.147,12€ brutos anuales para 2024, por lo que habría que ajustar el cálculo en el caso de que el desarrollo del proyecto se alargue hasta dicha fecha.

Con ello, aplicando una cotización de AT y EP del 1,65 %, obtenemos un coste empresa de 35.011,81€, que se detalla en la tabla 3.1.

Dado que el tiempo invertido ha sido de 7 meses entre investigación y desarrollo, estimando una dedicación de un tercio de la jornada aproximadamente, se hace el cálculo del porcentaje correspondiente con las pagas extra prorrateadas, dando lugar a 6.807,56€.

Dado que plazo de amortización de los equipos informáticos según el coeficiente lineal máximo establecido por la Agencia Tributaria es de 4 años, y el proyecto ha sido desarrollado en un

[3] https://www.boe.es/boe/dias/2023/07/26/pdfs/BOE-A-2023-17238.pdf

Salario bruto	26.614,83 €
Complementos	0€
Contingencias Comunes	6.281,1 €
Accidentes de Trabajo y Enfermedades Profesionales (AT y EP)	439,14 €
Desempleo	1.463,82 €
Formación Profesional	159,69 €
Fondo de garantía salarial (FOGASA)	53,23 €
Total	35.011,81 €

Tabla 3.1: Coste empresa anual de salario del empleado.

equipo de 2017, el hardware se considera amortizado.

En el caso de las licencias software, se ha utilizado Visual Studio Code, Postman, Github y Docker. El primero es un proyecto de código abierto distribuido sin coste. En el caso de Postman, GitHub y Docker, se ha utilizado el tier gratuito de las herramientas.

Para el cálculo de otros gastos generales, como electricidad, conexión a internet etc, se toma como referencia un coste mensual de 100€.

Finalmente, el coste del uso de la API de OpenAI para las pruebas, asciende a 4,16$, que equivalen a 3,84€. En el caso de AWS S3 y MongoDB Atlas, se ha utilizado la capa gratuita de ambos servicios, por lo que no han supuesto coste alguno durante el desarrollo.

En la tabla 3.2 podemos ver la estimación final de costes.

Salarios	6,807,56
Software	0€
Hardware	0€
Gastos generales	700€
Proveedores	3,84€
Total	7.511,69€

Tabla 3.2: Detalle estimación de costes.

Capítulo 4

Diseño

El presente capítulo detalla la metodología y principios seguidos para diseñar nuestro sistema conversacional, las tecnologías y herramientas utilizadas, así como la arquitectura final propuesta.

4.1. Diseñando un sistema conversacional

A la hora de diseñar un sistema conversacional, conviene tener en cuenta metodologías específicas del campo, como **Conversation-Driven Development (CDD)**.

La **Conversation-Driven Development (CDD)** es una metodología en seis fases que adopta un enfoque iterativo e interactivo para el desarrollo de asistentes conversacionales que se basa en recopilar y analizar las conversaciones de los usuarios para mejorar continuamente el asistente.

Kong y Wang [2] identifican sus fases como:

- **Compartir el bot (Share).** Publicar un prototipo del chatbot tan pronto como sea posible para descubrir interacciones no contempladas inicialmente.
- **Revisar conversaciones (Review).** Analizar conversaciones reales entre nuestro chatbot y los usuarios en diferentes fases de desarrollo. Esto nos permite descubrir funciones que no se habían descubierto con anterioridad o posibles errores.
- **Etiquetar ejemplos NLU (Annotate).** Mejorar el etiquetado de intents y entities con datos de usuarios reales.
- **Probar nuestro bot (Test).** Usando conversaciones completas como tests end-to-end. Adicionalmente, podemos añadir prácticas como Continuous integration (CI) y continuous deployment (CD).
- **Seguimiento del progreso (Track).** Analizar la tasa de éxito del bot mediante métricas objetivas. Por ejemplo, número de consultas realizadas o llamadas telefónicas para ampliar información.
- **Solucionar problemas (Fix).** Analizando tanto las conversaciones exitosas como las fallidas. Usando las primeras como nuevos casos de test y las las segundas para identificar bugs o detectar qué datos de entrenamiento nos faltan.

No es un proceso lineal, en la figura 4.1 podemos ver un ejemplo de las relaciones en una iteración completa de CDD.

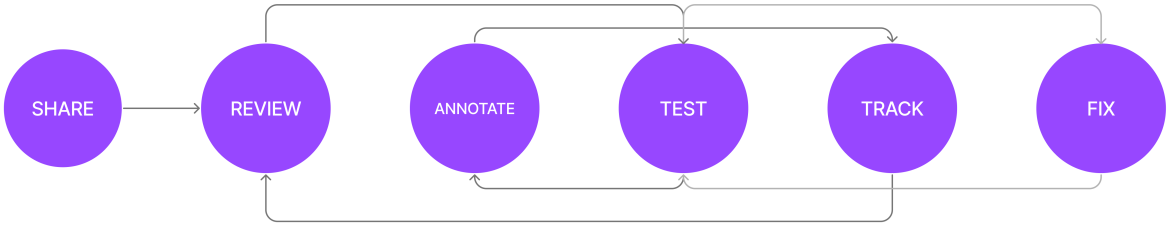


Figura 4.1: Ejemplo de una iteración completa en la metodología CDD.

Algunas recomendaciones a considerar para crear una interacción efectiva y amigable son:

- **Entender las intenciones del usuario.**
  El sistema debe identificar y entender, dentro de sus capacidades, las intenciones del usuario, respondiendo de forma acorde a las peticiones.

- **Dar instrucciones claras.**
  Las respuestas al usuario deben ser claras y concisas, dado información relevante y guiando la conversación de forma efectiva.

- **Comprensión del contexto.**
  El sistema debe mantener el contexto a lo largo de la conversación, evitando hacer peticiones redundantes, recordando interacciones pasadas y preferencias del usuario.

- **Implementar mecanismos de respaldo.**
  Debemos habilitar respuestas amistosas cuando no sea posible detectar la intención o entidades, que guíen al usuario y permitan realizar otra acción o complementar la información.

- **Preservar la privacidad y seguridad del usuario.**
  Debemos asegurar que la información personal del usuario se gestiona de forma correcta, comunicando de forma clara como es usada y/o almacenada.

- **Valorar capacidades multi-modales.**
  Por ejemplo, dar soporte a voz y texto, o permitir análisis de imágenes enviadas por el usuario.

- **Personalidad y tono.**
  La personalidad y el tono debe ser consistente en todas las respuestas del asistente. Es importante ya que esto creará una buena experiencia de usuario y favorecerá la imagen de marca.

- **Probar e iterar.**
  Esto es común a otros tipos "tradicionales" de proyectos de software, pero aquí se hace imprescindible introducir elementos del Diseño Centrado en el Usuario (DCU) y probar regularmente con usuarios reales para obtener feedback.

- **Coherencia entre canales.**
  Es importante mantener, el la medida de lo posible, la coherencia entre el tono y las acciones disponibles entre los distintos canales en los que está disponible nuestro asistente.

- **Escalabilidad.**
  Diseñar el sistema para que sea escalable. Hay que tener en cuenta que en este tipo de interacción el usuario se impacienta con mayor frecuencia y espera que las respuestas sean más rápidas. Alternativamente, si una respuesta tarda más de lo necesario, dar alguna retroalimentación al usuario que deje claro que la acción está en curso.

4.2. Requisitos del sistema

Para analizar los requisitos funcionales de nuestro proyecto, utilizaremos dos herramientas: historias de usuario y casos de uso.

4.2.1. Historias de usuario

Esta sección recopila todas las historias de usuario creadas durante el desarrollo del proyecto. Las historias de usuario nos permiten capturar los requisitos del sistema desde el punto de vista del usuario final.

- **Como** usuario, **quiero** realizar una búsqueda, **para** ver la información disponible

- **Como** usuario, **quiero** descargar el archivo de un conjunto de datos, **para** usarla en Excel

- **Como** usuario, **quiero** ver una gráfica, **para** entender los datos

- **Como** usuario, **quiero** un resumen del conjunto de datos, **para** tener una idea general de su contenido

- **Como** usuario, **quiero** datos estadísticos, **para** ampliar información sobre los datos

- **Como** usuario, **quiero** preguntar información sobre un conjunto de datos, **para** resolver dudas y ahorrar tiempo

4.2.2. Casos de uso

A partir de las historias de usuario, se elabora una serie de casos de uso.

4.2.2.1. Iniciar conversación

Actores: Usuario

Propósito: Iniciar conversación con el chatbot.

Precondiciones: El usuario debe disponer cuenta en Telegram.

Flujo principal:

1. El usuario abre la aplicación y busca el bot.
2. El usuario pulsa el botón iniciar conversación.

Postcondiciones: El bot envía el mensaje de bienvenida.

4.2.2.2. Buscar un conjunto de datos

Actores: Usuario

Propósito: Encontrar un conjunto de datos.

Precondiciones: El usuario debe disponer cuenta en Telegram y haber iniciado una conversación con el bot.

Flujo principal:

1. El usuario solicita realizar una búsqueda en lenguaje natural.
2. El bot muestra un menú con los resultados de búsqueda.
3. El usuario pulsa el botón con la opción deseada.
4. El bot muestra el menú de acciones disponibles.

Flujo alternativo:

1. El usuario solicita realizar una búsqueda den lenguaje natural pero no incluye términos de búsqueda.
2. El bot solicita los términos de búsqueda.
3. El usuario indica los términos de búsqueda en lenguaje natural.
4. El bot muestra un menú con los resultados de búsqueda.
5. El usuario pulsa el botón con la opción deseada.
6. El bot muestra el menú de acciones disponibles.

Postcondiciones: El sistema dispone del id del conjunto de datos deseado y queda a la espera de más acciones.

4.2.2.3. Descargar un archivo

Actores: Usuario

Propósito: Descargar un archivo en un formato concreto con los datos de un conjunto de datos.

Precondiciones: El usuario debe disponer cuenta en Telegram y haber iniciado una conversación con el bot.

Flujo principal:

1. El usuario solicita descargar un archivo en un formato concreto en lenguaje natural.
2. El bot envía el archivo al usuario.

Flujo alternativo:

1. El usuario solicita descargar un archivo en lenguaje natural pero no indica formato o el formato no está disponible.
2. El bot muestra un menú con los formatos disponibles.
3. El usuario pulsa el botón con la opción deseada.
4. El bot envía el archivo al usuario.

Postcondiciones: El usuario recibe el archivo en el formato deseado.

4.2.2.4. Ver una gráfica sobre los datos

Actores: Usuario

Propósito: Visualizar una gráfica con los datos de un conjunto de datos concreto.

Precondiciones: El usuario debe disponer cuenta en Telegram y haber iniciado una conversación con el bot.

Flujo principal:

1. El usuario ver una gráfica en lenguaje natural.
2. El bot envía la gráfica como imagen al usuario.

Postcondiciones: El usuario recibe el archivo de imagen con la gráfica.

4.2.2.5. Obtener un resumen de los datos

Actores: Usuario

Propósito: Obtener una descipción que resuma los contenidos del conjunto de datos.

Precondiciones: El usuario debe disponer cuenta en Telegram, haber iniciado una conversación con el bot y haber realizado una búsqueda.

Flujo principal:

1. El usuario solicita una descripción o resumen.
2. El bot responde con la descripción.

Flujo alternativo:

1. El usuario pulsa el botón "Descripción" en el menú de opciones.
2. El bot responde con la descripción.

Postcondiciones: El usuario recibe la descripción sobre el conjunto de datos.

4.2.2.6. Obtener datos estadísticos

Actores: Usuario

Propósito: Obtener una serie de datos estadisticos sobre un conjunto de datos.

Precondiciones: El usuario debe disponer cuenta en Telegram, haber iniciado una conversación con el bot y haber realizado una búsqueda.

Flujo principal:

1. El usuario solicita datos estadísticos con lenguaje natural.
2. El bot responde con los datos.

Flujo alternativo:

1. El usuario pulsa el botón "Datos estadísticos" en el menú de opciones.
2. El bot responde con los datos.

Postcondiciones: El usuario recibe los datos solicitados.

4.2.2.7. Realizar una consulta personalizada

Actores: Usuario

Propósito: Obtener información concreta sobre el conjunto de datos.

Precondiciones: El usuario debe disponer cuenta en Telegram, haber iniciado una conversación con el bot y haber realizado una búsqueda.

Flujo principal:

1. El usuario solicita realizar una consulta personalizada mediante lenguaje natural.
2. El bot responde con la información solicitada.

Flujo alternativo:

1. El usuario pulsa el botón "Consulta personalizada" en el menú de opciones.
2. El bot solicita la consulta a realizar.
3. El usuario responde con la consulta deseada en lenguaje natural.
4. El bot responde con la información solicitada.

Postcondiciones: El usuario recibe la innformación solicitada sobre el conjunto de datos.

4.3. Plataforma para IA Conversacional

Será necesario contar con una herramienta que proporciones capacidades de Procesamiento de Lenguaje Natural (PLN), o de forma más específica, Comprensión del Lenguaje Natural (CLN). Principalmente, será necesario realizar tareas de detección de *intents* y extracción de *entities*.

También será necesario que la solución elegida tenga capacidades de **Dialogue Management (DM)**, aplicando reglas y acciones que permitan determinar el flujo de la conversación.

Para ello se decide utilizar Rasa Open Source, herramienta que como su nombre indica, es de código abierto[1].

En nuestro caso de uso, Rasa Open Source ofrece ciertas ventajas frente a sus competidores:

- Es de código abierto, con licencia Apache 2.0[2] que permite su uso comercial.
- Mayor capacidad de extensión y personalización.

[1] https://github.com/RasaHQ/rasa
[2] https://www.apache.org/licenses/LICENSE-2.0.html

- Reduce la dependencia de proveedores.
- Permite la integración con canales y herramientas de diferentes proveedores.
- Control total sobre el despliegue.
- Integración en flujos CI/CD fuera del ecosistema de cada proveedor.

Además, en un ámbito ácademico, su elección aporta más interés que un producto de terceros "llave en mano", ya que permite tomar decisiones a la hora de definir la configuración de Rasa NLU, personalizando el flujo de trabajo, así como una serie de políticas, que afectarán al entrenamiento del modelo, como veremos en en 5.1

Su principal desventaja es que no dispone de capacidades de reconocimento de voz (ASR), por lo que, en principio, está limitado a bots de texto. Esto no es un requisito para nuestro proyecto en el estado actual.

4.3.1. Arquitectura general de Rasa

Como veremos más adelante en el presente capítulo, nuestra arquitectura tiene componentes adicionales, pero es interesante describir aquí los componentes principales de la arquitectura de Rasa que podemos ver en la figura 4.2.

Rasa NLU

Rasa NLU Core es un componente de aprendizaje automático que se encarga de comprender el lenguaje natural del usuario. Para ello, utiliza un modelo de lenguaje entrenado en un conjunto de datos.

Puede realizar tareas como:

- **Clasificar las intenciones del usuario:** esto significa identificar el propósito de una consulta. Por ejemplo, si un usuario dice "Quiero *descargar datos* sobre *presupuestos* en XLS", Rasa NLU Core debe identificar que la intención del usuario es descargar un archivo.
- **Extraer entidades:** esto significa identificar los objetos o conceptos que se mencionan en una consulta del usuario. Por ejemplo, en la consulta anterior, Rasa NLU Core debe identificar que las entidades mencionadas son *"presupuestos"* y *"XLS"*, y que pertenecen a los *térmidos de búsqueda* y *formato de archivo* respectivamente.

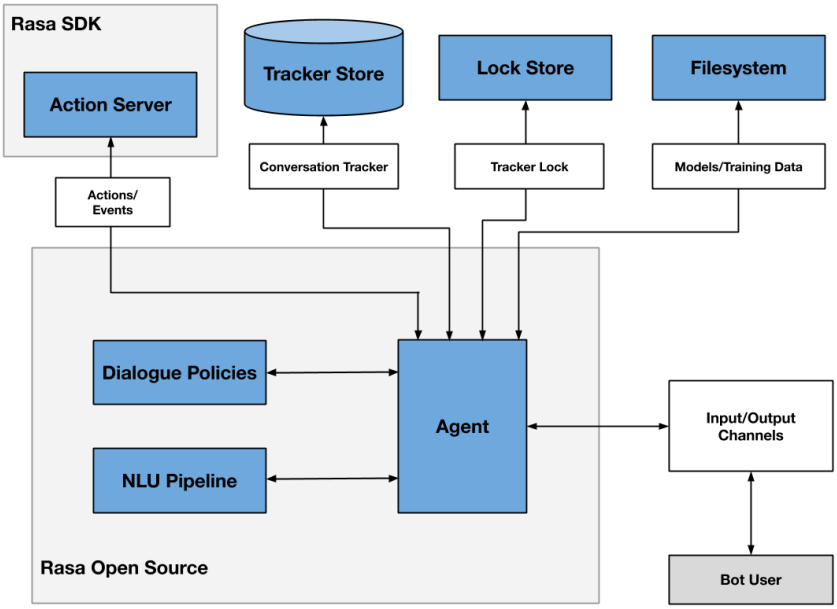


Figura 4.2: Arquitectura general de un chatbot en Rasa.

Rasa Dialogue Manager

Es el componente responsable de gestionar la conversación con el usuario mediante un modelo de planificación que tiene en cuenta el contexto: estado actual de la conversación, objetivos del usuario y reglas del sistema. En la figura 4.2 se muestra como Dialogue Policies.

Realiza las siguientes tareas:

- **Determinar la siguiente acción a realizar:** decide qué acción debe realizar el chatbot para avanzar en la conversación. Usando el ejemplo anterior, en el que el usuario dice "Quiero *descargar datos* sobre *presupuestos* en XLS", el Dialogue Manager debe determinar la siguiente acción a realizar, que podría ser "buscar un archivo en un conjunto de datos.º "pedir al usuario información adicional"si no se hubiesen identificado todad las entidades requeridas.
- **Generar respuestas:** esto significa generar respuestas que sean relevantes para la consulta del usuario, por ejemplo, que contengan la información solicitada u orienten al usuario en la conversación.
- *Mantener el estado de la conversación:* esto significa mantener un registro de lo que se ha dicho en la conversación hasta el momento, en el contexto de Rasa se utilizarán los **Slots**.

Rasa SDK Action Server

Se encarga de ejecutar las *custom actions* que se definen en Rasa. Las acciones pueden ser cosas como buscar información en un servicio web, procesar un conjunto de datos o generar un gráfico. Tendremos más información sobre esto en 5.2.

Tracker Store

Este componente que almacena el estado de la conversación en memoria o en un sistema de almacenamiento persistente. Esto permite que el chatbot continúe la conversación en una sesión posterior.

Los principales *tracker stores* son:

- InMemoryTrackerStore (por defecto),
- SQLTrackerStore (SQLite, PostgreSQL u Oracle)
- RedisTrackerStore
- MongoTrackerStore

4.3.2. Modelos spaCy

spaCy[3] es una biblioteca de software de código abierto para Procesamiento de Lenguaje Natural escrita en Python. Sirve para realizar tareas comunes de procesamiento de texto, como tokenización, lematización, reconocimiento de entidades o análisis sintáctico. Destaca por su eficiencia y velocidad en comparación con otras bibliotecas de Procesamiento de Lenguaje Natural.

En nuestro proyecto, como veremos en 5.1, utilizaremos los modelos de spaCy[4] *es_core_news_sm* o *es_core_news_md* que mejoran la detección de intents y entidades en idioma español, manteniendo una buena velocidad de respuesta y tiempo de entrenamiento.

4.4. Generación de Lenguaje Natural

Una vez resuelto el requisito anterior, se plantea la necesidad hacer uso de un LLM para tareas de Generación de Lenguaje Natural, específicamente para generación de consultas sobre conjuntos de datos.

En un primer análisis, se detecta que los modelos Llama 2, mejor candidato de entre los modelos open source disponibles, no puede ser ejecutado en nuestro equipo por limitaciones del hardware. Por ello, probamos a aplicar cuantización al modelo Llama 7B.

[3] https://spacy.io/
[4] https://spacy.io/models/es/

La cuantización se utiliza para reducir el tamaño y la complejidad de los modelos, lo que puede mejorar su rendimiento y eficiencia computacional. Utilizando la herramienta llama.cpp[5] se aplica una cuantización Q2_K al modelo Llama 2 7B, lo que nos permitirá su uso en una GPU con memoria reducida (4GB).

En la tabla 4.1 podemos ver los BPW resultantes de cada proceso de cuantización.

Cuantización	Bits por weight (BPW)	Peso total
Q2_K	3,35	2,83GB
Q3_K_M	3,91	3,30GB
Q4_K_M	4,84	4,08GB
Q5_K_M	5,68	4,78GB
Q6_K	6,56	5,53GB

Tabla 4.1: Características de modelos Llama 2 tras aplicar cuantización.

Lamentablemente, como podemos ver en la figura 4.3, el análisis y generación de tokens en entorno local es excesivamente lento ($\sim$ 2,5 tokens/s). Además, comprobamos en AWS[6] el coste del modelo de instancia EC2 más económico con GPU de NVIDIA que permitiría la ejecución de los modelos en producción, el *g4dn.xlarge,* que dispone de 4 vCPUs, 16GB de RAM y una GPU T4 Tensor Core. Esta instancia tiene un coste de 0,526$ por hora, lo que ocasionaría un coste de al menos 378,72$ mensuales, como veremos en 7.3, esto no resulta razonable a nivel económico para nuestro caso de uso.

Figura 4.3: Ejecución local del modelo Llama 2.

Para acotar el alcance del proyecto, así como adaptarnos a los recursos disponibles buscando que el proyecto sea viable económicamente, se plantea el uso de un LLM comercial. Además se plantea como requisitos que tanto la ventana de contexto sea como la capacidad de análisis de código sean amplias.

La elección de un modelo de OpenAI se justifica por las siguientes razones:

- Grandes ventanas de contexto (especialmente en el último modelo), que permiten analizar conjuntos de datos extensos.

[5] https://github.com/ggerganov/llama.cpp
[6] https://aws.amazon.com/es/ec2/pricing/on-demand/

- Aunque no ha trascendido el número de parámetros, el modelo es grande y ha sido entrenado en un dataset diverso y de gran calidad, lo que permite obtener respuestas correctas en múltiples contextos.
- Resultados de gran calidad, dada su alta capacidad en análisis de código y tareas matemáticas.
- Pago por uso con un coste razonable.
- API bien documentada.
- SDK disponible para proyectos Python y con desarrollo muy activo.

Cabe destacar, que en el inicio del proyecto, sólo estaba disponible GPT3.5 Turbo. Durante el desarrollo del mismo se decide modificar el modelo por defecto a una versión más actual por las siguientes razones:

- **De GPT-3.5 Turbo a GPT-4:** GPT-4 mejora considerablemente su capacidad de comprensión de código respecto a su antecesor, lo que repercute directamente en la calidad de las respuestas ofrecidas al realizar consultas sobre conjuntos de datos. Además la mayor ventana de contexto nos permite disponer de más tokens de entrada, por lo que se podrán analizar conjuntos de datos más extensos. Esta mejora en la experiencia justifican el incremento de costes.
- **De GPT-4 a GPT-4 Turbo:** En este caso, no hay duda posible, ya que el modelo más actual reduce su coste y además responde con más rapidez, tiene mayor ventana de contexto y ofrece mejores resultados. Aunque se encuentre en acceso previo, ya que no ha sido lanzado oficialmente, tras realizar pruebas se confirma que está en un estado apto para su uso en entornos de producción.

En la tabla 4.2 podemos ver el coste por mil tokens generado por las consultas a los principales modelos de la familia GPT de OpenAi

Modelo	Input	Output
gpt-4-1106-preview	\$0.01 / 1K tokens	\$0.03 / 1K tokens
gpt-4-32k	\$0.06 / 1K tokens	\$0.12 / 1K tokens
gpt-4	\$0.03 / 1K tokens	\$0.06 / 1K tokens
gpt-3.5-turbo	\$0.0010 / 1K tokens	\$0.0020 / 1K tokens

Tabla 4.2: Coste por tokens de entrada y salida de modelos de la familia GPT de OpenAI.

4.5. Sistema de logging

Aumque ha recibido mejoras importantes en los últimos meses, la información ofrecida por los registros de la API de OpenAI resultan insuficientes para nuestras necesidades.

Se propone crear un sistema de logs personalizado que nos permita analizar el coste de la API según la extensión del conjunto de datos, acción realizada y consulta del usuario.

4.6. Arquitectura ampliada

La figura 4.4 muestra nuestra arquitectura completa, donde añadimos nuevos elementos sobre la arquitectura general de Rasa. Todos estos elementos son utilizados por el **Action Server** para diversas tareas.

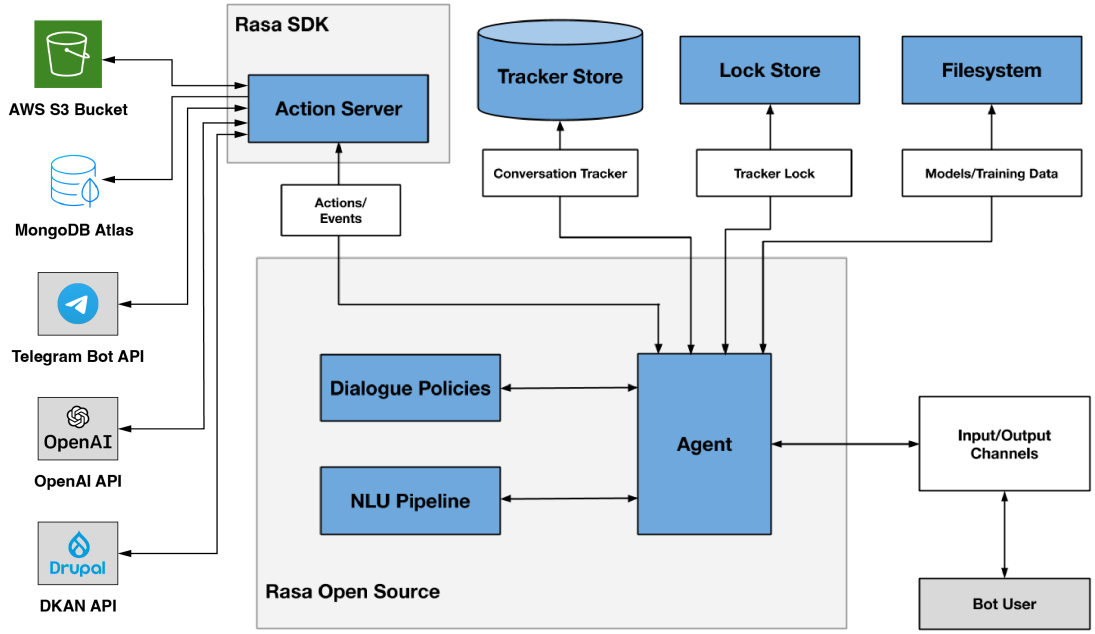


Figura 4.4: Arquitectura ampliada de nuestra aplicación.

API DKAN

DKAN es el Data Management System que utilizamos en el Portal de Datos Abiertos. Dispone de una API que hace uso de la especificación **JSON:API**[7]. Las principales ventajas del uso de esta especificación son:

- **Estructura de datos bien definida:** Esto facilita a los clientes entender cómo están organizados los datos.
- **Relaciones:** JSON API permite representar relaciones entre recursos. Esto permite a los clientes acceder a los datos relacionados de forma sencilla.
- **Metadatos:** JSON API permite añadir metadatos a los recursos. Esto puede ser útil para proporcionar información adicional sobre los datos, como su origen, su última modificación o la cantidad de objetos en el caso de que sea una lista de recursos.

[7] https://jsonapi.org/

Disponemos de más información sobre DKAN en el A.1.

Telegram

En nuestra propuesta dejamos claro que nuestro *"frontend"* debía ser una aplicación de mensajería, la elección de **Telegram** sobre el resto de opciones viene motivada por:

- Dado que nuestro público potencial es la ciudadanía general descartamos aplicaciones de mensajería de ámbito profesional u orientadas al gaming como Slack o Discord.
- Dentro de las aplicaciones de mensajería generalistas, Telegram tiene una gran base de usuarios.
- Telegram dispone de funciones avanzadas.
- Permite mayor privacidad al no requerir un número de teléfono para crear una cuenta.
- Dispone de una API para creación de bots, la **Telegram Bot API**[8], muy completa y bien documentada, lo que nos facilitará las tareas de desarrollo y nos permitirá implementar todas las funciones necesarias.
- Telegram permite integración directa con Rasa a través de sus canales[9].

Hacemos uso de la **Telegram Bot API**, tanto desde la integración en los canales de Rasa, como mediante llamadas directas al webhook.

La integración directa mediante la configuración de canales de Rasa facilita mucho el desarrollo, pero sus características no contemplan todas las funcionalidades presentes en la Bot API. Para compensar estas carencias, se hace uso del webhook de forma directa ocasionalmente.

API de OpenAI

Se realiza la conexión y peticiones a la API de OpenAI utilizando el SDK[10] disponible para Python. En concreto, se hace uso del endpoint Chat[11] para todas las tareas de Generación de Lenguaje Natural de nuestro asistente.

Object Storage en AWS S3

Para el envío a través de Telegram Bot API de archivos generados mediante acciones de nuestro Action Server, principalmente gráficas con Matplotlib[12],es necesario disponer de un servidor de archivos. Para ello, utilizamos un bucket en AWS S3.

[8] https://core.telegram.org/bots/api
[9] https://rasa.com/docs/rasa/connectors/telegram
[10] https://platform.openai.com/docs/api-reference
[11] https://platform.openai.com/docs/api-reference/chat
[12] https://matplotlib.org/

MongoDB

MongoDB[13] es un sistema de gestión de bases de datos **NoSQL** que proporciona un enfoque flexible y escalable para el almacenamiento y la recuperación de datos. Utiliza un formato de almacenamiento de documentos JSON llamado BSON (Binary JSON).

Su puesta en producción se realiza mediante el servicio DBaaS (Database as a Service)[14] MongoDB Atlas.

Este servicio es utilizado para nuestro sistema de registros. La elección principal de una solución NoSQL sobre una base de datos relacional es la flexibilidad que ofrece NoSQL, en concreto MongoDB, para modificar la estructura de los datos, lo que facilitará bastante el cambio a otro LLM en un futuro.

Se dispone de información detallada sobre el proceso de alta y configuración de estos servicios de terceros en el anexo B.

[13] https://www.mongodb.com
[14] https://www.mongodb.com/database-as-a-service

Capítulo 5

Desarrollo del proyecto

En este capítulo detallamos todo el proceso de desarrollo de nuestro chatbot: la configuración de nuestro flujo y políticas de entrenamiento, nuestro dominio, definición de datos de entrenamiento e integración con servicios de terceros mediante los SDKs proporcionados por los proveedores.

5.1. Configuración de Rasa NLU

El primer paso es configurar los componentes de nuestro pipeline y las políticas que controlarán el entrenamiento de nuestro modelo.

5.1.1. Pipeline

Rasa permite definir un flujo de trabajo con diferentes componentes que se ejecutarán de forma secuencial para transformar la entrada del usuario en una salida estructurada.

Estos componentes principalemente son los encargados de realizar clasificación de intención (intent), extracción de entidades (entitites), selección de respuesta (utterances o acciones), preprocesado.

Aunque hay disponibles más tipos de componentes, detallamos utilizados en nuestro proyecto.

SpacyNLP

Componente que carga el modelo e inicializa las estructuras requeridas por spaCy. Debe ser el primer componente en el pipeline.

SpacyTokenizer

Tokenizador requerido al usar modelos spaCy. Un tokenizador es una herramienta que divide el texto en unidades más pequeñas, como palabras o subpalabras. Es un paso fundamental en las tareas de Procesamiento del Lenguaje Natural.

SpacyFeaturizer

Caracterizador requerido al usar modelos spaCy. Un caracterizador o extractor de características es una función o módulo que transforma los datos sin procesar en un formato que puede ser utilizado por los algoritmos de aprendizaje automático. Esto suele implicar crear representaciones numéricas de datos categóricos, extraer y codificar características del texto y normalizar datos numéricos.

RegexFeaturizer

Componente para realizar extracción de entidades y clasificación de intenciones mediante expresiones regulares. Utilizado por Rasa, entre otras cosas, para los lookups definidos en nuestos datos de entrenamiento. Requiere usar DIETClassifier con posterioridad en el pipeline.

LexicalSyntacticFeaturizer

Crea características para extracción de entidades utilizando la técnica de *"sliding window"* sobre los tokens del mensaje del usuario.

CountVectorsFeaturizer

Utiliza la técnica "bag-of-words" para crear una representación numérica de los mensajes del usuario, intenciones y respuestas.

DIETClassifier

El clasificador **Dual Intent Entity Transformer (DIET)** tiene una arquitectura multitarea y se utiliza para clasificación de intenciones y extracción de entidades.

EntitySynonymMapper

Se utiliza para mapear entidades que sean sinónimos al mismo valor.

ResponseSelector

Utilizado por el Dialogue Manager, predice la respuesta a partir de una lista de candidatas.

FallbackClassifier

Clasifica los mensajes no reconocidos con la etiqueta *nlu_fallback*. Mantiene un umbral para determinar si la intención o entidad no ha sido reconocida en función de los valores obtenidos por los clasificadores anteriores.

Configuración final

A continuación se muestra la configuración utilizada:

```
pipeline:
    - name: SpacyNLP
        model: "es_core_news_sm" #se puede utilizar es_core_news_md si
            nuestro proyecto lo requiere
    - name: SpacyTokenizer
    - name: SpacyFeaturizer
    - name: RegexFeaturizer
    - name: LexicalSyntacticFeaturizer
    - name: CountVectorsFeaturizer
    - name: CountVectorsFeaturizer
        analyzer: char_wb
        min_ngram: 1
        max_ngram: 4
    - name: DIETClassifier
        epochs: 100
        constrain_similarities: true
    - name: EntitySynonymMapper
    - name: ResponseSelector
        epochs: 100
        constrain_similarities: true
    - name: FallbackClassifier
        threshold: 0.3
        ambiguity_threshold: 0.1
```

5.1.2. Policies

Las políticas son usadas por el chatbot para decidir qué acción tomar en cada paso de la conversación. Las podemos dividir en dos tipos: basadas en Aprendizaje Automático y basadas en reglas.

Como se muestra en la tabla 5.1, existe un nivel de prioridad para las diferentes políticas , que se utiliza para desempatar en el caso de que dos políticas sean elegidas con iguarl confianza:

Política	Valor
RulePolicy	6
MemoizationPolicy	3
AugmentedMemoizationPolicy	3
UnexpecTEDIntentPolicy	2
TEDPolicy	1

Tabla 5.1: Prioridad para las diferentes políticas de Rasa.

RulePolicy

Es una política basada en reglas. Se utiliza para aplicar reglas más estrictas de lógica de negocio utilizando las *reglas* definidas en los datos de entrenamiento.

MemoizationPolicy

Es una política de tipo ML. Se utiliza para predecir la siguiente acción de forma menos estricta que con las reglas utilizando las *stories* definidas en los datos de entrenamiento.

UnexpecTEDIntentPolicy

Es una política de tipo ML. Es una política auxiliar que permite responder a intenciones inesperadas, por lo que debe utilizarse con al menos otra política, normalmente, TEDPolicy.

Permite configuración específica, siendo las más relevante:

- **epochs:** es el número de pases (veces que el algoritmo verá los datos) de entrenamiento. Lo configuramos como 200, lo que nos da un buen resultado sin alargar demasiado el tiempo de entrenamiento.
- **max_history:** Configura la cantidad de historia de la conversación que el modelo revisa a la hora de tomar una acción.

TEDPolicy

Es una política de tipo ML. **Transformer Embedding Dialogue (TED)** es una política multitarea para la predicción de la próxima acción y el reconocimiento de entidades. Su arquitectura consta de varios transformadores.

Permite configuración específica, siendo las más relevante:

- **epochs:** es el número de pases (veces que el algoritmo verá los datos) de entrenamiento. Lo configuramos como 200, lo que nos da un buen resultado sin alargar demasiado el tiempo de entrenamiento.

- **max_history:** Configura la cantidad de historia de la conversación que el modelo revisa a la hora de tomar una acción.

- **constrain_similarities:** Aplica *sigmoid cross entropy loss,* función que mide la diferencia entre la predicción del modelo y el valor real, ajustando los parámetros del modelo para reducir la pérdida.

Configuración final

A continuación se muestra la configuración de policies.

```
policies:
  - name: MemoizationPolicy
  - name: RulePolicy
  - name: UnexpecTEDIntentPolicy
    max_history: 5
    epochs: 100
  - name: TEDPolicy
    max_history: 5 # cuidado con este parametro, nos permite realizar
       conversaciones mas complejas pero aumenta mucho el tiempo de
       procesado
    epochs: 200
    batch_size: 50
    max_training_samples: 300
    constrain_similarities: true
```

5.2. Elementos de dominio

Una vez configurado nuestro pipeline y políticas de entrenamiento, debemos pasar a definir nuestro dominio. El dominio es el contexto general del chatbot, es decir los elementos y objetivos que componen el mundo que rodea al chatbot.

Para identificar nuestros elementos de dominio, es esencial el uso de herramientas como las historias de usuario o los casos de uso detalladas en 4.2.

Todo esto se define en un único archivo llamado *domain.yml* ubicado en la raíz de nuestro proyecto.

Intents

Aunque ya lo definimos de forma general en 2.1, aquí lo revisamos en el contexto de Rasa. Un *intent* es una unidad de significado que representa una intención por parte del usuario que normalmente tendrá una respuesta (acción o utterance) por parte de nuestro chatbot. Como veremos posteriormente, en el dominio solo lo nombraremos y posteriormente pondremos ejemplos en los datos de entrenamiento.

Inicialmente los definimos en *domain.yml*, en nuestro caso particular obtenemos:

```
intents:
  - start_telegram
  - greet
  - goodbye
  - thanks
  - affirm
  - deny
  - start_over
  - bot_challenge
  - search_data
  - plot_data
  - download_data
  - explain_data
  - statistics_data
  - custom_query_data
```

Aunque no lo usamos en nuestro código, se permite asociar o excluir **entidades** de forma explícita a un *intent* mediante la siguiente sintaxis:

```
intents:
    - search_data
        use_entities:
            - search_terms
    - plot_data
        ignore_entities:
            - file_format
```

Entities

En el contexto de Rasa, las entidades son tipos de datos que el chatbot puede entender y son extraídos por el *entity extractor*. Pueden ser de tipo simple, como un nombre o una dirección, o de tipo complejo, como una fecha o una hora.

```
entities:
  - username
  - data_search_terms
  - dataset_uuid
```

```
  - data_file_format
  - data_custom_query
```

Adicionalmente, aunque no se utilice en nuestro proyecto, podemos definir roles y grupos para nuestras entidades, lo que nos permite asignar contextos adicionales en nuestros datos de entrenamiento, por ejemplo, si un formato de archivo se refiere a un formato de entrada o un formato de salida.

```
entities:
   - data_file_format:
       roles:
       - input
       - output
```

Cabe destacar que las entidades permiten definir un elemento que el sistema reconoce y por ello puede influenciar la conversación afectando a la siguiente acción elegida (a menos de que se haya excluido de forma explícita de un intent como vimos anteriormente). Pero no funcionan como una variable, por no que el sistema no guarda su valor para uso posterior una vez lo ha interpretado. Para ello, debemos usar slots.

Slots

Los **slots** son variables que se utilizan para almacenar información durante una conversación. Funcionan como un almacén de clave-valor que se puede usar tanto para almacenar información proporcionada por el el usuario como información generada durante una acción.

Al definir un **slot** podemos incluir información adicional, como su tipo (texto, booleano, categoría, float, lista o any), si debe influenciar la conversación o desde dónde se pueden mapear.

A continuación se muestra la definición del **slot** que almacena el formato de archivo a descargar:

```
slots:
    data_file_format:
    type: text
    influence_conversation: true
    mappings:
      - type: from_entity
        entity: data_file_format
        intent: download_data
      - type: from_text
        conditions:
          - active_loop: data_download_form
            requested_slot: data_file_format
```

Además, aunque esta característica no se utiliza en nuestro proyecto, se pueden definir tipos personalizados de slots añadiendo una clase de Python en la carpeta */addons*.

```
from rasa.shared.core.slots import Slot

class NombreDeSlot(Slot):
  # definición del slot
```

Responses

Las respuestas (o utterances) son un tipo sencillo de acciones, que solo contienen un mensaje que el bot envía al usuario. Su nombre debe empezar por *utter_* y se pueden personalizar definiendo varias variantes para una misma respuesta o incluyendo información almacenada en los slots.

A continuación se muestra una respuesta al saludo del usuario, que elige una variante condicionalmente si el slot *username* contiene información o no.

```
responses:
    utter_greet:
    - condition:
        - type: slot
          name: username
          value: null
      text: "Hola! Qué tal estás?"
    - condition:
        - type: slot
          name: username
          value: null
      text: "Hey! Qué tal estás?"
    - text: "Hola {username}! Qué tal estás?"
    - text: "Hola {username}! Cómo va tu día?"
```

Actions

El segundo tipo de respuestas son las acciones personalizadas, que nos permiten ejecutar cualquier código, para por ejemplo, hacer consultas a una API, guardar información en una base de datos, etc.

Se deben definir en nuestro archivo de dominio y como una clase Python en el archivo */actions/actions.py*. A continuación mostramos un ejemplo de a acción que ejecuta una consulta personalizada sobre el conjunto de datos utilizando la API de OpenAI.

Definimos la acción en nuestro dominio:

```
actions:
    - action_custom_query_data
```

A continuación creamos la clase con las acciones a ejecutar:

```
class ActionCustomQueryData(Action):
    def name(self) -> Text:
        return "action_custom_query_data"

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict[Text, Any]]:
        # Get conversation id
        conversation_id = tracker.sender_id
        message_id = tracker.get_slot("menu_message_id")

        if message_id is not None:
            clear_menu(conversation_id, message_id, dispatcher, self.
               name())

        data = tracker.get_slot("data")
        query = tracker.get_slot("data_custom_query")

        prompt = f"""
        I need you to answer a question in spanish about the data in
           json delimited by triple backticks:
        The question is: {query}

        ```{data["results"]}```
        """

        try:
            response = get_completion(
                conversation_id=conversation_id,
                rasa_action="action_custom_query_data",
                prompt=prompt,
            )
            print(response)
            dispatcher.utter_message(text=response)
        except Exception as e:
            dispatcher.utter_message(
                text="Ha ocurrido un error al evaluar los datos, el
                   conjunto de datos es demasiado grande."
            )

        return [SlotSet("data_custom_query", None), SlotSet("
           menu_message_id", None)]
```
```

Forms

Los **formularios** son un tipo especial de acciones que ayudan a recolectar información de un usuario. Requieren la inclusión del componente **RulePolicy** en las políticas de nuestro chatbot. Los formularios se activan y desactivan mediante rules y stories.

A continuación se muestra el formulario que permite solicitar al usuario el formato de archivo deseado.

```
forms:
  data_download_form:
    required_slots:
      - data_file_format
```

Adicionalmente se puede añadir la palabra clave *ignored_intents* para que ciertos intents no disparen el formulario.

Hay un tipo especial de respuestas y acciones, que deben tener la nomenclatura *utter_ask_nombre_slot* o *action_ask_nombre_slot*, que son enviadas como mensaje al usuario en la solicitud de información al disparar un formulario.

```
actions:
    - utter_ask_data_search_terms
responses:
    utter_ask_data_search_terms:
      - text: "Qué tema te interesa?"
      - text: "Qué datos quieres?"
      - text: "Qué quieres que busque?"
```

Validaciones

Otro tipo especial de acciones son las validaciones, que van asociadas a un formulario. Requieren que su nombre sea del tipo *validate_nombre_form* y deben definirse en la sección de acciones del dominio, así como en nuestro archivo *actions.py*.

En primer lugar, lo definimos en nuestro dominio:

```
actions:
  - validate_data_search_terms_form
```

A continuación, definimos la acción a realizar por el validador:

```
class ValidateDataDownloadForm(FormValidationAction):
    def name(self) -> Text:
        return "validate_data_download_form"
```

```
    def validate_data_file_format(
        self,
        slot_value: Any,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: DomainDict,
    ) -> Dict[Text, Any]:
        """Validate `data_file_format` value."""

        data_meta = tracker.get_slot("data_meta")

        allowed_file_formats_dynamic = []

        for distribution in data_meta["distribution"]:
            allowed_file_formats_dynamic.append(distribution["format"].
               lower())

        # Check and normalize file format
        if slot_value.replace(".", "").lower() not in
           allowed_file_formats_dynamic:
            dispatcher.utter_message(text="El formato no es valido.")
            return {"data_file_format": None}

        return {"data_file_format": slot_value.replace(".", "").lower()
           }
```

5.3. Datos y elementos para el entrenamiento

Tras determinar nuestro dominio, debemos definir nuestros datos de entrenamiento, cuyo objetivo es ayudar a que nuestro modelo entienda y extraiga información estructurada de los mensajes del usuario en lenguaje natural.

Se dividirán en tres archivos situados en la carpeta */data*:

- **rules.yml** para las reglas
- **stories.yml** para las historias
- **nlu.yml** para el resto de datos de entrenamiento: intents, entities, regex, lookups y sinónimos.

Intents y entities

En 5.2 definimos los intents y entities en el dominio de nuestra aplicación, solo resta ingresar ejemplos que permitan entrenar nuestro modelo de forma que reconozca dichos elementos. En los datos de entrenamiento las entidades se definen siempre en el contexto de un intent.

Dado que el archivo que incluye los datos para NLU son extensos, se muestra un ejemplo parcial de unos de nuestros intents *(download_ data)*:

```
nlu:
    - intent: download_data
    examples: |
      - /download_data
      - /download_data [CSV](data_file_format)
      - /download_data [.CSV](data_file_format) [deuda viva](
         data_search_terms)
      - Descargar datos
      - Descargar datos en [CSV](data_file_format)
      - Necesito descargar datos en [XLS](data_file_format)
      - Busco datos en [PDF](data_file_format) sobre [deuda viva](
         data_search_terms)
      - Tienes datos sobre [presupuestos](data_search_terms) en [.xlsx
         ](data_file_format)?
      - Quiero descargar datos sobre [residuos urbanos](
         data_search_terms) en [csv](data_file_format)
      - Quiero datos sobre [residuos](data_search_terms) en [csv](
         data_file_format)
```

Podemos definir nuestras entidades con la sintaxis *[palabra](nombre_ intent)*.

Regex

Puede utilizar expresiones regulares para mejorar la clasificación de intenciones y la extracción de entidades. Para ello, será necesario definir respectivamente **RegexFeaturizer** y **RegexEntityExtractor** en nuestra pipeline.

En nuestro caso, definimos un regex para ayudar a identificar la entidad *dataset_ uuid*.

```
nlu:
    - regex: dataset_uuid
    examples: |
      - \b[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}\
         b
```

Synonyms

Aunque los sinónimos no se utilizan en nuestro proyecto, principalmente porque el modelo spaCy suple dicha funcionalidad, por una cuestión de completitud conviene describirlos en este apartado.

Permite, como su nombre indica, definir sinónimos para nuestras entidades, complementando así *pipelines* de entrenamiento más simples. Lo consiguen mapeando las entidades extraídas

a un valor diferente del texto literal extraído, además no tiene en cuenta las mayúsculas y minúsculas.

```
nlu:
    - synonym: formato
      examples: |
        - formato de archivo
        - tipo de archivo
        - extensión de archivo
```

Lookups

Un lookup es una tabla de datos que se utiliza para asignar valores a entidades, permitiendo que el chatbot entienda que varios términos se refieren a una entidad concreta sin tener que definirlos de forma más verbosa en los ejemplos de *intents* de los datos de entrenamiento.

Para ello, se tratan como patrones de expresiones regulares no sensibles a las mayúsculas, por lo que requieren incluir **RegexFeaturizer** y **RegexEntityExtractor** en nuestro pipeline.

Aquí se muestra un extracto del lookup utilizado para los formatos de archivo:

```
nlu:
    - lookup: data_file_format
    examples: |
      - csv
      - .csv
      - CSV
      - .CSV
      - xls
      - .xls
      - xlsx
      - .xlsx
      - .XLSX
      - json
      - .json
      - .JSON
      - pdf
      - .pdf
      - .PDF
      - geojson
      - .geojson
      - .GEOJSON
      - arcgis
      - .arcgis
      - .ARCGIS
      - kml
      - .kml
      - xml
```

```
28        - .xml
29        - tsv
30        - .tsv
```

Rules

Legamos al primer tipo de dato de entrenamiento referente al *dialogue manager*. La reglas describen fragmentos pequeños de conversaciones que deben seguir un camino estricto. Se puede utilizar para, por ejemplo, responder a interrupciones urgentes del usuario.

Para utilizar reglas debemos incluir el componente **RulePolicy** en nuestras *policies*.

Aquí podemos ver un ejemplo para disparar el mensaje de bienvenida cuando el usuario inicia la conversación en Telegram con nuestro chatbot.

```
1 rules:
2   - rule: Telegram start command
3     steps:
4       - intent: start_telegram
5       - action: utter_welcome
```

Las reglas deber ser utilizadas solo cuando sean imprescindibles, siendo las **stories** nuestro elemento preferente para entrenamiento del *dialogue manager*.

Stories

Las **stories** son el segundo tipo de dato de entrenamiento que nos permitirán entrenar al *dialogue manager*. Son **stories** menos estrictas que las **reglas**, pero a su vez más potentes, ya que permiten generalizar caminos de conversación que no se hayan definido de forma explícita.

Las **stories** se representan como una conversación entre el usuario y el chatbot, expresando las interacciones del usuario como *intents* y las respuestas del chatbot como *acciones*.

Además, pueden contener eventos, que sirven principalmente para activar *formularios* y rellenar *slots*.

A continuación se muestra una historia para el intent "search_data" que activa el formulario adecuado y muestra el menú al usuario.

```
1 stories:
2   - story: User requests to search data
3     steps:
4       - intent: search_data
5       - action: data_search_terms_form
6       - active_loop: data_search_terms_form
7       - slot_was_set:
```

```
8           - requested_slot: null
9       - active_loop: null
10      - action: data_select_dataset_form
11      - active_loop: data_select_dataset_form
12      - slot_was_set:
13          - requested_slot: null
14      - active_loop: null
15      - action: action_search_data
16      - action: utter_anything_else
17      - action: action_show_menu
```

5.4. Entrenamiento de modelos

Una vez definidos todos los componentes, podemos proceder a entrenar nuestros modelos, para lo que disponemos de una serie de comandos CLI.

El más importante, nos permite entrenar nuestro modelo con los datos disponibles, se guardará automáticamente en la carpeta */models* al finalizar el proceso de entrenamiento.

```
1 rasa train
```

También podemos iniciar una sesión interactiva que nos permite generar nuevos datos para el entrenamiento del modelo sin tener que añadir los mismos manualmente a los diferentes archivos del proyecto.

```
1 rasa interactive
```

Este comando inicia una conversación en el terminal con el último modelo entrenado. No genera nuevos datos como el comando anterior y solo sirve para probar nuestro modelo.

```
1 rasa shell
```

5.5. Integración con servicios de terceros mediante SDKs

Durante el desarrollo del proyecto, se han utilizado diversos SDK para facilitar la integración con AWS S3, MongoDB Atlas y la API de OpenAI. En el caso de Telegram, hemos utilizado la integración mediante canales de Rasa, así como peticiones directas al webhook.

5.5.1. OpenAI

OpenAI dispone de los librerías[1] para Python y Node.js. En el caso de Python, incluye tipos que facilitan mucho el desarrollo y detección de bugs.

Aunque Rasa incluye integración con LLMs[2] en su rama experimental, esta no está indicada aún para entornos de producción y está limitada a la versión Rasa Plus. Además, el uso de la librería proporcionada por OpenAI, nos ofrece un mayor control sobre la configuración de las peticiones a su API completions.

A continuación, se muestra un ejemplo de uso de la librería en nuestro proyecto:

```
response: ChatCompletion = client.chat.completions.create(
    model=model,  # this is the model that the API will use to generate
        the response
    messages=[
        {"role": "user", "content": prompt}
    ],  # this is the prompt that the model will complete
    temperature=0.5,  # this is the degree of randomness of the model's
        output
    max_tokens=2000,  # this is the maximum number of tokens that the
       model can generate
    top_p=1,  # this is the probability that the model will generate a
       token that is in the top p tokens
    frequency_penalty=0,  # this is the degree to which the model will
       avoid repeating the same line
    presence_penalty=0,  # this is the degree to which the model will
       avoid generating offensive language
)
```

5.5.2. AWS S3

Boto3[3] nos permite crear, configurar y gestionar los servicios de AWS. En nuestro proyecto, utilizamos esta librería en la acción *"action_plot_data"* para publicar en nuestro bucket las gráficas generadas en formato de imagen, para su posterior envío al usuario vía Telegram.

Existe un bug conocido[4] al utilizar las regiones lanzadas después de 2019, que se soluciona incluyendo el argumento *endpoint_url* de forma obligatoria.

```
s3_client = boto3.client(
            "s3",
            region_name=os.getenv("AWS_REGION"),
```

[1] https://platform.openai.com/docs/libraries
[2] https://rasa.com/docs/rasa/next/llms/large-language-models
[3] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[4] https://github.com/boto/boto3/issues/2864

```
            endpoint_url=f"https://s3.{os.getenv('AWS_REGION')}.
                amazonaws.com",
            aws_access_key_id=os.getenv("AWS_ACCESS_KEY_ID"),
            aws_secret_access_key=os.getenv("AWS_SECRET_ACCESS_KEY"),
        )
```

Como detallamos en el anexo C.2, para garantizar la privacidad y seguridad de nuestro bucket, se aplica una política de permisos restrictiva, prohibiendo el acceso público al mismo. Por ello, es necesario generar una URL prefirmada de AWS S3 para su envío al usuario como archivo de imagen a través de Telegram.

```
# Get a presigned url to avoid public access
    presigned_image_url = s3_client.generate_presigned_url(
        "get_object",
        Params={
            "Bucket": f"{os.getenv('S3_BUCKET_NAME')}",
            "Key": f"{conversation_id}.png",
        },
        ExpiresIn=3600,
    )
    dispatcher.utter_message(
        image=presigned_image_url,
    )
```

5.5.3. MongoDB

Utilizamos PyMongo[5] para enviar logs a nuestra colección en MongoDB Atlas.

Se hace uso de esta librería en las acciones *"action_explain_data"* y *"action_statistics_data"* y *"action_custom_query_data"*.

```
# Log completion to MongoDB
try:
    mongo_client.admin.command("ping")
    print("Pinged your deployment. You successfully connected to
        MongoDB!")
    db = mongo_client.get_database("logs")
    collection = db.get_collection("completions")
    document = {
        "created": response.created,
        "conversation_id": conversation_id,
        "model": response.model,
        "rasa_action": rasa_action,
        "prompt": prompt,
        "usage": {
            "prompt_tokens": response.usage.prompt_tokens,
```

[5] https://pymongo.readthedocs.io/en/stable/index.html

```
15              "completion_tokens": response.usage.completion_tokens,
16              "total_tokens": response.usage.total_tokens,
17              "cost": get_openai_token_cost_for_model(
18                  model_name=model, num_tokens=response.usage.
                       total_tokens
19              ),
20          },
21      }
22      collection.insert_one(document)
23  except Exception as e:
24      print(e)
```

Capítulo 6

Pruebas

Aquí detallamos las pruebas realizadas, que se dividirán en tres tipos: validación, pruebas end-to-end y pruebas con usuarios finales.

6.1. Validación

Aunque no es una prueba en el sentido estricto, Rasa permite detectar errores validando los datos de dominio, NLU e historias. mediante el comando *validate*, que es conveniente ejecutar cuando se realicen cambios significativos en los archivos que contienen dicha información o como primera herramienta de debug.

Es importante señalar que el comando validate no comprueba incoherencias entre *stories* y *rules*, pero el componente **RulePolicy** realizará una comprobación antes de iniciar el entrenamiento.

```
rasa data validate
```

6.2. Pruebas end-to-end

Podemos realizar las pruebas definidas en la carpeta */tests* mediante el comando:

```
rasa test
```

Este comando produce como salida hasta tres tipos de archivos, para cada uno de los elementos o componentes sobre los que se realizan las pruebas (normalmente DIETClassifier, intents, stories y TEDPolicy).

- un informe *(componente_ report.json)*,
- una matriz de confusión *(componente_ confusion_ matrix.png)*,
- un histograma *(componente_ histogram.png)*.

6.2.1. Informe

El informe muestra la **precisión, recall** y **f1-score** para cada intent, además de datos generales.

La **precisión** es el ratio $t_p/(t_p + f_p)$, donde t_p es el número de verdaderos positivos y f_p es el número de falsos positivos. Señala la habilidad del clasificador de no etiquetar como positiva una muestra que es negativa.

La puntuación **recall** es el ratio $t_p/(t_p + f_n)$, donde t_p es el número de verdaderos positivos y f_n es el número de falsos negativos. Mide la habilidad del clasificador de encontrar muestras positivas.

Finalmente **f1-score** es la media armónica de la precisión y el recall, se calcula mediante: $F1 = 2 * (precision * recall)/(precision + recall)$

A continuación se muestra un ejemplo parcial de informe para el componente DIETClassifier:


```
{
  "search_terms": {
    "precision": 1.0,
    "recall": 0.8,
    "f1-score": 0.888888888888889,
    "support": 10,
    "confused_with": {}
  },
  "file_format": {
    "precision": 1.0,
    "recall": 1.0,
    "f1-score": 1.0,
    "support": 2,
    "confused_with": {}
  },
  "username": {
    "precision": 1.0,
    "recall": 1.0,
    "f1-score": 1.0,
    "support": 3,
    "confused_with": {}
  },
  "micro avg": {
    "precision": 1.0,
    "recall": 0.8666666666666667,
    "f1-score": 0.9285714285714286,
    "support": 15
  },
  "macro avg": {
    "precision": 1.0,
    "recall": 0.9333333333333332,
    "f1-score": 0.9629629629629629,
    "support": 15
  },
```

```
  "weighted avg": {
    "precision": 1.0,
    "recall": 0.8666666666666667,
    "f1-score": 0.9259259259259259,
    "support": 15
  },
  "accuracy": 0.9944289693593314
}
```


6.2.2. Matriz de confusión

Una matriz de confusión es una tabla que se utiliza para evaluar el rendimiento de un modelo de aprendizaje automático en tareas de clasificación. La matriz de confusión se construye comparando las predicciones del modelo con las etiquetas reales de los datos de prueba.

La matriz de confusión muestra qué elementos, como intents o entities, han sido confundidos con otros. La prueba guarda información adicional sobre el etiquetado erróneo en un archivo con nombre *componente_ errors.json*.

La figura 6.1 muestra un ejemplo de prueba de nuestro proyecto, en el que todos los intents han sido identificados correctamente.

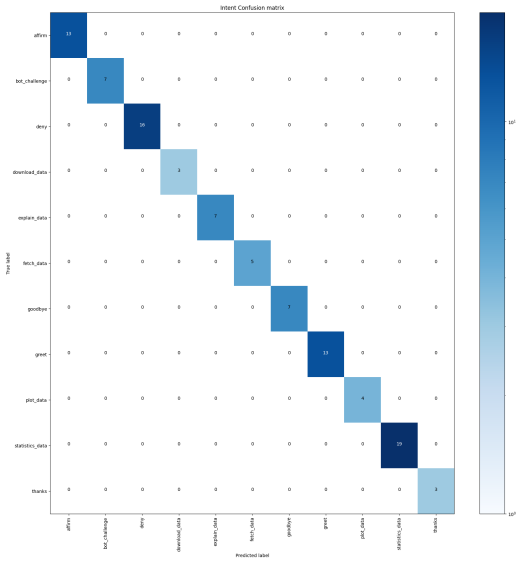


Figura 6.1: Ejemplo de matriz de confusión de intents

6.2.3. Histograma

Un histograma es una representación gráfica de una distribución de frecuencia. Se utiliza para mostrar la distribución de los valores de una variable continua, en nuestro caso, la puntuación de confianza.

El histograma de la figura 6.2 muestra la puntuación de confianza de todas las predicciones. Los resultados negativos tienen color rojo y los positivos color verde, por lo que podemos ver que la prueba ha finalizado con éxito.

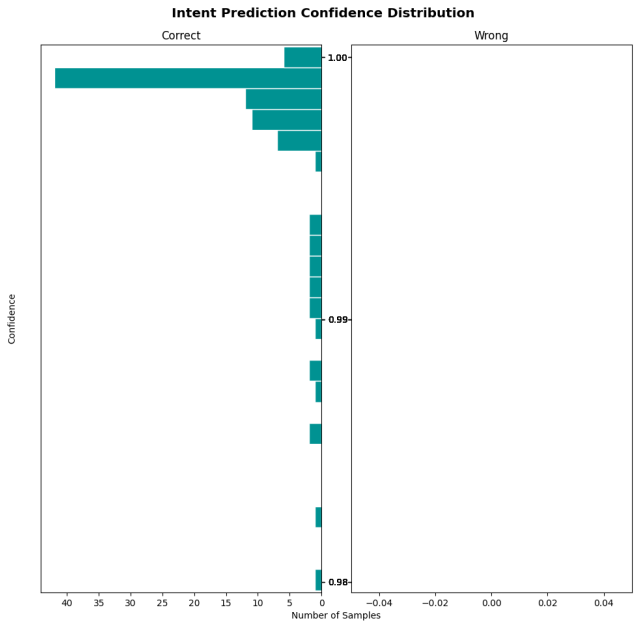


Figura 6.2: Ejemplo de histograma de intents

6.3. Pruebas con usuarios

Durante el desarrollo del proyecto, se realizan diferentes entrevistas y pruebas con usuarios finales, para detectar problemas de experiencia de usuario (UX) aplicando principios del Diseño Centrado en el Usuario (DCU).

Los perfiles de usuario presentes en las pruebas fueron:

- **User A**
  Mujer, aproximadamente 35 años, estudios superiores, aptitudes digitales avanzadas.
- **User B**
  Hombre, aproximadamente 50 años, estudios medios, aptitudes digitales medio-bajas.
- **User C**
  Hombre, aproximadamente 18 años, cursando estudios superiores, aptitudes digitales altas.
- **User D**
  Mujer, aproximadamente 65 años, estudios superiores, aptitudes digitales medias.
- **User E**
  Mujer, aproximadamente 80 años, sin estudios, aptitudes digitales bajas, problemas de concentración y movilidad.

Dichas pruebas nos permiten detectar problemas en las rutas disponibles en las historias y puntos de confusión para los diferentes usuarios, lo que nos permite añadir dos funcionalidades no contempladas inicialmente: menú de selección de conjunto de datos y validación de formatos disponibles para las descargas.

Capítulo 7

Conclusiones y trabajos futuros

Este capítulo se dedica a realizar una retrospectiva sobre el trabajo realizado y sobre las mejoras propuestas para dar continuidad al proyecto. Además se incluye una estimación del coste de mantenimiento anual.

7.1. Conclusiones

La totalidad de los objetivos y funcionalidades proyectadas inicialmente han sido desarrolladas, incluyendo en las últimas semanas alguna mejora sobre el planteamiento inicial del proyecto, como el despliegue mediante Docker, que facilita enormemente su puesta en producción.

Durante el desarrollo del proyecto, tuve la oportunidad de participar en el **Rasa Community Challenge,** donde presenté un prototipo preliminar que fue seleccionado. Aunque finalmente no quedó entre los proyectos finalistas, esta experiencia me sirvió para ver qué funcionaba y dónde había margen de mejora. Además el contacto con una comunidad muy comprometida con el desarrollo de proyectos de IA conversacional me ayudó a comprender el potencial de Rasa y los LLM.

Aunque la totalidad de asignaturas que forman parte del Plan de Estudios han ayudado en mayor o menor medida a la finalización con éxito de este proyecto, me gustaría destacar dos asignaturas en concreto.

La elección de una metodología con la que no estaba familiarizado, *Conversation Driven Design (CDD),* hizo que la opción más lógica fuese buscar similitudes con metodologías estudiadas a lo largo del Grado, como el *Diseño Centrado en el Usuario (DCU),* que fue parte de los contenidos de la asignatura **Usabilidad y Accesibilidad**, lo que agilizó mucho el proceso iterativo del proyecto.

Para comprender conceptos relacionados con el Procesamiento de Lenguaje Natural, la asignatura **Fundamentos de Inteligencia Artificial** fue un pilar esencial, que por una parte, ayudó a comprender conceptos base de esta rama del conocimiento, y por otra parte, me familiarizó con sistemas similares a los desarrollados en este proyecto, como los Sistemas Basados en Reglas.

Durante el desarrollo del proyecto encontré principalmente dos dificultades.

En primer lugar, la rápida evolución del sector en el último año, con nuevos modelos, nuevas organizaciones y nuevas herramientas apareciendo de forma constante, dificultó enormemente el desarrollo del proyecto. Por un lado, la gran cantidad de información disponible, complicó la búsqueda de información relevante; por otro lado, información que ya se había incluido en la

memoria quedaba obsoleta en cuestión de pocos meses.

Por otra parte, el cambio en mi vida profesional que supuso la finalización de mi contrato en el Ayuntamiento de Cádiz, espacio en el que se detectó la necesidad del proyecto y en el que se inició su desarrollo, hizo necesario realizar un cambio de perspectiva y objetivos del proyecto, orientándolo hacia un espacio más generalista que le diese viabilidad y relevancia fuera del contexto local.

Pese a ello, estimo que estas dificultades, lejos de perjudicar al proyecto, han dado como resultado una propuesta más sólida y un camino, aunque con altibajos, muy gratificante a nivel académico y profesional.

7.2. Trabajos futuros

Durante el desarrollo han surgido nuevas funcionalidades que considero quedan fuera del alcance del proyecto, pero que sirven para configurar una hoja de ruta para el próximo año de vida del proyecto.

Traducción a otros idiomas.

Durante el desarrollo del PFG, surgió la oportunidad de participar en Rasa Community Challenge. Durante el transcurso del evento, quedó patente la necesidad de contar con una versión en inglés del proyecto, para garantizar mayor adopción y facilitar su difusión en estos entornos.

Mejora automática de metadatos

Utilizando las capacidades generativas del proyecto, se plantea pasar de únicamente generar descripciones de los conjuntos de datos a petición del usuario, a utilizar esa generación de descripciones de forma activa, revisando los conjuntos de datos que carecen de ella y utilizando la API de DKAN para actualizar los datasets con metadatos incompletos. Esto se complementaría con los flujos de trabajo disponibles en Drupal, para garantizar una revisión humana antes de su publicación.

Compatibilidad con CKAN.

Aunque en el proyecto se ha utilizado DKAN, por cuestiones abordadas en el anexo A.1. Se plantea como mejora la integración con el Data Management System más utilizado a nivel global: CKAN[1].

[1] https://ckan.org/

Optimizar el coste de consultas a la API de OpenAI.

Como pudimos ver en la tabla 4.2, el coste del modelo GPT-3.5 Turbo, es bastante inferior al de GPT-4 Turbo, aunque también lo son sus resultados.

Se propone implementar un sistema que permita hacer consultas al modelo GPT3.5 Turbo para consultas sencillas o cuando no sea necesaria una ventana de contexto tan amplia, por ejemplo, para datasets de pequeño tamaño.

Explorar integración de PandasAI.

Como ya se comentó en el capítulo 2.6, durante el desarrollo del Proyecto Fin de Grado que nos ocupa, se inició el desarrollo de un proyecto con objetivos similares. Aunque en la actualidad existe un conflicto de dependencias entre PandasAI y la versión actual de Rasa, parece que dichos problemas serán solucionados en la próxima versión 3.7.

La integración con PandasAI nos permitiría realizar cierto procesado de datos que reduciría el coste de las solicitudes a la API de OpenAI.

Explorar otros LLM.

En el capítulo 2.3 se presentaba una visión general del desarrollo de LLMs actual. En un entorno cambiante como este (dando perspectiva, cuando se presentó el anteproyecto de este PFG, ChatGPT no había sido presentado), resultaría interesante estar al corriente de las actualizaciones y valorar la adopción de otro modelo, preferiblemente open source, que nos permita reducir costes operativos.

Logs

Como mejora del proyecto, se propone añadir una aplicación web con un dashboard para la consulta y filtrado de los registros, momento en es que se podrá configurar un nuevo servicio en *docker-compose-pro.yml* con la imagen oficial de MongoDB[2] y eliminar la dependencia de MongoDB Atlas.

7.3. Estudio de costes de mantenimiento

Mediante la información contenida en los registros, utilizando información sobre distintas consultas sobre conjuntos de datos de diferente extensión, se ha calculado una media de 254 tokens de entrada y 227 tokens de salida por consulta.

[2] https://hub.docker.com/_/mongo

Aunque poco realista, tomando como referencia el uso de proyectos anteriores en el Ayuntamiento de Cádiz, se puede realizar una previsión unas 2500 consultas al mes por parte de los usuarios. Esta previsión deberá ser revisada con datos reales del uso de la herramienta en producción y deberá ser escalada de forma adecuada según las dimensiones de la administración en la que se quiera implantar.

Incluye también el coste de un clúster compartido M2 de MongoDB Atlas en caso de que el tier gratuito no esté disponible o sea insuficiente, aunque como hemos visto en la hoja de ruta, este servicio será sustituido por un contenedor Docker en los próximos meses.

En la tabla 7.1 podemos ver una estimación del coste anual. Se ha realizado conversión a euros del precio original en dólares en algunos casos.

Concepto	Precio Unitario	Cantidad	Total
VPS 2vCPU/2GB	16,50	12	198€
API Input por token	0,00001	7.623.000	76,23€
API Output por token	0,00003	6.810.000	204,3€
MongoDB Shared M2	8,30	12	99,6
Total			578,13€

Tabla 7.1: Coste de mantenimiento anual. Fuente: elaboración propia.

Bibliografía

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[2] X. Kong, G. Wang, and A. Nichol, *Conversational AI with Rasa: Build, test, and deploy AI-powered, enterprise-grade virtual assistants and chatbots*. Packt Publishing Ltd, 2021.

[3] S. Vajjala, B. Majumder, A. Gupta, and H. Surana, *Practical natural language processing: A comprehensive guide to building real-world NLP systems*. O'Reilly Media, 2020.

[4] R. Batish, *Voicebot and Chatbot Design: Flexible Conversational Interfaces with Amazon Alexa, Google Home, and Facebook Messenger*. Packt Publishing Ltd, 2018.

[5] S. Ozdemir, *Quick Start Guide to Large Language Models: Strategies and Best Practices for Using ChatGPT and Other LLMs*. Addison-Wesley Professional, 2023.

[6] O. Caelen and M. A. Blete, *Developing Apps with GPT-4 and ChatGPT*. O'Reilly Media, 2023.

[7] Rasa. Rasa open source docs. [Online]. Available: https://rasa.com/docs/rasa/

[8] F5. nginx documentation. [Online]. Available: https://nginx.org/en/docs/

[9] Telegram. Telegram bot api documentation. [Online]. Available: https://core.telegram.org/bots/api

[10] AWS. Aws documentation. [Online]. Available: https://docs.aws.amazon.com

[11] MongoDB. Mongodb documentation. [Online]. Available: https://www.mongodb.com/docs/

[12] PyPA. virtualenv user guide. [Online]. Available: https://virtualenv.pypa.io/en/latest/user_guide.html

[13] OpenAI. Openai api reference. [Online]. Available: https://platform.openai.com/docs/api-reference

[14] ngrok. ngrok docs. [Online]. Available: https://ngrok.com/docs

[15] Inflection AI. (2023) Inflection-2. [Online]. Available: https://inflection.ai/inflection-2

[16] E. Beeching, C. Fourrier, N. Habib, S. Han, N. Lambert, N. Rajani, O. Sanseviero, L. Tunstall, and T. Wolf. (2023) Open llm leaderboard. https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard.

Apéndice A

DKAN

A.1. Introducción

DKAN[1] es un Data Management System de código abierto, desarrollada por **Civic Actions**[2].

Basado en **CKAN**[3], se publica como una distribución de Drupal especializada en la gestión de datos abiertos y la publicación de catálogos de datos. La plataforma está diseñada para ayudar a las organizaciones a organizar, publicar y compartir conjuntos de datos de manera efectiva, facilitando el acceso a la información y promoviendo la transparencia.

Disponemos de un repositorio público[4] en Github y podemos acceder a una demo[5] de la última versión.

En la última versión, se opta por una arquitectura desacoplada que presenta un backend implementado en Drupal y un frontend mediante un SPA[6] en React.js[7].

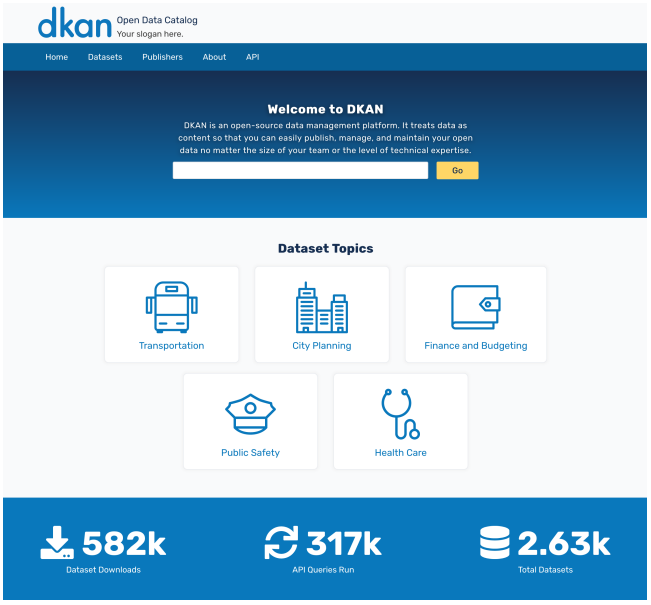


Figura A.1: Home de la demo pública de DKAN

[1] https://getdkan.org/
[2] https://civicactions.com/
[3] https://ckan.org/
[4] https://github.com/GetDKAN/dkan
[5] https://demo.getdkan.org/
[6] https://react.dev/
[7] https://github.com/getdkan/data-catalog-app

Algunas de las características y funcionalidades de DKAN son:

- **Catálogo de Datos:** DKAN proporciona una interfaz para que las organizaciones creen catálogos en línea de conjuntos y distribuciones de datos, lo que facilita la búsqueda y el descubrimiento de los mismos.
- **Metadatos:** La plataforma permite la entrada de metadatos detallados sobre cada conjunto de datos, lo que ayuda a describir su contenido, origen, formato y otros detalles relevantes.
- **APIs:** DKAN proporciona una API[8] que permite el acceso a los datos de manera programática, lo que facilita la integración con otras aplicaciones y servicios.
- **Soporte para Estándares Abiertos:** DKAN cumple con estándares abiertos para la publicación de datos. Está más alineado con la legislación estadounidense, pero cumple perfectamente con los requisitos de la Unión Europea.
- **Visualización de datos:** En la última versión (v2) lamentablemente pierde el módulo de visualizaciones de datos, presente en la v1, aunque esto se puede extender en la SPA desarrollada con React.js con librerias especializadas de visualización de datos, por ejemplo, D3.js[9].

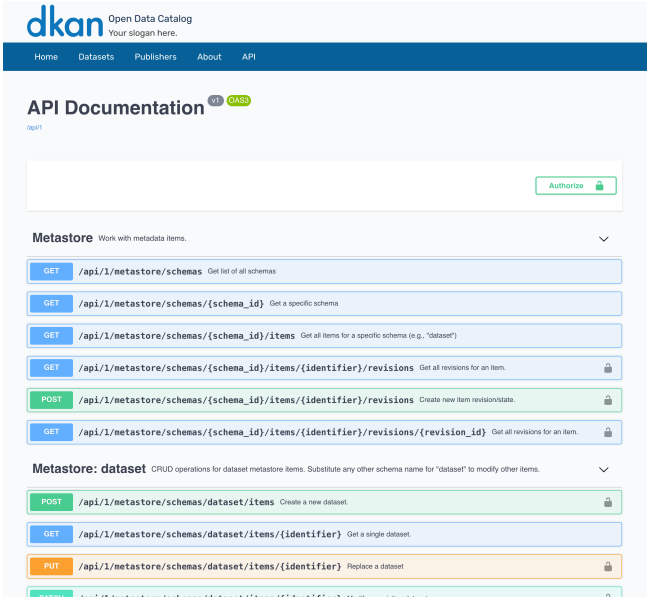


Figura A.2: Detalle de la API de DKAN con Swagger

A.2. ¿Por qué DKAN?

La elección de **DKAN** sobre otras soluciones – como por ejemplo, **CKAN** – se debe a una mera cuestión práctica. Durante el tiempo que trabajé como personal eventual en el **Ayuntamiento de Cádiz**, realicé el desarrollo del Portal de Datos Abiertos Municipal[10] utilizando DKAN.

[8] https://demo.getdkan.org/api
[9] https://d3js.org/
[10] https://datos.cadiz.es/

El stack tecnológico para desarrollo web en la corporación se componía principalmente por Drupal y React/Next.js, por lo que la familiaridad con ambas tecnologías ayudó a compensar los limitados recursos disponibles durante el desarrollo del proyecto. Esto permitió realizar diversas mejoras al proyecto, como la internacionalización mediante react-i18next[11], contribuyendo de esta forma a la comunidad Open Source.

Dado que era necesario acotar el ámbito del Proyecto Fin de Grado e inicialmente el chatbot estaba destinado a ser puesto en producción en el Ayuntamiento de Cádiz, me pareció lógico continuar utilizando **DKAN** como fuente de datos y plantear la compatibilidad con **CKAN** como una futura mejora en el **Roadmap** 7.

[11] https://react.i18next.com/

Apéndice B

Entorno de desarrollo

B.1. Repositorio Git

Git[1] es un sistema de control de versiones distribuido de código abierto. Fue creado por Linus Torvalds en 2005 y se ha convertido en una de las herramientas más utilizadas en el desarrollo de software.

El repositorio del proyecto está disponible de forma pública en **GitHub**[2].

Instalación de Git

En primer lugar, debemos instalar Git en nuestro equipo. Para ello, podemos ejecutar los siguientes comandos.

En macOS, mediante Homebrew[3]:

```
brew install git
```

En Ubuntu o Debian:

```
sudo apt install git
```

Podemos encontrar formas de instalación alternativas en la documentación de Git[4].

Clonación del repositorio

Una vez instalado, procedemos a clonar el proyecto en nuestro equipo local mediante el comando:

```
git clone https://github.com/alex-ahumada/opendata-genai-bot.git
```

[1] https://git-scm.com/
[2] https://github.com/alex-ahumada/opendata-genai-bot
[3] https://brew.sh/
[4] https://git-scm.com/downloads

Alternativamente, podemos usar el siguiente comando para copiarlo mediante SSH.

```
gir clone git@github.com:alex-ahumada/opendata-genai-bot.git
```

Dependabot

Dependabot[5] es un servicio de automatización integrado en Github que ayuda a los desarrolladores a mantener actualizadas las dependencias de sus proyectos de software.

Una de las características destacables de Dependabot, son las alertas de seguridad, que durante el desarrollo del proyecto ayudó a detectar y resolver diversas vulnerabilidades presentes en el código.

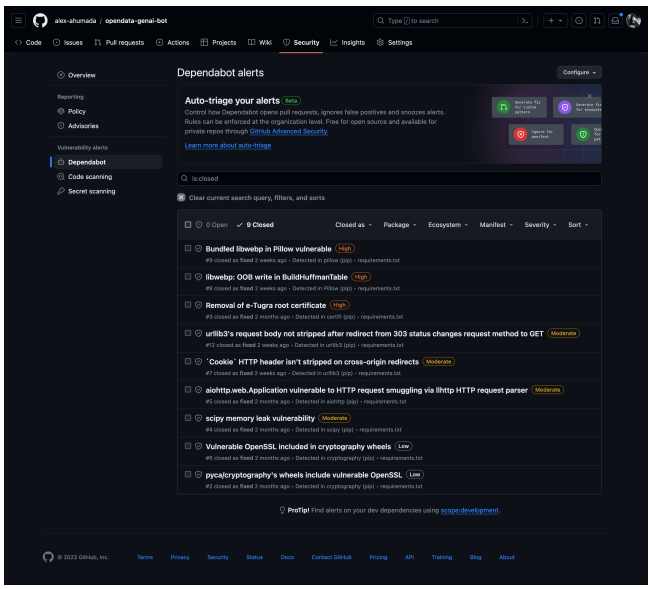


Figura B.1: Alerta de vulnerabilidades en Dependabot

B.2. Entorno de desarrollo

A continuación se detallan diversas herramientas utilizadas durante el desarrollo del proyecto.

pyenv

pyenv[6] es una herramienta que permite gestionar múltiples versiones de Python en un mismo sistema. Con *pyenv*, puedes instalar y cambiar entre diferentes versiones de Python, lo que es útil si estás trabajando en varios proyectos que requieren versiones específicas del lenguaje.

[5] https://docs.github.com/en/code-security/dependabot/working-with-dependabot
[6] https://github.com/pyenv/pyenv

También permite establecer una versión de Python global para tu usuario o una versión predeterminada para un directorio local.

Para instalar *pyenv* en nuestro sistema, podemos ejecutar los siguientes comandos:

En macOS:

```
brew install pyenv
```

En Ubuntu/Debian:

```
curl https://pyenv.run | bash
```

A continuación, debemos configura nuestro entorno de shell para que pyenv funcione correctamente.

```
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'command -v pyenv >/dev/null || export PATH="$PYENV_ROOT/bin:$PATH
    "' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
```

Finalmente, podemos instalar la versión de Python deseada. En el caso de Rasa, la versión recomendada actualmente es la 3.10.

```
pyenv install 3.10.13 # instalar version deseada
pyenv local 3.10.13 # establecer como version predeterminada (local)
```

virtualenv e instalación de dependencias

virtualenv[7] es una herramienta en Python que se utiliza para crear entornos virtuales aislados. Un entorno virtual es un directorio que contiene una instalación de Python y permite que un proyecto o aplicación tenga sus propias dependencias sin interferir con otras aplicaciones.

En Python 3.3 y versiones posteriores, se incluye **venv,** herramienta similar que dispone de un subconjunto de las funciones de virtualenv. En nuestro caso de uso puede ser una buena alternativa si no queremos instalar dependencias adicionales.

Para instalar *virtualenv*, podemos usar la herramienta *pipx*, disponible en Python v3.7 en adelante.

```
pipx install virtualenv
```

Finalmente, podemos crear el entorno virtual, activarlo e instalar las dependencias mediante PIP con los siguientes comandos.

[7] https://virtualenv.pypa.io/en/latest/

```
virtualenv venv #creacion del entorno virtual
source venv/bin/activate #activacion del entorno virtual
pip install -r requirements.txt #instalacion de dependencias
```

Si al crear el entorno virtual, *virtualenv* utiliza la versión por defecto de Python instalada en nuestro sistema en lugar de la versión configurada mediante pyenv, podemos indicar una versión específica mediante el comando:

```
virtualenv -p /Users/alexahumada/.pyenv/shims/python venv
```

El entorno local se desactiva de esta forma.

```
deactivate
```

B.3. Tunelización de localhost

Durante el desarrollo local, tendremos que exponer el puerto de Rasa NLU Core, por defecto *5005*, para la configuración del webhook de Telegram. Para ello se recomienda el uso de una herramienta de tunelización.

Para la configuración de las credenciales de los canales de Rasa, entre los que encontramos Telegram, será necesario renombrar o copiar credentials.example.yml como credentials.yml e incluir:

- token de acceso de Telegram,
- nombre del bot de Telegram,
- url del webhook, que obtendremos mediante *ngrok* o *cloudflared*.

```
telegram:
  access_token: "[telegram_access_token]"
  verify: "[botname]_bot"
  webhook_url: "[url]/webhooks/telegram/webhook"
```

ngrok

ngrok[8] es una herramienta que permite exponer de manera temporal, mediante túneles seguros, un servidor local a Internet, lo que puede ser útil para desarrollo local, pruebas, demostraciones

[8] https://ngrok.com/

y otras situaciones en las que necesitas proporcionar acceso externo a una aplicación o servicio que se está ejecutando localmente en tu máquina.

En nuestro caso, debemos exponer el puerto *5005* (o el puerto que configuremos manualmente como veremos más adelante).

```
ngrok http 5005
```

Mediante este comando, obtendremos una URL como, por ejemplo, *https://6483-83-47-18-111.ngrok-free.app*.

cloudflared

Como alternativa, podemos utilizar Cloudflare Tunnel mediante la herramienta de línea de comando **cloudflared**[9].

La podemos instalar en Debian/Ubuntu mediante Cloudflare Package Repository[10] y en macOS con el siguiente comando.

```
brew install cloudflare/cloudflare/cloudflared
```

Finalmente, de forma similar a ngrok, creamos el túnel al puerto deseado.

```
cloudflared tunnel --url localhost:5005
```

Con ello, btendremos una URL similar a *https://finance-hdtv-mention-grid.trycloudflare.com*, que también podremos incluir en *credentials.yml*.

B.4. Ejecución de Rasa en entorno local

Se proponen dos opciones para ejecutar **Rasa NLU Core** y **Rasa SDK Actions** en entorno local. Podemos ejecutar ambos servidores directamente mediante nuestro terminal con el comando *rasa run*. Como opción preferente, ideal para familiarizarnos con el uso de Docker Compose, ya que será la herramienta que usaremos en producción, también podemos utilizar **Docker** para el desarrollo local.

Mediante CLI

En primer lugar es importante recordar activar el entorno virtual descrito en B.2 e instalar las dependencias presentes en *requirements.txt*. Adicionalmente, para la ejecución de los servidores

[9] https://github.com/cloudflare/cloudflared
[10] https://pkg.cloudflare.com/index.html

de Rasa mediante CLI, tendremos que editar el archivo endpoints.yml para configurar el Action Server, ya que en el repositorio lo hemos configurado por defecto para funcionar con Docker Compose. Las líneas a modificar son:

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

Rasa NLU Core

Para iniciar el servidor principal de Rasa, debemos ejecutar el siguiente comando:

```
rasa run --enable-api --endpoints endpoints.yml --port 5005 --
    credentials credentials.yml --model models --cors "*" --debug
```

- - enable-api (opcional) Habilita la API.

- - endpoints (opcional) Define un archivo de endpoints. Por defecto, *endpoints.yml*.

- - port (opcional) Permite cambiar el puerto por defecto *5005*.

- - credentials (opcional) Define un archivo de credenciales. Por defecto, *credentials.yml*.

- - model (opcional) Permite definir el modelo a cargar. Por defecto utiliza el último modelo entrenado.

- - cors (opcional) Habilita CORS para el origen indicado. También se puede usar el wildcard * para permitir todos los orígenes.

- - debug (opcional) Este argumento activa el modo debug.

Si usamos las opciones por defecto, simplemente podemos ejecutar:

```
rasa run --enable-api --debug
```

Rasa SDK Action Server

En el caso del motor de acciones, su puesta en funcionamiento es mucho más sencilla y tan solo requiere ejecutar el comando:

```
rasa run actions --actions actions --port PORT --debug
```

- - actions (opcional) El argumento opcional - -actions permite definir una carpeta personalizada para las acciones. Por defecto se encuentra en la carpeta */actions*.

- - port (opcional) Permite cambiar el puerto por defecto *5055*.

- - debug (opcional) Este argumento activa el modo debug.

Para ejecutar el comando en desarrollo con las opciones por defecto, se recomienda utilizar:

```
rasa run actions --debug
```

Mediante Docker

La forma más cómoda de instalar Docker para desarrollo local es Docker Desktop[11], una aplicación de escritorio que proporciona una GUI y herramientas adicionales que simplifican el proceso de desarrollo, prueba e implementación de aplicaciones mediante contenedores.

En en el anexo C.2, se realizará una explicación detallada de cómo construir imágenes, así como una ampliación de la información sobre Docker y Docker Compose.

Se define un archivo *docker-compose.yml,* que será el utilizado para desarrollo local, con dos servicios, para el servidor NLU y el servidor de acciones.

Ambos servicios hacen uso de imágenes para el Rasa NLU Core Server[12] y Rasa SDK Action Server [13] que se encuentran disponibles en **Docker Hub**[14].

```
version: "3.7"

services:
  rasa:
    image: alexahumada/opendata-genai-bot-rasa:latest
    ports:
      - 5005:5005
    volumes:
      - ./:/app
    command:
      - run
      - --debug
    restart: unless-stopped
  action_server:
    image: alexahumada/opendata-genai-bot-actions:latest
    volumes:
      - ./actions:/app/actions
    expose:
      - 5055
    command:
      - start
      - --actions
      - actions.actions
```

[11] https://www.docker.com/products/docker-desktop/
[12] https://hub.docker.com/repository/docker/alexahumada/opendata-genai-bot-rasa
[13] https://hub.docker.com/repository/docker/alexahumada/opendata-genai-bot-actions
[14] https://hub.docker.com/

```
24       - --debug
25     restart: unless-stopped
```

Para crear los contenedores e iniciar ambos servidores podemos utilizar el siguiente comando desde la carpeta raíz del proyecto:

```
1 docker compose up -d
```

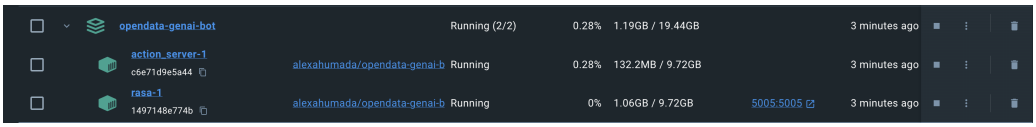


Figura B.2: Contenedores en Docker Desktop

Podemos parar todos los contenedores mediante el comando:

```
1 docker compose stop
```

Finalmente, el comando *down* para apagará y eliminará los contenedores:

```
1 docker compose down
```

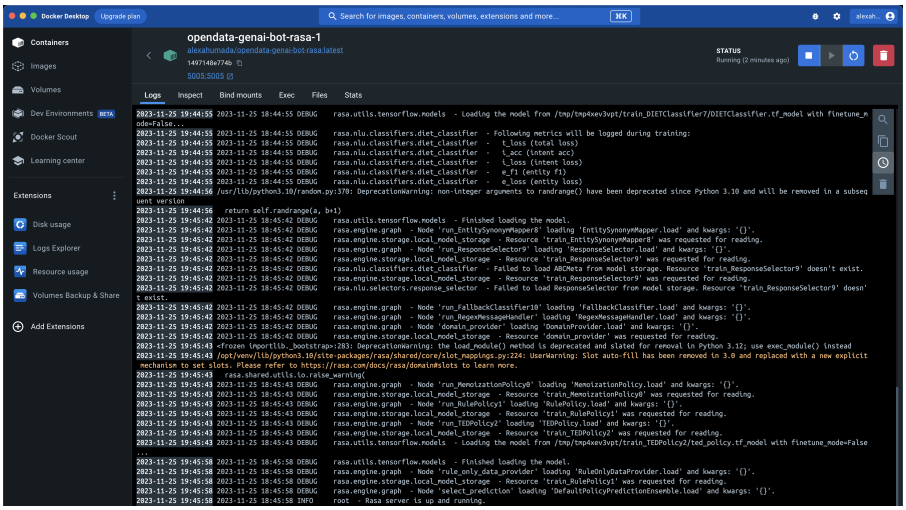


Figura B.3: Logs en Docker Desktop

Al hacer click en uno de los contenedores, se muestran sus logs(B.3). De forma alternativa, podemos acceder a los logs mediante CLI.

```
1 docker logs
```

Es importante tener en cuenta que no se define ningún servicio para el **Tracker Store,** por lo que se utilizará el *InMemoryTrackerStore* por defecto y el historial de conversaciones se perderá al apagar los contenedores.

B.5. Entrenamiento de modelos

Disponemos de una serie de comandos CLI para entrenar y probar nuestros modelos.

El más importante, nos permite entrenar nuestro modelo con los datos disponibles, se guardará automáticamente en la carpeta *models* al finalizar el proceso de entrenamiento.

```
rasa train
```

También podemos iniciar una sesión interactiva que nos permite generar nuevos datos para el entrenamiento del modelo sin tener que añadir los mismos manualmente a los diferentes archivos del proyecto.

```
rasa interactive
```

Este comando inicia una conversación en el terminal con el último modelo entrenado, sin necesidad de usar los endpoints configurados.

```
rasa shell
```

Realiza los tests definidos en el proyecto sobre el último modelo entrenado.

```
rasa test
```

Apéndice C

Despliegue en producción

C.1. Requisitos del sistema

Recursos disponibles

Aunque se deberán escalar los recursos del sistema en función de la demanda por parte de los usuarios, como podemos ver en la tabla C.1 se recomiendan unas especificaciones mínimas para cada uno de los modelos spaCy utilizados.

Modelo	CPU	RAM
es_core_news_sm	2 vCPUs	2 GB
es_core_news_md	2 vCPUs	4 GB

Tabla C.1: Especificaciones mínimas recomendadas

Software necesario

Será necesaria la instalación de:

- Git
- Nginx y Certbot
- Docker

C.2. Instrucciones de despliegue

Repositorio Git

Como ya se trató en la sección B.1 del Anexo 2, será necesario instalar **Git**[1] y clonar el repositorio del proyecto[2].

[1] https://git-scm.com/
[2] https://github.com/alex-ahumada/opendata-genai-bot

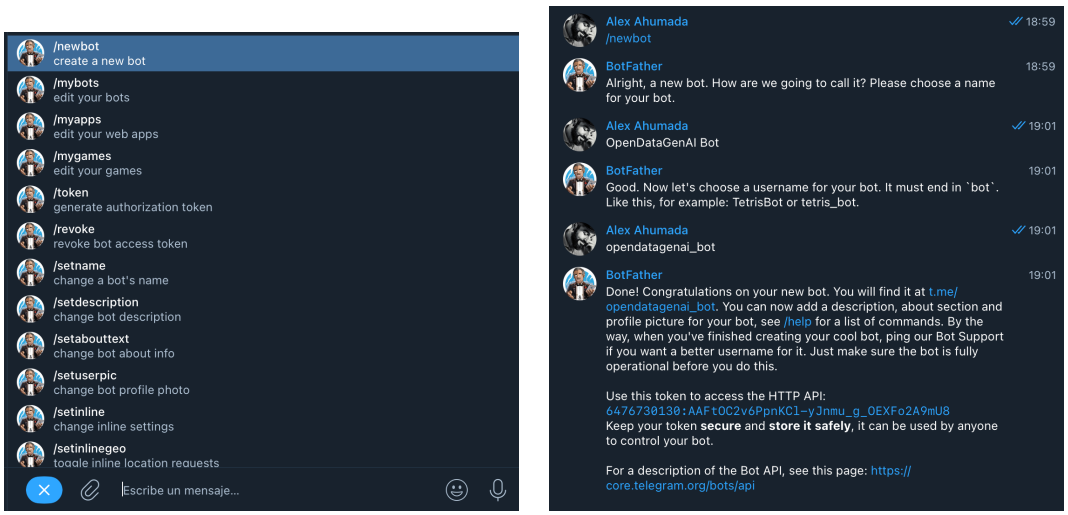
```
sudo apt install git
git clone https://github.com/alex-ahumada/opendata-genai-bot.git
```

Alta de bot en Telegram

Telegram permite la creación y gestión de bots mediante el bot **@BotFather**[3], para lo que tendremos que seguir una serie de pasos:

1. Iniciar una conversación con **BotFather**.
2. Crear un bot con el comando */newbot*.
3. Introducir nombre y nombre de usuario, teniendo en cuenta que el nombre de usuario debe acabar en *_bot*

Como podemos ver en la figura C.1, durante el proceso de creación del bot, se nos facilitará un token para el acceso a la API de **Telegram** que debemos anotar ya que lo necesitaremos para la configuración de credenciales en **Rasa**.



(a) Menu de comandos de BotFather

(b) Definición del nuevo bot mediante el comando /newbot

Figura C.1: Proceso de creación de nuevo bot

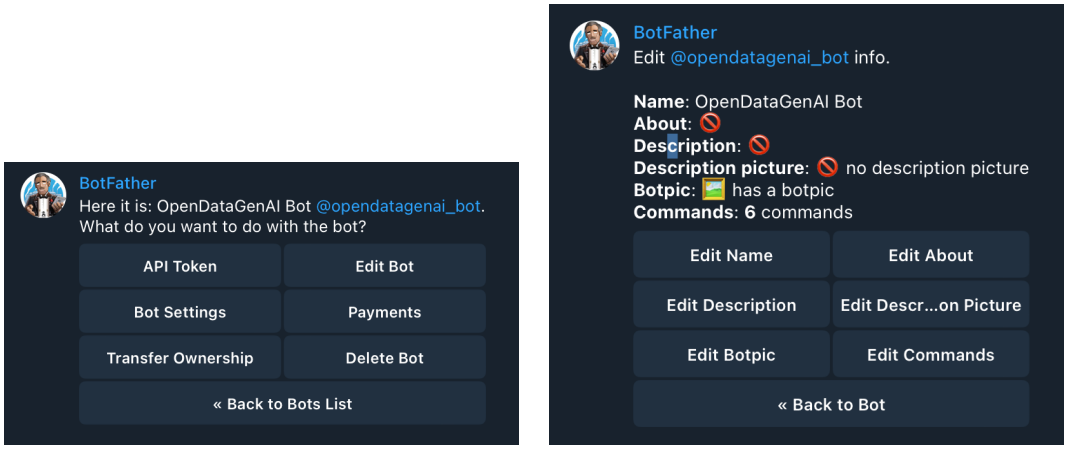
BotFather también ofrece una serie de opciones adicionales.

- **Personalizar el bot.**
  Podemos añadir una imagen de perfil y una descripción

[3] https://t.me/BotFather

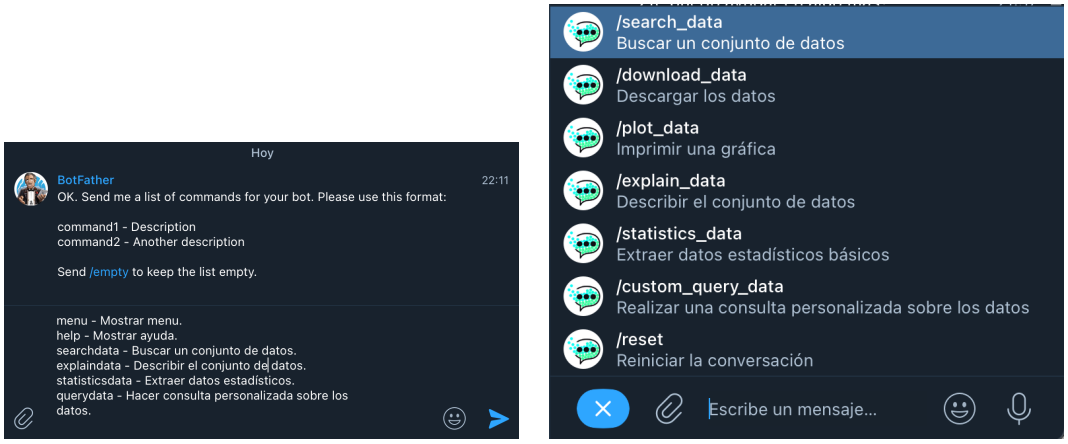
- **Definir comandos.**
  Los comandos definidos aparecerán el el menú de comandos del bot, mejorando de esta forma la experiencia del usuario.

- **Opciones avanzadas.**
  Las opciones avanzadas permiten, entre otras acciones: crear otro API token, transferir la propiedad a otro usurio o eliminar el bot.



(a) Menú de configuración (b) Menú de edición

Figura C.2: Edición del bot



(a) Introducción de comandos (b) Menú de comandos

Figura C.3: Configuración de comandos

MongoDB

Como detallamos en 4.6, **MongoDB**[4] es un sistema de gestión de bases de datos **NoSQL**. En nuestro proyecto, utilizamos **MongoDB** para almacenar los logs de las consultas, que incluyen la siguiente información:

[4] https://www.mongodb.com

- ID anonimizado de conversación.
- Acción ejecutada (descripción, datos estadísticos o custom query).
- Modelo utilizado.
- Prompt.
- Coste: por tokens de entrada y de salida.

Pasos a seguir

La forma mas sencilla de disponer de una instalación de MongoDB, es usar MongoDB Atlas[5], un servicio de base de datos en la nube ofrecida por MongoDB. Dispone de un tier gratuito que será suficiente para el inicio de nuestro proyecto.

Para ello será necesario seguir una serie de pasos:

1. Dar de alta una cuenta en MongoDB Atlas.
2. Crear un clúster (C.4).
3. Definir los permisos de un nuevo usuario (C.6) y obtener un connection string (C.7).
4. Añadir una nueva base de datos llamada *"logs"* con una colección llamada *"completions"*(C.8).

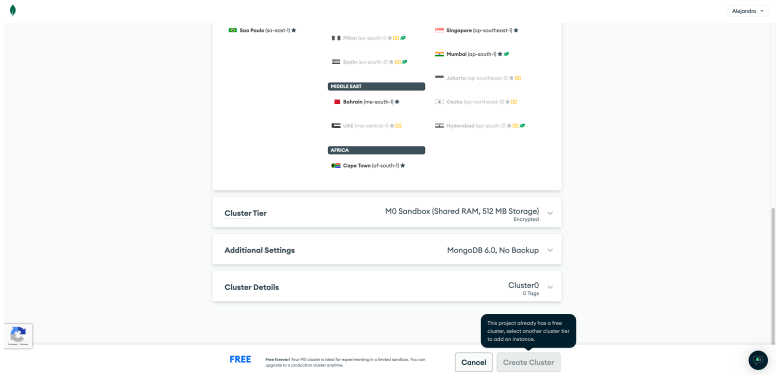


Figura C.4: Creación de un nuevo clúster en MongoDB Atlas

¿Por qué MongoDB Atlas?

Una de las ventajas del uso de MongoDB Atlas es que añade una interfaz web para realizar consultas sobre las colecciones, además de otra serie de herramientas como gráficas y consultas sobre las colecciones mediante IA.

En 7.2 se propone una futura mejora que sustituye el uso de MongoDB Atlas por una instalación local de MongoDB.

[5] https://www.mongodb.com/atlas

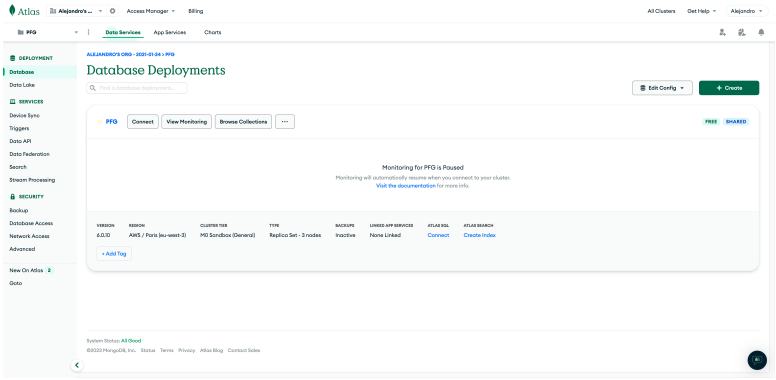


Figura C.5: Panel database deployments

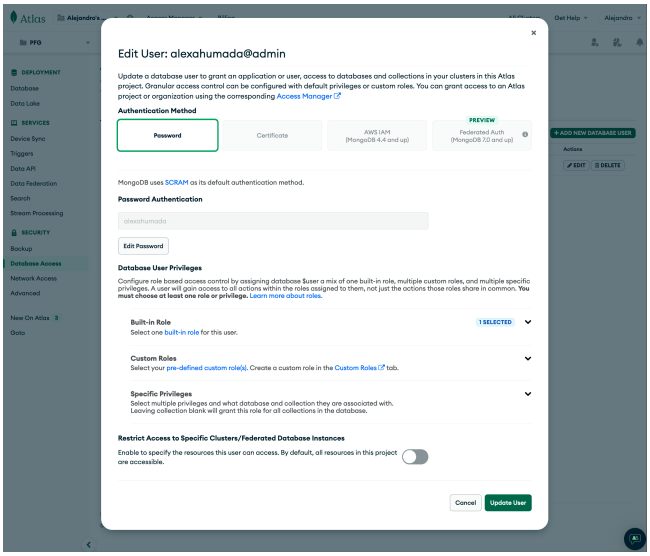


Figura C.6: Creación de usuario y gestión de credenciales en MongoDB Atlas

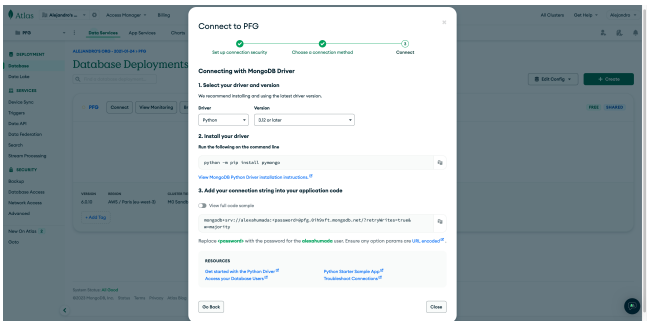


Figura C.7: Detalle del connection string

AWS S3

Amazon S3 (Amazon Simple Storage Service)[6], es un servicio de almacenamiento en la nube proporcionado por Amazon Web Services.

En AWS, necesitamos realizar tres acciones:

[6] https://aws.amazon.com/es/s3/

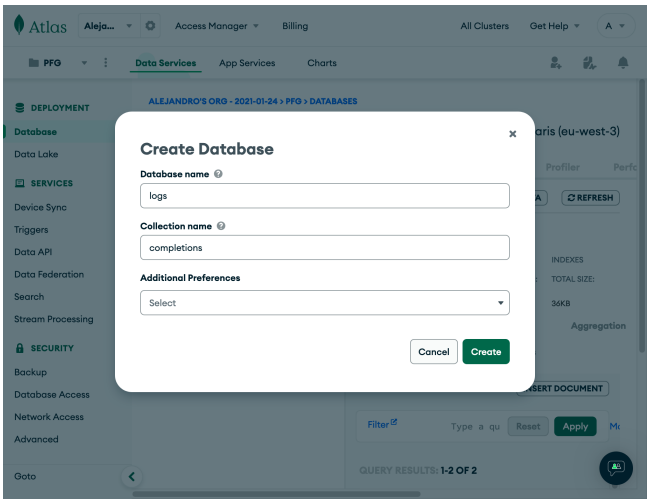


Figura C.8: Creación de database y colección en MongoDB Atlas

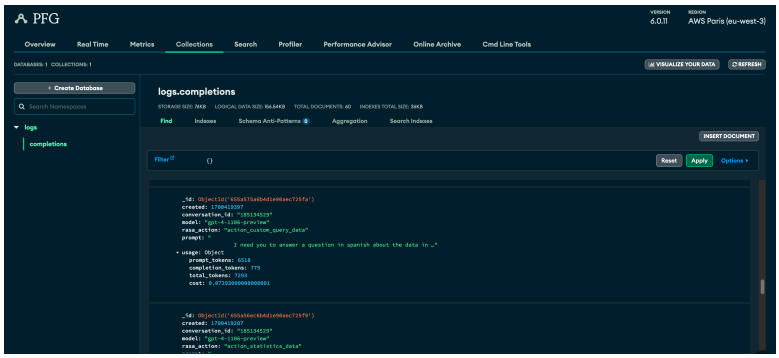


Figura C.9: Ejemplo de consulta en MongoDB Atlas.

1. Crear un bucket y configurar sus permisos.
2. Crear una politica IAM.
3. Crear un usuario, asignarle la política y obtener clave de acceso.

Creación del bucket

Podemos crear un nuevo bucket en el menú principal de S3[7]. Es importante destacar que el nombre del bucket debe ser único a nivel global.

En la configuración del bucket debemos bloquear todo el acceso público, ya que nuestras acciones en Rasa se encargan de generar un enlace público para cada objeto durante el envío al usuario.

Creación de política IAM

Para crear una nueva política, debemos acceder a *IAM / Administración del acceso / Políticas.* Para facilitar el proceso, podemos utilizar este código en el que únicamente necesitamos cambiar

[7] https://s3.console.aws.amazon.com/s3/home

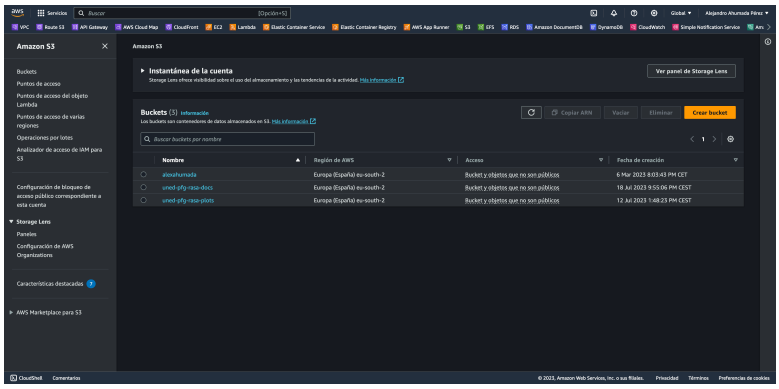


Figura C.10: Listado de buckets en AWS S3

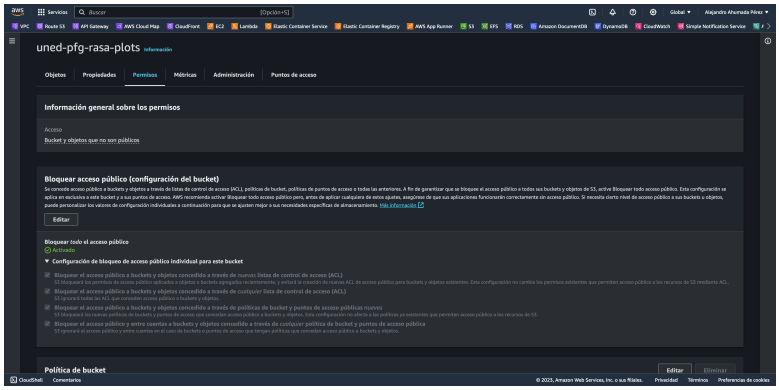


Figura C.11: Configuración del bucket

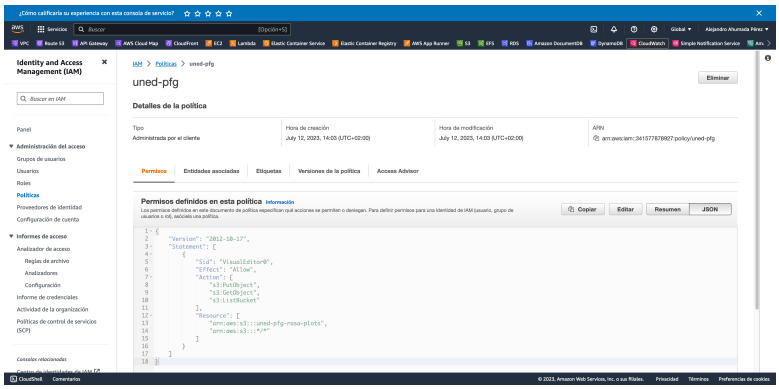


Figura C.12: Creación de una nueva política en IAM

el nombre del bucket.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
```

```
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::nombre-bucket",
                "arn:aws:s3:::*/*"
            ]
        }
    ]
}
```

Creación de usuario en IAM

Una vez dispongamos de nuestra política de permisos, podemos proceder a crear un nuevo usuario y asignarle la política con acceso a S3. También tendremos que obtener una clave de acceso ya que se utilizará en las variables de entorno de Rasa.

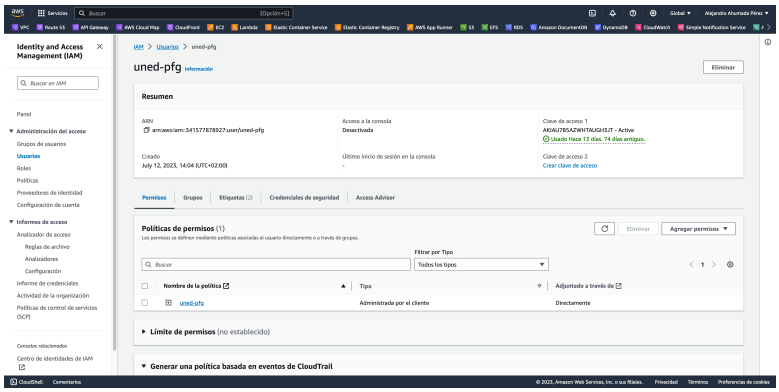


Figura C.13: Creación de una nuevo usuario en IAM

OpenAI

Para usar la API de **OpenAI**[8], tendremos que crear un nuevo usuario de la API y generar una clave de acceso.

El la sección *Usage*[9] podemos ver un desglose del uso y coste de la API. Cabe señalar que no es necesario ser usuario de pago de **ChatGPT** para poder acceder a la API, que solo requiere pago por uso. Al crear la cuenta, únicamente podremos usar modelos de OpenAI hasta gpt-3.5-turbo, el uso de gpt-4 en adelante requiere cumplir de ciertos requisitos que han variado en el tiempo. En el momento de redactar esta memoria, se requiere realizar una compra de al menos $5.00 en créditos prepago[10].

[8] https://platform.openai.com
[9] https://platform.openai.com/usage
[10] https://platform.openai.com/account/billing/overview

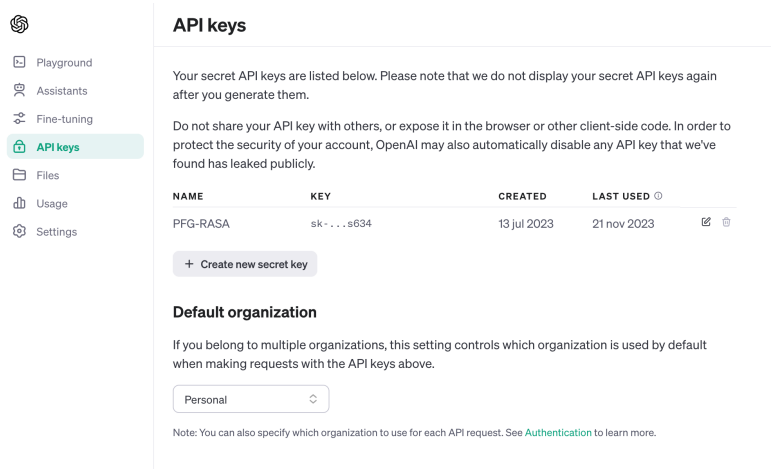


Figura C.14: Creación de API key en OpenAI

Figura C.15: Panel Usage en OpenAI

Variables de entorno

Para configurar las variables de entorno es necesario copiar y renombrar el archivo *.env.example* como *.env* e incluir toda la información de conexión a Telegram, OpenAI, DKAN, AWS y MongoDB obtenida en las secciones anteriores del presente anexo.

```
TELEGRAM_API_TOKEN="" # Telegram API token
TELEGRAM_BOT_NAME="" # Telegram nombre de bot
TELEGRAM_WEBHOOK_URL="" # Telegram webhook url
DKAN_API="" # DKAN API url
OPENAI_API_KEY="" # OpenAI API key
OPENAI_MODEL="" # OpenAI modelo por defecto
AWS_ACCESS_KEY_ID="" # AWS access key id
AWS_SECRET_ACCESS_KEY="" # AWS secret access key
AWS_REGION="" # AWS region
S3_BUCKET_NAME="" # AWS S3 nombre de bucket
MONGODB_URI="" # MongoDB connection URI
```

Configuración adicional de Rasa

Configuración de canales

Para la configuración de las credenciales de los canales de Rasa, entre los que encontramos Telegram, será necesario renombrar o copiar credentials.example.yml como credentials.yml e incluir:

- token de acceso de Telegram,
- nombre del bot de Telegram,
- url del webhook, que coincidirá con el server_name definido durante la configuración de NGINX.

```
telegram:
  access_token: "[telegram_access_token]"
  verify: "[botname]_bot"
  webhook_url: "[https://your_url.com]/webhooks/telegram/webhook"
```

Copiar nuevos modelos en el servidor

Podemos trasladar los nuevos modelos entrenados localmente mediante el comando SCP.

```
scp ./models/nombre-modelo.tar.gz user@server:/ruta/a/proyecto/opendata
    -genai-bot/models
```

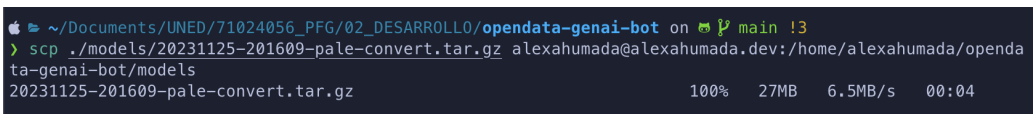


Figura C.16: Uso de SCP para traslado de modelos a producción

Configuración de NGINX

NGINX[11] es un servidor web y servidor proxy inverso de código abierto. Es conocido por su capacidad para manejar grandes cargas de trabajo y proporcionar un rendimiento sólido y escalable.

Para exponer el webhook de **Telegram,** será necesario configurar un proxy al puerto 5005 (o el que configuremos manualmente al ejecutar Rasa NLU Core Server) en **NGINX.**

[11] https://www.nginx.com/

```
sudo apt install nginx # Instalar NGINX
sudo ufw allow 'Nginx HTTP'
sudo ufw allow 'Nginx HTTPS'
```

Luego, crearemos un nuevo server block, se muestra la configuración utilizada.

```
sudo vi /etc/nginx/sites-available/bot.your_domain # creamos el archivo
    con Vim u otro editor
```

```
server {
        listen 8000;
        listen [::]:8000;

        server_name bot.yourdomain.com;

        location / {
        proxy_pass http://localhost:5005;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        }

        location ~ /\.ht {
                deny all;
        }t


    listen [::]:443 ssl; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/bot.yourdomain.com/fullchain.
       pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/bot.yourdomain.com/
       privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by
       Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
```

Finalmente, habilitamos el server block mediante un enlace simbólico, comprobamos la configuración y reiniciamos **NGINX.**

```
sudo ln -s /etc/nginx/sites-available/bot.your_domain /etc/nginx/sites-
    enabled/
sudo nginx -t # comprobamos la configuracion
sudo systemctl restart nginx # reiniciamos NGINX
```

Obtener un certificado con Certbot

Como podemos ver en el ejemplo, estamos utilizando un certificado SSL en nuestra configuración de **Nginx**.

Certbot[12] es una herramienta de software de código abierto, desarrollada por la Electronic Frontier Foundation (EFF)[13], diseñada para simplificar y automatizar el proceso de obtención, renovación e instalación de certificados Let's Encrypt. **Let's Encrypt**[14] es una organización sin ánimo de lucro que ofrece certificados SSL de forma gratuita.

A continuación detallamos como crear el certificado para nuestro subdominio haciendo uso de **Certbot**.

```
sudo apt install certbot python3-certbot-nginx # instalar dependencias
sudo certbot --nginx -d bot.yourdomain.com # crear certificado
```

Podemos comprobar que la renovación automática del certificado está activa mediante el comando.

```
sudo systemctl status certbot.timer
```

O ejecutar la renovación si por algún motivo no se hubiese renovado automáticamente.

```
sudo certbot renew
```

Ejecución de servidores mediante Docker Compose

Docker[15] es una plataforma de software que utiliza contenedores para facilitar la implementación y ejecución de aplicaciones. Un contenedor es una unidad ligera y portátil que incluye todo lo necesario para ejecutar software, incluyendo el código, las bibliotecas, las dependencias y las configuraciones, lo que garantiza que se ejecute de manera consistente en cualquier entorno.

En el entorno local, instalamos Docker mediante la aplicación de escritorio Docker Desktop. Para instalar Docker CE en nuestro servidor de producción debemos seguir los siguientes pasos:

En primer lugar, debemos instalar las claves GPG de Docker y añadir el repositorio a APT.

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
    dearmor -o /etc/apt/keyrings/docker.gpg
```

[12] https://certbot.eff.org/
[13] https://www.eff.org/
[14] https://letsencrypt.org/
[15] https://docs.docker.com/compose/

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Finalmente, instalamos Docker CE y sus dependencias.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
    buildx-plugin docker-compose-plugin
```

Docker Compose[16] es una herramienta que permite orquestar contenedores en Docker. Podemos definir un entorno multi-contenedor en un archivo *YAML,* especificando cómo interactúan esos contenedores entre sí. Esto es útil para definir y orquestar aplicaciones complejas que pueden constar de varios servicios.

Se define un archivo *docker-compose-pro.yml,* que será el utilizado en producción, con tres servicios: rasa, action_server y postgres.

```
version: "3.7"

services:
  rasa:
    image: alexahumada/opendata-genai-bot-rasa:latest
    ports:
      - 5005:5005
    volumes:
      - ./:/app
    command:
      - run
      - --endpoints
      - endpoints-pro.yml
    restart: unless-stopped
  action_server:
    image: alexahumada/opendata-genai-bot-actions:latest
    expose:
      - 5055
    restart: unless-stopped
  postgres:
    image: postgres:latest
    volumes:
      - rasa-postgres:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: rasa
      POSTGRES_PASSWORD: rasa
      POSTGRES_DB: rasa
    restart: unless-stopped

volumes:
  rasa-postgres:
```

[16] https://docs.docker.com/compose/

En el caso de Rasa, ambos servicios hacen uso de imágenes personalizadas para el Rasa NLU Core Server[17] y Rasa SDK Action Server [18] que se encuentran disponibles en **Docker Hub**[19] e incluyen dependencias adicionales que no se incluyen en la imagen oficial de Rasa.

Por otra parte, para habilitar un **Tracker Store** con persistencia de datos, se habilita un tercer servicio con un servidor **PostgreSQL**[20] mediante su imagen oficial[21] en Docker Hub. Para garantizar la persistencia de datos al volver a crear los contenedores, se habilita un volumen llamado *rasa-postgres.*

Desde el directorio raíz de nuestro proyecto, donde se encuentra el archivo *docker-compose-pro.yml,* podemos levantar todos los servicios necesarios.

```
docker compose -f docker-compose-pro.yml up -d # el flag -d lo ejecuta
    en segundo plano
```

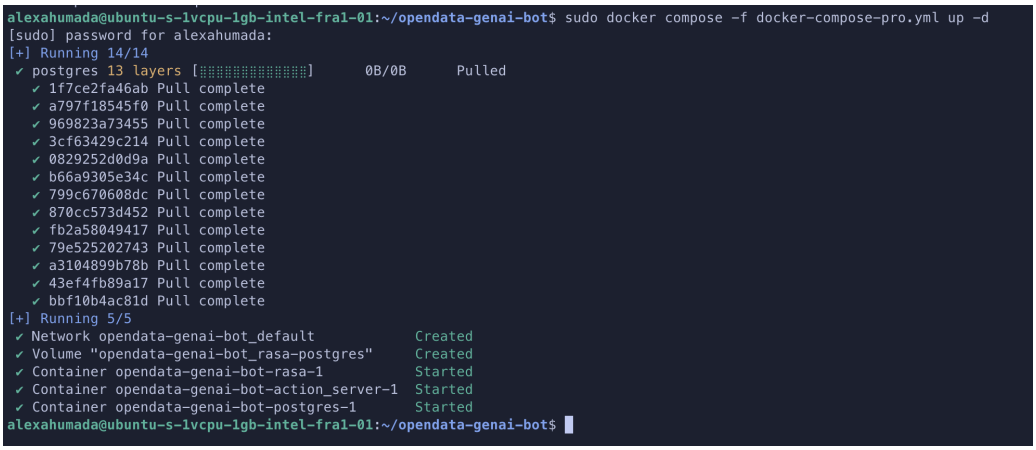


Figura C.17: Ejecución del comando up en Docker Compose

Para mostrar los contenedores en ejecución:

```
docker ps
```

Podemos mostrar los logs de los contenedores usamos:

```
docker compose -f docker-compose-pro.yml logs # todos los logs
docker compose -f docker-compose-pro.yml logs nombre_servicio # logs de
    servicio
```

También podemos vigilar el consumo de recursos de los contenedores:

```
docker stats # Actualizacion constante
```

[17] https://hub.docker.com/repository/docker/alexahumada/opendata-genai-bot-rasa
[18] https://hub.docker.com/repository/docker/alexahumada/opendata-genai-bot-actions
[19] https://hub.docker.com/
[20] https://www.postgresql.org/
[21] https://hub.docker.com/_/postgres/

```
docker stats --no-stream # Resultado puntual
```

El comando *stop* detiene la ejecución todos los contenedores:

```
docker compose -f docker-compose-pro.yml stop
```

Finalmente, el comando *down* para apagará y eliminará los contenedores:

```
docker compose -f docker-compose-pro.yml down
```

Podemos pasar el flag *-v* para que elimine los volúmenes además de los contenedores.

```
docker compose -f docker-compose-pro.yml down -v
```

Build de imágenes personalizadas

En primer lugar, disponemos de un archivo *Dockerfile* para construir la imagen del servidor de acciones. Esto instalará las dependencias de Python presentes en *actions/requirements-actions.txt,* además de copiar las acciones a la nueva imagen.

```
# Extend the official Rasa SDK image
FROM rasa/rasa-sdk:3.6.2

# Use subdirectory as working directory
WORKDIR /app

# Copy any additional custom requirements, if necessary (uncomment next
    line)
COPY actions/requirements-actions.txt ./
COPY .env ./

# Change back to root user to install dependencies
USER root

# Install extra requirements for actions code, if necessary (uncomment
   next line)
RUN pip install -r requirements-actions.txt

# Copy actions folder to working directory
COPY ./actions /app/actions

# By best practices, don't run the code with root user
USER 1001
```

Podemos generar una nueva imagen para el servidor de acciones, se recomienda realizar esta tarea en un equipo local o mediante algún workflow de CI/CD, evitando generar la imagen

desde el servidor de producción, ya que consume muchos recursos y puede resultar un proceso largo en equipos más limitados.

```
docker build --no-cache -t [username]/opendata-genai-bot-actions:latest
    .
```

De forma adicional, disponemos de un archivo llamado *Dockerfile-rasa* que nos permite generar la imagen para el servicio Rasa NLU Core. Usa como base la imagen oficial *rasa/rasa:3.6.14-full*, añadiendo otras dependencias como **spaCy** y los modelos *es_core_news_sm* y *es_core_news_md*.

```
# Use the Rasa SDK image as the base image
FROM rasa/rasa:3.6.14-full

# Switch to the root user
USER root

# Install the spacy package
RUN pip install spacy

# Download the es_core_news_es and es_core_news_md models
RUN python -m spacy download es_core_news_sm
RUN python -m spacy download es_core_news_md
```

Si realizamos alguna modificación en el proyecto que requiera nuevas dependencias, las podemos añadir a dicho archivo y ejecutar el siguiente comando.

```
docker build --no-cache -t [username]/opendata-genai-bot-rasa:latest -f
    Dockerfile-rasa .
```

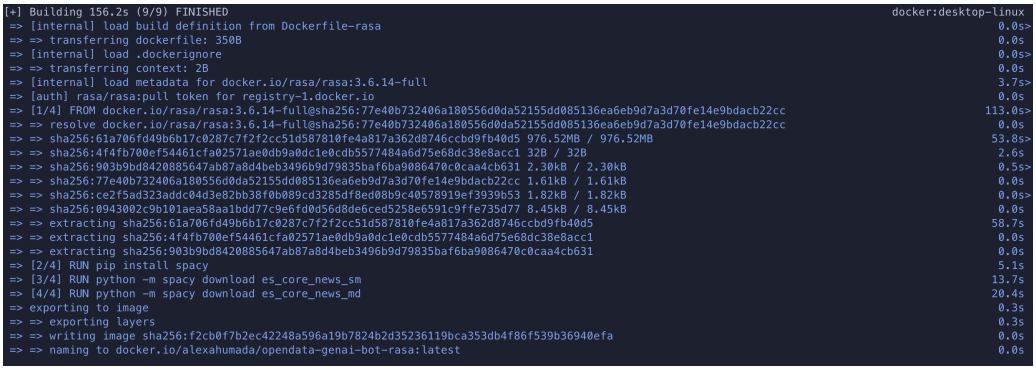


Figura C.18: Generación de nueva imagen de Docker

Finalmente, podemos publicar las nuevas imágenes a Docker Hub. No debemos olvidar cambiar el nombre de la nueva imagen en los archivos de configuración de Docker Compose. Para descargar imágenes en otro equipo, podemos usar el comando *pull*.

```
docker push [username]/opendata-genai-bot-actions:latest
docker push [username]/opendata-genai-bot-rasa:latest
```

Alternativas a Docker

tmux

tmux[22] es un *terminal multiplexer* que permite dividir una terminal en varias ventanas o paneles, lo que facilita la multitarea así como persistir las sesiones con comandos en ejecución al cerrar el terminal.

Podemos instalar tmux en Debian/Ubuntu con el siguiente comando:

```
sudo apt install tmux
```

A continuación, procedemos a crear y entrar *(attach)* en una sesión.

```
tmux new -s session-name # crear nueva sesion
tmux attach -t session-name # entrar en sesion
```

Una vez dentro de la sesión, podemos ejecutar cualquier comando (por ejemplo, rasa run), que persistirá al salir de la misma. Para salir de una sesión *(detach)*, debemos pulsar *crtl+b*, y a continuación *d*. Para volver a entrar en la sesión, podemos usar el comando *attach* descrito anteriormente.

En nuestro caso concreto, debemos crear dos sesiones, una para ejecutar **Rasa NLU Core** (rasa run) y otra para la ejecución de **Rasa SDK Actions** (rasa run actions). También será necesario editar el archivo *endpoints.yml* y modificar las siguientes líneas:

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

También podemos crear un nuevo archivo endpoints-local.yml e indicarlo al ejecutar *rasa run* mediante el flag - - *endpoints endpoints-local.yml*.

Adicionalmente, tendremos que instalar y configurar MySQL, PostgreSQL, Redis o MongoDB para persistir los datos del **Tracker Store**[23] de Rasa.

Recomiendo que esto sea un método temporal para hacer pruebas de producción si no se tienen conocimientos de Docker o Kubernetes, en lugar de una solución definitiva.

Kubernetes

Kubernetes[24] (también conocido como K8s) es una plataforma de código abierto para la orquestación y gestión de contenedores. Desarrollado por **Google** y posteriormente donado

[22] https://github.com/tmux/tmux/wiki
[23] https://rasa.com/docs/rasa/tracker-stores/
[24] https://kubernetes.io/es/

a la **Cloud Native Computing Foundation (CNCF)**[25]**,** que forma parte de The Linux Foundation.

Es el método de despliegue recomendado[26] por Rasa. De hecho, han publicado un **Helm Chart**[27] para facilitar el despliegue haciendo uso de dicha tecnología.

La principal razón de la elección de Docker sobre Kubernetes es que uso a diario la primera y mi experiencia con la segunda es muy limitada, siendo las características y funciones ofrecidas por Docker suficientes para el estado de desarrollo actual del proyecto. Bien es cierto que en escenarios de gran demanda, Kubernetes está preparado para escalar adecuadamente, por lo que se incluye como mejora a futuro en el roadmap7.

[25] https://www.cncf.io/
[26] https://rasa.com/docs/rasa/deploy/introduction
[27] https://github.com/RasaHQ/helm-charts/tree/main/charts/rasa

Apéndice D

Manual de usuario

A continuación se incluye una breve guía de usuario que deberá ser actualizada cuando se añadan nuevas funcionalidades a nuestro chatbot.

D.1. Iniciando la conversación con el bot

En primer lugar, como se muestra en la figura D.1 debemos buscar el bot **OpenDataGenAI** en el buscador de chats integrado en nuestra aplicación de **Telegram.**

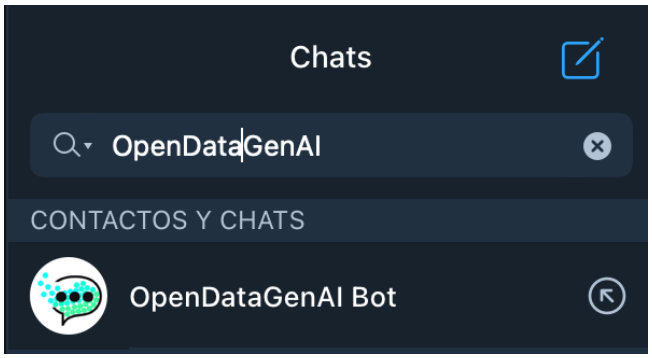


Figura D.1: Buscando el bot OpenDataGenAI Bot en el buscador de Telegram.

Finalmente, iniciamos una conversación pulsando en el botón iniciar. En la figura D.2 podemos ver el estado del chat antes de iniciar la conversación.

Una vez iniciada la conversación, el chatbot nos dará la bienvenida y podremos realizar alguna de las acciones disponibles.

D.2. Acciones disponibles

Nuestro chatbot nos permite realizar diversas acciones:

- buscar un conjunto de datos
- descargar un archivo especificando el formato deseado
- imprimir una gráfica

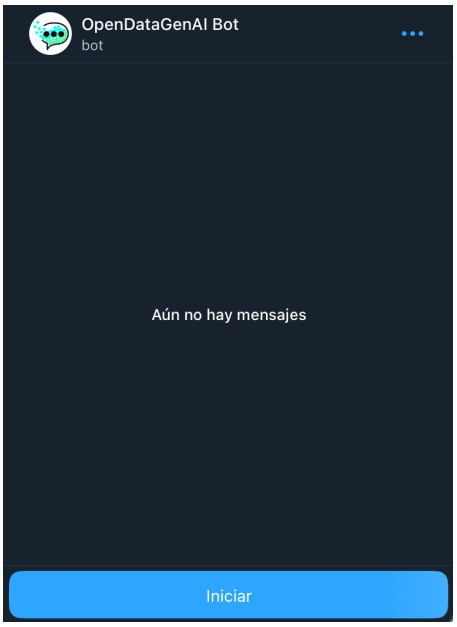


Figura D.2: Inicio de conversación con el chatbot.

- obtener una descripción con información general de los datos
- obtener datos estadísticos
- realizar una consulta personalizada

Menú de comandos

Como podemos ver en la figura D.3 y la figura D.4, disponemos de un menú de comandos integrado en la aplicación de Telegram, mediante el que podremos acceder a las diferentes acciones de nuestro chatbot sin usar lenguaje natural.

Figura D.3: Menú de comandos en Telegram (sin desplegar).

Adicionalmente, una vez tengamos un conjunto de datos seleccionado seleccionado, nuestro chatbot nos mostrará un menú contextual con las acciones disponibles, como podemos ver en la figura D.5.

Buscar un conjunto de datos (/search_data)

La primera acción que podemos realizar, es buscar un conjunto de datos. La acción se inicia escribiendo un mensaje al chatbot, por ejemplo, *"Busco datos sobre presupuestos"*.

A continuación, como vemos en la figura D.6 ,el bot nos responde con un menú que nos permite elegir uno de los conjuntos de datos disponibles.

Figura D.4: Menú de comandos desplegado en Telegram.

Descargar archivo (/download_data)

Nuestro chatbot también permite descargar un archivo con la información de un conjunto de datos en diferentes formatos. Para ello, podemos seleccionar la opción específica en uno de los menús, **"Descargar archivo",** o solicitarlo utilizando lenguaje natural, por ejemplo, enviando un mensaje como *"Quiero descargar datos sobre deuda en CSV"*, tras lo que el bot nos mostrará los conjuntos de datos disponibles para esos términos de búsqueda.

Si no se ha especificado el formato, o el formato especificado no está disponible para ese conjunto de datos, el chatbot nos mostrará un menú con los formatos disponibles como podemos ver en la figura D.7.

Finalmente, el bot nos enviará el archivo solicitado, como se muestra en la figura D.8.

Imprimir una gráfica (/plot_data)

Para generar una gráfica, de forma análoga a las acciones anteriores, podemos solicitar la acción al bot mediante un mensaje o seleccionar la opción correspondiente en el menú contextual **"Mostrar gráfico"** o menú de comandos /**plot_data**.

Un ejemplo de mensaje sería *"Muéstrame una gráfica sobre deuda"*.

En la figura D.9 podemos ver un ejemplo de gráfica generada por nuestro chatbot.

Describir el conjunto de datos (/explain_data)

Nuestro chatbot puede analizar el conjunto de datos seleccionado y generar una descripción de forma dinámica, que será vigente aunque los datos se actualicen de forma frecuente.

Figura D.5: Menú con acciones disponibles en la conversación.

Figura D.6: Menú de selección de conjunto de datos.

Para ello realizar una consulta mediante un mensaje, por ejemplo, *¿Podrías explicar los datos?*, o seleccionar la opción adecuada en uno de los menús disponibles.

En la figura D.10, podemos ver un ejemplo de generación de descripción.

Mostrar datos estadísticos (/statistics_data)

Para mostrar datos estadísticos podemos usar lenguaje natural, como *"Muéstrame estadísticas sobre deuda"* o pulsar la opción correspondiente en los menús.

Con ello nuestro bot calculará diferentes valores estadísticos como la media, mediana, moda, desviación estándar en el caso de que se puedan calcular sobre el conjunto de datos, o nos indicará la razón por la que no pueden ser calculados en caso contrario.

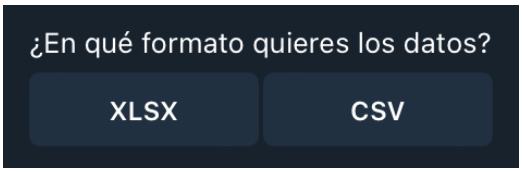


Figura D.7: Menú de selección de formato.

Figura D.8: Archivo solicitado disponible en la conversación.

Consulta personalizada (/custom_query_data)

Por último, podemos realizar una consulta personalizada en lenguaje natural sobre el conjunto de datos. Para ello debemos tener un conjunto de datos seleccionado y pulsar en la opción **"Consulta personalizada"** del menú contextual o solicitarlo con lenguaje natural escribiendo en el chat *"Quiero hacer una consulta personalizada"*.

Una vez iniciada la acción, el chatbot nos solicitará la consulta a realizar.

Ejemplos de consultas válidas serían:

- *¿Cuántos arquitectos hay en plantilla?*. En un conjunto de datos de personal.
- *¿Cuál es el valor más alto?* En un conjunto de datos sobre pago a proveedores.

En la figura D.11 podemos ver un ejemplo de respuesta a una consulta personalizada.

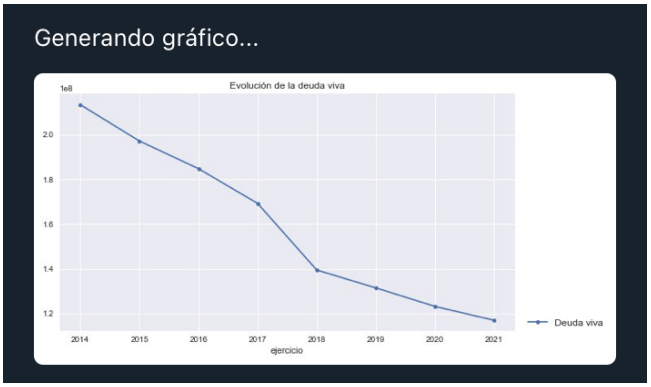


Figura D.9: Ejemplo de generación de gráfica.

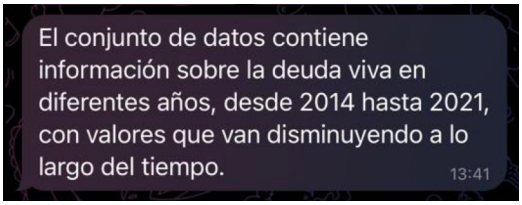


Figura D.10: Descripción generada por el bot.

Figura D.11: Respuesta del chatbot a consulta personalizada.