EL TROL VERSÁTIL
THE VERSATILE TROLL

ENRIQUE BALLESTEROS HORCAJO
SERGIO CALERO ROBLES
JAIME FERNANDEZ DÍAZ
VENTURA MATEOS PÉREZ

4º CURSO. TRABAJO DE FIN DE GRADO. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de grado del Grado en Ingeniería Informática

Fecha

11 de enero de 2022

Profesor Director:

MANUEL NÚÑEZ GARCÍA

Autorización de difusión

Autores

Enrique Ballesteros Horcajo
Sergio Calero Robles
Jaime Fernández Díaz
Ventura Mateos Pérez

Fecha

11 de enero de 2022

Los arriba firmantes, matriculados en el Grado en Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: “El Trol Versátil”, realizado durante el curso académico 2021-2022 bajo la dirección de Manuel Núñez García en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Las distintas redes sociales ofrecen cada vez mejores herramientas para la automatización de la interacción de los usuarios. Así el mercado de interacciones en redes sociales es cada vez es más amplio y tiene más competencia. En este contexto se hace necesario el uso de herramientas para la automatización de las interacciones en redes sociales que sean diferenciales.

El fin de este trabajo es ofrecer un servicio para la creación de redes de bots que interaccionen automáticamente con cuentas de la red social *Twitter* de forma configurable y personalizada para cada cliente. La interacción debe ser coherente y difícilmente distinguible de la que realizaría un usuario real.

Palabras clave

Inteligencia Artificial, Aprendizaje Automático, Bots, Redes Sociales, Twitter, Frontend, Backend

Abstract

The social networks offer every day better tools for automating user interaction. Therefore, the market of social network interactions is becoming larger and larger each day. In this context it is necessary to use tools for the automation of interactions in social networks that are differential.

The purpose of this project is to offer a service for the creation of bot networks that automatically interact with accounts of the social network *Twitter* in a configurable and personalized way for each client. The interaction must be coherent and hardly distinguishable from what a real user would do.

Keywords

Artificial Intelligence, Machine Learning, Bots, Social Networks, Twitter, Frontend, Backend

Índice general

Índice I

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Objetivo . . . 1
- 1.3. Estructura de la memoria . . . 2

1. Introduction 3
- 1.1. Motivation . . . 3
- 1.2. Objective . . . 3
- 1.3. Memory structure . . . 4

2. Estado del Arte 5
- 2.1. Principales empresas web de habla hispana de servicios para redes sociales . . . 5
- 2.2. Estado del mercado internacional . . . 7
- 2.3. Conclusiones sobre el estado del arte . . . 8

3. Business Plan 9
- 3.1. Modelo del servicio . . . 10
- 3.2. Gastos fijos de la empresa . . . 11
  - 3.2.1. Salarios . . . 11
  - 3.2.2. Coste de los servidores . . . 12
- 3.3. Modelo de negocio sostenible . . . 12
- 3.4. Plan de negocio . . . 13
  - 3.4.1. Modelo de negocio sin necesidad de buscar financiación externa . . . 13
  - 3.4.2. Modelo de negocio con inversión externa . . . 14
- 3.5. Plan de marketing . . . 15
- 3.6. Análisis de debilidades, amenazas, fortalezas y oportunidades . . . 16
  - 3.6.1. Debilidad . . . 16
  - 3.6.2. Mayor amenaza: mes con balance negativo . . . 17
  - 3.6.3. Fortalezas, amenazas y oportunidades de negocio . . . 17
- 3.7. Conclusiones del plan de negocio . . . 18

4. Propuesta y diseño 19
- 4.1. Propuesta de interfaz para el usuario . . . 19
- 4.2. Modelos de generación de texto . . . 22
  - 4.2.1. Implementación . . . 23

4.2.2. models.py 24
4.2.3. responseGeneration.py 27
4.3. Análisis de sentimiento 29
4.4. Bots 30
4.4.1. Implementación 30
4.4.2. Aprendizaje de modelos 31
4.4.3. Selección de modelo para generar una respuesta 31
4.4.4. Creación e interacción con un Bot 31
4.5. Filter Parameters 32
4.5.1. Filter Parameters en los scripts 33
4.6. Inferencia de parametros de filtrado 33
4.6.1. Inferencia de *Positivity Factor* 33
4.6.2. Inferencia de *keywords* 34
4.6.3. Inferencia de *Slang Factor* 34
4.7. Base de datos de relaciones 34
4.7.1. Implementación 35
4.7.2. factsGeneration.py 35
4.8. Unificación de entidades 36
4.9. Scripts 36
4.10. Generador de corpus 37
4.11. Setup e instalación de entorno Python 38

5. Arquitectura e Implementación **39**
5.1. Arquitectura de la aplicación frontend 39
5.1.1. Consideraciones previas en el desarrollo de una APP 39
5.1.2. Estructura básica 40
5.1.3. Patrón del flujo de datos 41
5.1.4. Inicialización de la aplicación frontend 43
5.1.5. Diagramas de clases de las Stores y Models 43
5.1.6. Diagramas de clases de los Services 45
5.2. Arquitectura del servidor de NodeJS 46
5.2.1. Consideraciones previas 46
5.2.2. Estructura básica 46
5.2.3. Flujo de enrutado de las peticiones HTTP 47
5.2.4. Diagramas de clases de los Controllers, Helpers y Models 49
5.2.5. Diagramas de clases de los Services 57
5.3. Arquitectura de los scripts de Python para la utilización de Twitter 58
5.3.1. Consideraciones previas para la integración de Twitter en Python . . 58
5.3.2. Estructura básica y patrón de datos 59
5.3.3. Arquitectura de la base de datos de la interacción con Twitter 59

6. Casos de Estudio 60
6.1. Aplicando la solución en un entorno propuesto . . . 60
6.2. Caso de uso: Crear y entrenar una nueva trollnet . . . 61
6.2.1. Fase 1: Frontend Web - procesar el input del usuario . . . 61
6.2.2. Fase 2: Servidor Web - iniciar la creación de la trollnet . . . 63
6.2.3. Fase 3: Ejecución Python - creación de un bot . . . 63
6.2.4. Fase 4: Servidor Web - control y fin de la creación de la trollnet . . . 63
6.2.5. Fase 5: Ejecución Python - vincular cuentas a los bots . . . 64
6.2.6. Fase 6: Servidor Web - iniciar proceso de entrenamiento de la trollnet 64
6.2.7. Fase 7: Ejecución Python - entrenamiento de un bot . . . 65
6.2.8. Fase 8: Servidor Web - control y fin del entrenamiento de la trollnet . 65
6.2.9. Fase 9: Frontend Web - chequeo periódico del progreso . . . 66
6.3. Caso de uso: Entrenar un modelo . . . 66
6.3.1. Fase 1: Servidor Web - detección del corpus y comienzo del proceso . 66
6.3.2. Fase 2: Ejecución Python - entrenamiento de un modelo a partir de un corpus . . . 67
6.3.3. Fase 3: Servidor Web - actualización de la MDL y control del proceso 68
6.4. Caso de uso: Generar una conversación . . . 68
6.4.1. Fase 1: Frontend Web - procesar el input del usuario . . . 68
6.4.2. Fase 2: Servidor Web - preparación inicial y petición de escucha . . . 69
6.4.3. Fase 3: Ejecución Python - escuchar la próxima publicación de la cuenta objetivo . . . 69
6.4.4. Fase 4: Servidor Web - inicio del proceso de conversación . . . 70
6.4.5. Fase 5: Ejecución Python - generación de respuesta . . . 70
6.4.6. Fase 6: Servidor Web - recoger respuesta y lanzar flujo de publicación 70
6.4.7. Fase 7: Ejecución Python - publicación de respuesta en Twitter . . . 71
6.4.8. Fase 8: Servidor Web - control y fin del proceso . . . 71

7. Conclusiones 72
7.1. Inteligencia de un bot . . . 72
7.1.1. Generación de texto . . . 72
7.1.2. Base de datos de relaciones . . . 73
7.1.3. Inferencer . . . 74
7.1.4. Análisis de sentimiento . . . 74
7.1.5. Conclusiones de la sección . . . 74
7.2. Conclusiones generales . . . 75
7.2.1. Líneas de trabajo futuro . . . 76

7. Conclusions 77
7.1. Bot's intelligence . . . 77
7.1.1. Text generation . . . 77
7.1.2. Relationships database . . . 78
7.1.3. Inferencer . . . 78

7.1.4. Sentiment analysis 79
7.1.5. Section conclusions 79
7.2. Main conclusions 79
7.2.1. Future work 80

8. Contribuciones **81**
8.1. Enrique Ballesteros Horcajo 81
8.2. Sergio Calero Robles 83
8.3. Jaime Fernández Díaz 85
8.4. Ventura Mateos Pérez 86

Bibliography **90**

A. Imágenes del Estado del Arte y del Business Plan **91**

B. Imágenes de la Arquitectura e Implementación y de los Casos de Uso **100**

Capítulo 1

Introducción

En este capítulo se expondrán de forma breve las motivaciones y objetivos del equipo del proyecto a la hora de realizar este Trabajo de Fin de Grado (TFG), así como una pequeña guía sobre la estructura de esta memoria.

1.1. Motivación

Las redes sociales continúan creciendo a un ritmo constante y cada vez más personas se adaptan a su uso. Lo que en el pasado podría ser una simple tarjeta con un número de teléfono, hoy es un perfil online con varios enlaces a las distintas cuentas que un usuario tiene en diversas redes sociales. Para muchos, se han convertido hasta en una necesidad, en un medio indispensable para estar al tanto de lo que ocurre a su alrededor, o para comunicarse con los suyos. Y no es sólo eso, también son un gran medio para publicitar multitud de servicios, y compartir opiniones sobre los mismos. Por otro lado, la Inteligencia Artificial ha mejorado mucho en los últimos años, habiéndose desarrollado multitud de técnicas muy potentes y novedosas para, por ejemplo, el *Machine Learning*, una tecnología de gran utilidad. Y es aquí donde uno se pregunta, ¿por qué no hacer uso de estas revolucionarias herramientas para crear un servicio que aproveche el gran impacto que tienen las redes sociales hoy en día? Esto mismo, junto a una leve curiosidad de ver hasta qué punto se pueden aprovechar estas técnicas, es lo que ha motivado al desarrollo de la aplicación que se presenta en este TFG, mediante la cual cualquier usuario o empresa podrá contratar a unos bots para que discutan y opinen sobre los temas que ellos deseen, generando interacciones en redes sociales (en el caso de este proyecto, en Twitter).

1.2. Objetivo

Es esencial, por supuesto, ofrecer un servicio único e innovador, y ese es el principal objetivo de este TFG. Se pretende crear una aplicación que no simplemente genere mensajes que se parezcan a los escritos por un ser humano, sino que hable consigo misma y sus propios mensajes, tenga opiniones positivas o negativas sobre distintos temas, según la cuenta (de

Twitter, en este caso) que se esté usando para hablar. Se quiere desarrollar un servicio innovador que otorgue una personalidad totalmente única y nueva a cada bot, que no es ni más ni menos que un usuario cuyos mensajes serán escritos automáticamente. De esta forma, no sólo parecerán más humanos, sino que aportará variedad a la red de respuestas y opiniones que se busca generar.

Este servicio vendrá acompañado de una interfaz configurable donde el usuario podrá seleccionar qué tipo de bots quiere solicitar, según una gran variedad de parámetros. Además de desarrollar la interfaz de usuario y la inteligencia de los bots para escribir mensajes, también se tendrá que preparar una base de datos, donde se almacenarán multitud de datos necesarios para el mantenimiento del servicio, como las distintas personalidades de los bots existentes, los modelos que se usarán como base para el aprendizaje de los mismos (los cuales también habrá que obtener por separado) y las distintas relaciones y conceptos que vayan aprendiendo los bots. En resumen, se busca ofrecer un servicio a empresas que genere interacciones automáticas en redes sociales, en este caso Twitter, aunque esta tecnología podría ser fácilmente expandida a otras redes.

La elección de Twitter simplemente ha resultado ser la más conveniente. Twitter es una red social de blogs, pero con la peculiaridad de que los artículos publicados están limitados en el número de caracteres. Se autodefinen como un servicio para amigos, familia y trabajadores para comunicarse y permanecer conectados a través del intercambio rápido y frecuente de mensajes. Twitter está orientado a la publicación de mensajes de texto, aunque permite incluir fotos, vídeos, enlaces y otros contenidos habituales de redes sociales. Los mensajes publicados por un usuario de Twitter son leídos por el resto de usuarios suscritos y pueden ser compartidos o contestados por cualquier usuario de la plataforma. El servicio presentado en este TFG sirve para responder a e interaccionar con las publicaciones del usuario que lo contrate.

1.3. Estructura de la memoria

La memoria comienza con una sección donde se analiza la posible competencia que pudiera surgir a un servicio como el que se ha propuesto. Se observa como no existen realmente empresas que ofrezcan un servicio similar a un precio razonable. A continuación, en el tercer capítulo se explica detalladamente el plan de negocios de la empresa de este proyecto, analizando cómo será desarrollado el proyecto por los cuatro miembros del equipo, sus salarios, costes, debilidades, etc. La explicación de la aplicación en sí comienza en el capítulo 4, donde viene detallado el funcionamiento del aprendizaje de los bots y la generación de mensajes de los mismos. Seguidamente se expone la arquitectura empleada para desarrollar la interfaz de usuario, el servidor NodeJS empleado y los scripts usados para la conexión con Twitter. En el capítulo seis se desarrollan los casos de uso con los que podría toparse un usuario de la aplicación. Por último, en la sección siete se recopilan unas breves conclusiones del proyecto y, finalmente, en la sección ocho se presentan las contribuciones individuales de cada miembro del equipo.

Capítulo 1

Introduction

This chapter will briefly expose the motivations and objectives of the team project when carrying out this Final Degree Project (Undergraduate Thesis Project), as well as a short guide to the structure of this document.

1.1. Motivation

Social networks continue to grow steadily and more and more people start to use them each day. What in the past could be a simple card with a phone number, today it's an online profile with several links to different accounts that a user has on social networks. For many people, social networks have even become a need: an indispensable tool to be aware of what is happening around them and to communicate with their loved ones. Furthermore, they are also a great way to advertise a large number of services and share opinions about them on the web. On the other hand, Artificial Intelligence has improved a lot in recent years, having developed many powerful and innovative techniques. Thus this is where one wonders, why not use these revolutionary tools to create a service that can take advantage of the great impact that social networks have? This idea, together with the curiosity to know to what extent these techniques can be used, is what has motivated the development of the application. An application in which any user or company can hire bots to discuss and give their opinion on the topics they want, generating interactions on social networks and especially to improve their profile engagement in the social network *Twitter*.

1.2. Objective

It is essential to offer a unique and innovative service in the social network environment, and that is the main objective of this Final Degree Project. The objective is to create an application that not only generates messages that resemble those written by a human being, but also generates bot profiles that can talk to themselves and respond their own messages. Besides, they have positive or negative opinions on different topics and generate enough engagement in any topic so they can receive responses from real users. The development

should be an innovative service that gives a totally unique and new personality to each bot, which is a user of social networks whose messages will be written automatically, but also that nobody could identify it as a bot. In this way, they will not only appear more human, but will also add some variety to the network of responses and opinions that they seek to generate.

The service will be coupled with a configurable friendly graphical user interface where the user can select the type of bots that he wants according to a wide variety of parameters. In addition to developing the user interface and the intelligence of the bots to publish messages, there will be a central database that stores and centralizes all data necessary for the environment operation: the different personalities of the existing bots, the models that will be used as a basis for learning, the different relationships and concepts that the bots will learn and the social media accounts available for use. In summary, it seeks to offer a service that generates automatic interactions on social networks, especially for the social network *Twitter*, although this technology could be expanded to other networks.

Choosing *Twitter* has simply turned out to be the most convenient option because it is a blogging social network but with the peculiarity that published articles are limited in the number of characters. They define themselves as a service for friends, family and workers to communicate and stay connected through the exchange of messages. Although *Twitter* is not only oriented to the publication of text messages (as it also allows to include photos, videos, links and other media content) the text messages are the most commonly used. In *Twitter*, messages posted by a *Twitter* user are read by the rest of the subscribed users and can be shared or answered by any other user on the platform. The service presented in this Final Degree Project serves to respond to and interact with the publications of the user who hires it, in order to increase the number of interactions that his publications receive.

1.3. Memory structure

After this introduction, the document begins with the state of the art section, where the different competitors of services for the social network market are analysed. It is observed that there are no companies that offer a full service for social network interactions at a competitive price. In the third chapter, the business plan for this project is explained in detail, analyzing how the project will be developed by the four team members: analyzing the fixed costs, the weaknesses and strengths of the plan, elaborating a marketing plan, and in short, developing a business strategy. The explanation for the application itself begins in chapter four, where the bot learning and message generation processes are detailed. In the following chapter, the full architecture is explained: the architecture of the user interface front-end, the architecture for the NodeJS server and the scripts to connect to *Twitter* are exposed. Chapter six develops the principal use cases that an user might encounter in detail. The conclusions of the project are in chapter seven. Finally, in chapter eight are presented the individual contributions of each team member.

Capítulo 2

Estado del Arte

En el mercado actual hay numerosas empresas web que ofrecen todo tipo de servicios para mejorar el posicionamiento en redes sociales. El objeto del estudio realizado es obtener una visión completa de los principales servicios de valor añadido que ofrecen estas empresas para el aumento de la tasa de interacción en redes sociales[12].

Las redes sociales sobre las que se focalizará el estudio son las occidentales con mayor presencia de usuarios de habla inglesa o hispana, que además tienen el mayor número de usuarios activos a nivel mundial[5] y que su eje de funcionamiento central es la interacción social y conversacional: *Facebook* y *Twitter*.

El estudio estará dividido en dos partes: en una primera sección se llevará a cabo un análisis de las principales empresas que ofrecen servicios de interacción en redes sociales para el mundo hispano y en la segunda de las empresas a nivel mundial de habla inglesa del mismo sector.

En el análisis de empresas del ámbito hispano, para *Facebook* se le dará especial importancia a la oferta para compra de *Me gusta* y suscriptores de páginas empresariales. En el caso de *Twitter* el foco será la compra de seguidores y de *retuits*, las dos interacciones más comunes de esta plataforma. Para la segunda parte del estudio, realizada sobre empresas de ámbito mundial de habla inglesa se pondrá más esfuerzo en el análisis del servicio de comentarios y respuestas. Este servicio no se ofrece en el mercado hispano, y es el mayor valor diferencial de las empresas angloparlantes.

2.1. Principales empresas web de habla hispana de servicios para redes sociales

El universo de empresas dedicadas a servicios web para las redes sociales hispanohablantes ofrece todo tipo de servicios para aumentar la tasa de interacciones[12] de los compradores. Numerosas empresas que quieren aumentar el interés que generan y el abanico de usuarios a los que alcanzar, contratan estos servicios. A continuación se analizarán las principales empresas de habla hispana, para después estudiar qué servicios ofrecen y qué precios exigen.

La empresa más completa del mercado hispano es *COMPRASOCIALMEDIA*[1], propietaria de la web *compra-seguidores.com*, recomendada por *Encarni Arcoya* de *ActualidadeCommerce* para aumentar todo tipo de interacciones en las principales redes sociales.

Para los *Me gusta* de *Facebook* ofrecen cuatro paquetes (véase figura A.2 del Apéndice A), el más pequeño de 1.000 *Me gusta* y el más grande de 5.000. El coste por *Me gusta* está entre los 0,039 euros y los 0,032 euros. Para la compra de suscriptores también ofrecen cuatro paquetes diferentes, aunque en este caso el mayor ofrece 10.000. El abanico de precios en este caso va de los 0,039 euros a los 0,03.

En su oferta para retuits de la red social *Twitter* disponen de cuatro paquetes, el más pequeño de 1.000 *retuits* tiene un precio de 0,013 euros la interacción y el más grande de 10.000 ofrece un precio de 0,0086 euros por retuit. Para la compra de seguidores también ofrecen cuatro paquetes diferentes, aunque en este caso el mayor ofrece 10.000. El abanico de precios (véase figura A.1 del Apéndice A), va desde los 0,01 euros el seguidor hasta los 0,0085 euros por seguidor en su paquete más económico.

La empresa *FollowersYA*, también mencionada por *ActualidadeCommerce*[1] entre las cinco mejores empresas para invertir en la red, tiene una oferta también muy completa de servicios disponibles.

Centrada en el mercado hispano, garantiza la procedencia hispano-hablante de todas sus interacciones. En servicios para *Facebook* ofrece tres modalidades con cuatro paquetes cada una para el servicio de compra de *Me gusta*, en la modalidad más económica ofrece un precio mínimo de 0,013 euros el *Me gusta*, en la modalidad más costosa (véase figura A.4 en el Apéndice A) garantiza procedencia de España por un precio de hasta 0,42 euros el *Me gusta*. Esta diferencia refleja claramente margen de beneficio que puede haber si se dispone de un servicio de gran valor añadido.

Respecto a *Twitter* su oferta garantiza retuits de cuentas reales por 0,25 euros cada uno en su paquete más sencillo y 0,17 en su mejor paquete. Respecto a la compra de seguidores también garantizan la calidad de los mismos y lo precios van desde los 0,042 euros hasta los 0,025. Esta diferencia entre precios claramente ilustra la dificultad de conseguir ofrecer ciertos servicios.

La empresa *COMPRASEGUIDORES* (distinta a la anteriormente mencionada web de la empresa *COMPRASOCIALMEDIA*), referida por el *Huffingtonpost*[9] como la primera en el mercado de habla hispana para la compra de seguidores en la plataforma *Twitter*, centra su mercado en la compra de seguidores a un precio de mercado muy competitivo. Para el desarrollo de su negocio priman la oferta en cantidad de seguidores por encima de la calidad o la variedad.

La oferta de servicio para *Facebook* está centrada en suscriptores a un precio muy competitivo, disponen de seis paquetes que ofrecen cada suscriptor desde los 0,01 euros en su paquete más pequeño hasta los 0,005 en su paquete de 100.000 suscriptores. Resulta evidente que el objetivo es alcanzar a clientes que opten por una opción económica que engrose sus números, sin esperar una gran calidad en cada suscriptor.

Para la oferta de *Twitter* ofrecen seguidores desde los 0,022 euros hasta los 0,016 euros por cada seguidor. Los paquetes de retuits directamente no están temporalmente disponibles en la actualidad.

Se puede concluir que el coste de interacciones para hispano-hablantes varía mucho según la calidad del servicio ofrecido y especialmente del volumen que se contrate, alcanzando descuentos de hasta un cincuenta por ciento.

2.2. Estado del mercado internacional

En el mercado angloparlante se encuentra una opción aún más numerosa de empresas online que ofrecen directamente al comprador múltiples servicios para aumentar la tasa de interacción en redes sociales. En este la oferta de servicios es mucho más amplia que en el mercado de habla hispana. Destaca la opción de interacción conversacional en las redes sociales y debido a que es un servicio diferencial, el análisis que se va a realizar se centrará en ello.

La empresa *SOCIALFORMING:* es la que que ofrece las respuestas personalizadas por el mayor precio del mercado, el servicio cuesta 0,84 euros por cada respuesta personalizada que se requiera y es el mismo para las dos redes sociales del estudio (*Facebook* y *Twitter*). El cliente debe remitir por correo electrónico cada una de las respuestas personalizadas que quiere recibir en su perfil. Los paquetes que tienen en oferta no incluyen ningún descuento por cantidad, es un precio único fijo (véase la figura A.7 en el Apéndice A). Estos precios y la falta de descuentos por compra de volumen vienen propiciados por la necesidad de introducción manual de estas respuestas, requiriendo trabajo manual para cada interacción. Además, la necesidad de que el cliente deba redactar la respuesta resta valor al servicio, ya que requiere esfuerzo por parte del cliente y un constante trabajo de análisis y redacción.

En su presentación instan al usuario a comparar precio y valoraciones de distintas empresas, asegurando que por su parte ofrecen la mejor calidad-precio del mercado.

La segunda web del análisis será *BUYVIEWSLIKES*. Su oferta garantiza expresamente la calidad de las respuestas y de los perfiles que participan en las interacciones. Su opción de venta del servicio respuestas las únicas dos opciones que ofrecen son o bien que las escriba el propio cliente personalmente y las remita por correo electrónico o que sea una interacción positiva relacionada con el tema que se comenta. No ofrece por tanto la opción de generar polémica, discutir entre perfiles o crear algún tipo de debate. Es la misma oferta tanto para *Twitter* como para *Facebook*, alcanzando un precio de entre 0,5 y 0,2 euros por interacción según el paquete contratado.

Si bien añade la opción de que los comentarios sean automáticos, al ser comentarios favorables a las publicaciones no ofrecen el mismo aumento de la tasa de interacción que otro tipo de respuestas, ya que esto comentarios no incitan a la participación por parte del resto de usuarios de las plataformas.

Es necesario también destacar también la web *PAYSOCIALMEDIA*, con gran presencia en el mundo de los servicios para redes sociales por su campaña agresiva anunciándose en *Google Ads*, es la empresa que más facilita la compra del paquete de interacciones conversacionales que se requiera. Ofrece comentarios tanto para *Twitter* como para *Facebook* y permite seleccionar paquetes dedicados a mensajes, respuestas positivas y comentarios favorables o paquetes para interacciones definitas ad hoc por el cliente. Esto permite tener

un control completo y sencillo sobre lo que se quiere contratar. El precio es de 0,27 euros por comentario positivo automático o de 0,42 euros por comentario redactado por el propio cliente. Resulta muy interesante esta opción de poder elegir qué tipo de comentario puede comprarse, dando una sensación de profesionalidad a la página.

Disponen también de una opción más económica que ofrece 100 respuestas por 17,87 euros. Sin embargo, no debe olvidarse que es un servicio en inglés y que se ofrece en dólares, por lo que el precio en euros es al cambio actual (en el momento de la redacción 1 euro equivale a 1,13 dólares) y en un futuro podría variar. Aún así, ninguno de sus paquetes se acerca a los 0,15 euros por interacción.

Finalmente queda analizar el primer resultado que coloca *Google* en su buscador para la compra de comentarios y respuestas en redes, este es la empresa *BULKCOMMENTS*. Se centra en ofrecer comentarios y respuestas para redes sociales, como puede intuirse por su nombre. Ofrece respuestas interaccionales para *Twitter* y *Facebook* por un precio muy competitivo comparado con el resto de ofertas: disponen de cinco paquetes dependiendo del volumen para los que el precio va desde los 0,2 euros hasta los 0,3 euros por respuesta interaccional. Insisten en la calidad de las respuestas y comentarios y garantizan que no utilizan bots, que las interacciones se realizan desde cuentas por todo el mundo y desde diferentes direcciones IPs.

En el mercado anglosajón por tanto se encuentra una media de 0,515 euros la interacción en los paquetes más caros del mercado y de 0,35 si se analizan las opciones más económicas. La opción más barata que se ha encontrado es la de *PAYSOCIALMEDIA* que ofrece un paquete en el que cada interacción se encuentra por un precio de 0,17 euros.

2.3. Conclusiones sobre el estado del arte

En el estudio realizado sobre las empresas de habla hispana se han detectado grandes carencias en los servicios, comparándolos con los servicios de las empresas de habla inglesa líderes a nivel mundial es evidente que hay una gran distancia entre ambas. No solo a nivel de precios del servicio, de marketing y colocación de producto. Se ha observado que los productos que ofrecen las empresas angloparlantes son más completos y además ofrecen servicios de valor añadido que las hispano-hablantes no tienen en su catálogo. Destaca especialmente la oferta de respuestas y comentarios en redes sociales, ya que en las empresas de habla hispana no disponen de este servicio, que sin embargo aparece en todas las de habla inglesa.

Especialmente destacan las ofertas de respuestas positivas a favor de ciertas publicaciones señaladas por el cliente, un servicio que solo se ofrece en inglés y con precios muy elevados.

Se concluye, por tanto, que si bien existe un núcleo de servicios que ofrecen todas las empresas dedicadas a la venta de servicios para la mejora de la tasa de interacción en redes sociales, en la actualidad no hay ninguna empresa líder del mercado que ofrezca respuestas y comentarios a precios competitivos. Parece así que hay una falta clara de oferta de servicios de respuestas interaccionales complejas que generen debate e interacciones adicionales a precios competitivos y sin la necesidad de una participación continua del cliente.

Capítulo 3

Business Plan

A continuación se presenta el plan de desarrollo del negocio para la empresa de servicios de valor añadido en redes sociales. En este plan de negocio se orientará la oferta de servicios propuesta a interacciones conversacionales en redes sociales, con el objetivo de cubrir la falta de oferta que se ha podido detectar en el análisis del estado del arte presentado en el capítulo 2. El servicio tendrá un precio competitivo y una oferta de servicio de valor añadido que los competidores no tienen: respuestas que generen debate y nuevas interacciones de otros usuarios sobre las conversaciones en las que aparezcan. Se presentará un modelo de servicio al cliente similar al que ofrecen todas las empresas del sector, basado en la compra de paquetes de interacciones.

La plantilla de la empresa estará formada por cuatro ingenieros informáticos que trabajarán a jornada completa durante todo el año, en remoto y utilizando sus propios equipos informáticos para reducir los costes todo lo posible. El servicio estará completamente alojado en la nube de *Microsoft*, en *Azure*, eliminando así la necesidad de inversión inicial en infraestructuras.

En la estrategia de marketing se manejará una opción de búsqueda de capital a través de aceleradoras de *Startups* españolas, y otra opción alternativa que no requiere de inversión de capital externo.

Se analizarán las debilidades y puntos fuertes del negocio y de las estrategias de marketing y financiación que han sido seleccionadas.

En el plan de negocio propuesto se plantearán diversas estimaciones de evolución financiera en función de los puntos débiles que han sido detectados. Los objetivos principales serán alcanzar una situación de equilibrio financiero y el retorno completo de la inversión inicial.

3.1. Modelo del servicio

El modelo de servicio se plantea de forma similar al que ofrecen todas las empresas del sector: el cliente pagará por cada interacción conversacional que reciba su perfil y realizará la contratación de estas interacciones mediante la compra de distintos paquetes predefinidos que le ofrecerán reducciones en el precio dependiendo del volumen de interacciones que contrate:

- Pack *Prueba:* es el paquete básico y gratuito que se ofrece a los clientes que quieran probar el servicio. Incluye 10 interacciones conversacionales que podrá utilizar como quiera.
- Pack *Semanal:* es el paquete pensado para empresas que contratan por primera vez los servicios y quieran realizar pruebas complejas con el servicio. También está pensado para pequeñas empresas que no quieran realizar una gran inversión inicial. Ofrece el peor coste por interacción, 0,36 euros por respuesta que se emita.
- Pack *Estándar:* el paquete pensado para empresas medianas, ofrece 10 interacciones diarias durante un mes a un precio de 300 euros. Se espera que sea la opción contratada inicialmente por los clientes y que posteriormente la mayoría evolucionen a la opción *PRO*.
- Pack *PRO:* el objetivo de la empresa es que todo cliente importante contrate mensualmente este paquete, garantizando así la sostenibilidad de la empresa mediante clientes fieles al servicio.
- Pack *Premium:* se ofrecerá a los grandes clientes que excedan las 1.000 interacciones mensuales. Ofrece el precio más competitivo del mercado por interacción conversacional, y se espera que sea un buen reclamo para atraer grandes clientes.

Si bien el cliente podrá configurar el número de interacciones que recibirá por publicación, el mínimo de interacciones que se realizarán siempre que el cliente tenga contratado algún paquete será de mínimo 10 por día, 70 semanales o 300 mensuales. Necesario para asegurar que cada mes se cubran los costes que genera cada cliente. Como opción alternativa se ofrecerá un servicio compartido con otros cliente que no garantizará la misma calidad del servicio dedicado pero que dejará libertad en la velocidad del consumo de las interacciones compradas.

El objetivo de estos precios es ofrecer un servicio único por un precio competitivo muy complicado de igualar por las empresas existentes que utilizan metodologías manuales o automatizaciones muy elementales.

La contratación del paquete *Premium* será adicional a la contratación del paquete *PRO* y requerirá de un plan específico aprobado por la empresa.

Pack contratado	Coste total	Interacciones contratadas	Coste individual
Prueba	0€	10	0€
Semanal	25€	70	0,36€
Estándar	100€	300	0,33€
PRO	1.000€	5.000	0,20€
Premium	0,15€	1	0,15€

Cuadro 3.1: *Paquetes ofertados a cliente*

Coste salarios mensuales	Coste salarios anual
8.491,47€	101.897,6€

Cuadro 3.2: *Salarios*

3.2. Gastos fijos de la empresa

En los gastos que la empresa tendrá todos los meses se han incluido los salarios de los trabajadores y el coste de la infraestructura de computación, en este caso servidores virtuales en la nube de *Microsoft*.

3.2.1. Salarios

En una estimación inicial de costes la empresa deberá pagar el salario de los cuatro integrantes del equipo actual (véase figura A.11 del Apéndice A), que percibirán el salario bruto estimado para un ingeniero informático recién egresado. Este salario se ha fijado en 1.400 euros brutos basado en los valores medios que publica la empresa de contratación *Jobted*[7].

Una vez fijado el salario bruto que percibirán los empleados, se añade una estimación de los distintos impuestos que se deberán pagar por estas contrataciones basados en las estimaciones de *Marina Camacho* en el blog *Factorial*[2]: si bien puede variar según el convenio colectivo, se estima el gasto total por mes en 1.819,6 euros por ingeniero informático. Debe tenerse en cuenta además que la retribución anual será de 14 pagas, por lo que el gasto por ingeniero final será de 25.474,4 euros al año.

3.2.2. Coste de los servidores

El precio mensual de cada servidor necesario se ha obtenido tomando como referencia los costes aproximados que se indican en la web de *Azure* en función de los parámetros que requieren los bots, y se han comprobado mediante un piloto de prueba que se ha ejecutado en un servidor contratado utilizando la opción de prueba gratuita del servicio.

Los requerimientos de los servidores se han obtenido mediante pruebas de carga en un servidor virtual piloto creado en *Azure*, junto con las pruebas realizadas en los equipos locales. Los requerimientos en el número de servidores vienen por la necesidad de un servidor central, un servidor para los clientes en prueba y un servidor dedicado para cliente que contrate paquetes no gratuitos. En las pruebas de carga se ha estimado que es suficiente con un servidor dedicado por cliente:

Habrá por tanto en todo momento un servidor dedicado a la centralización de la información de los bots, al entrenamiento de estos a partir de los distintos modelos y peticiones de cliente, cuyo requerimiento es un mínimo de 8 GB de memoria RAM de acuerdo a las pruebas de carga realizadas. Con estos parámetros y utilizando la calculadora de *Microsoft*[8] para máquinas virtuales desplegadas en *Azure*, se estima un coste de 73,92 euros al mes por máquina(véase figura A.12 del Apéndice A).

Se proveerá además otro servidor de las mismas características donde se ejecutarán todos los bots conversacionales de planes gratuitos y donde se ejecutará la web de la propia empresa. Al ser el plan gratuito, las interacciones no serán muy numerosas, por lo que se estima que con uno será suficiente en todo momento. Hay que tener en cuenta que el plan gratuito estará limitado en tiempo para cada cliente. En este servidor adicionalmente podrán correrse bots de aquellos clientes que requieran pocas interacciones diarias, esto es menos de 10 interacciones que no suponen una carga excesiva para el mismo.

Adicionalmente, por cada cliente que contrate cualquier otro plan se provisionará una nueva máquina dedicada para el mismo. Esto es con el objetivo de que cualquier prueba, modificación o cambio sobre un cliente no afecte al resto de clientes importantes.

3.3. Modelo de negocio sostenible

Teniendo como objetivo principal encontrar un equilibrio económico sostenible para la empresa, la meta es que los ingresos por clientes de paquetes *PRO* (véase el cuadro 3.1) superen al conjunto de gastos de la empresa. El resto de paquetes tienen como objetivo que el cliente acabe contratando el paquete *PRO*, por lo que el resto de paquetes no se tendrán en cuenta para el balance contable. Por tanto se estima una necesidad mínima mensual de 10 clientes *PRO* de acuerdo a las estimaciones que pueden verse en la figura A.13 del Apéndice A. Para dar servicio a este número de clientes se requerirán 12 servidores basados en los test de carga realizados, que junto con los distintos gastos añadidos por personal (véase la figura A.11 del Apéndice A) arrojan los datos finales del balance sostenible: 7.508,53 euros (véase la figura A.13 del Apéndice A).

Este es el objetivo mínimo que se plantea conseguir a un año vista de cara a la sosteni-

bilidad futura de la empresa.

3.4. Plan de negocio

En el desarrollo del modelo de negocio se plantean dos escenarios posibles: un primer escenario sin necesidad de ninguna inversión de capital externo para el funcionamiento de la empresa (véase sección 3.4.1): para ello será necesario contar con el consenso de todos los integrantes de la empresa y la colaboración de empresas del sector del negocio de la interacción en redes sociales. Un segundo escenario en el que se buscará financiación externa (véase sección 3.4.2): para este escenario contemplaremos la opción de financiarnos a través de aceleradoras o inversores de *Startups*.

3.4.1. Modelo de negocio sin necesidad de buscar financiación externa

Se plantea un modelo de negocio que tiene como objetivo recuperar completamente la inversión inicial a partir del segundo año. Además, como segundo objetivo se plantea no requerir ninguna financiación a partir del mes 13, buscando lograr un balance positivo a partir del mismo.

Para la inversión inicial, requerida para cubrir los gastos de los primeros 14 meses, se estima necesario un capital inicial de 61.072 euros. Sin embargo, la primera propuesta para evitarlo es ofrecer a los empleados participaciones en la empresa durante los primeros 14 meses, reduciendo así los gastos previstos para los primeros 14 meses a 17.328 euros (debe tenerse en cuenta que sí deberá pagarse la parte correspondiente a impuestos de los salarios). Estos 17.328 euros podrían solicitarse como préstamo en bancos, sin embargo la primera opción es que se adelante por los mismos empleados de la empresa. De esta forma se ahorraría el pago de comisiones y los gastos derivados de todas la financiación bancaria y no se dependería de capital externo.

Con este horizonte en mente se propone un escenario de crecimiento mes a mes, puede verse un ejemplo en la tabla de la figura A.16 del Apéndice A, como objetivo de evolución del negocio.

Los dos primeros meses se plantean como toma de contacto con las empresas intermediarias, empresas de marketing, inmersión en el mercado, pruebas de concepto con clientes y contratación de los primeros paquetes Estándar, por lo que no se espera contar con clientes contratando ningún paquete *PRO*. A partir del tercer mes se espera haber logrado al menos un cliente *PRO*. Sin embargo, es pasado el cuarto mes cuando será necesario un crecimiento consistente de un cliente importante y fiel cada mes, que contrate el paquete *PRO* para alcanzar los objetivos de negocio. Para conseguirlo será necesario un producto de calidad que garantice clientes fieles que no se den de baja y que anime a los clientes pequeños a optar por este paquete. Así, a partir del primer año y ya con diez clientes de estas características se alcanzaría el objetivo un balance positivo mensual. Adicionalmente y con el objetivo de recuperar en 24 meses la inversión inicial se requiere mantener este crecimiento

hasta finalizar los dos primeros años. A partir del segundo año (mes 13) se comienza a tener un balance contable positivo (puede observarse con facilidad en la tabla de la figura A.16) y tras la finalización de los dos años, nos queda un balance global positivo de 7.508 euros. Una vez alcanzado este objetivo de retorno de la inversión inicial junto con la estabilidad en el balance contable, se comenzará a expandir y generar un conjunto propio de clientes de cara a la independización.

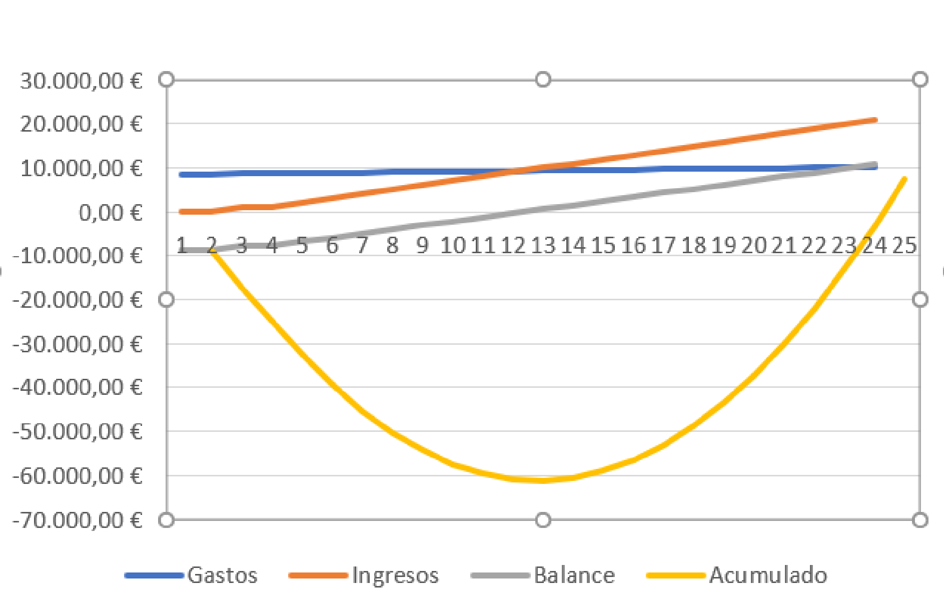


Figura 3.1: *Gráfica de evolución mensual*

En cuanto al retorno de la inversión, se puede ver en la gráfica de evolución mensual de la figura 3.1 cómo hasta el mes 24 no se alcanza un balance acumulado positivo. También puede observarse muy claramente cómo a partir del mes 12 los ingresos superan a los gastos, alcanzando un balance mensual positivo a partir de este.

Es importante hacer notar ese objetivo de llegar a partir del primer año a un balance positivo mensual, ya que de no ser así el acumulado seguiría descendiendo, agravando la situación económica y retrasando la fecha de recuperación de la inversión.

3.4.2. Modelo de negocio con inversión externa

En caso de que no exista consenso entre los integrantes de la empresa para proporcionar el capital inicial y acuerdo para el cobro de los salarios en participaciones, será necesario un mínimo de 61.072 euros de capital inversor inicial para arrancar con unas garantías mínimas el funcionamiento de la empresa. Además, en caso de que no se consiga llegar a un acuerdo con las empresas dominantes del sector hispanohablante de compra de interacciones en redes

sociales, será necesaria una inversión adicional en marketing de 13.000 euros mínimo(véase las sección 3.5), que podría subir a los 26.000 euros si quieren adelantarse los dos primeros años en marketing digital. En total se podría llegar a necesitar una inversión superior a los 74.000 euros solo para el primer año de funcionamiento.

Existen numerosas formas de conseguir financiación para una empresa nueva[6]: pedir un crédito al banco, inversores privados, fondos de capital riesgo, subvenciones públicas, financiada a través de los propios trabajadores, mediante campañas de financiación colectiva, colaborando con otras empresas, presentándose a concursos o pidiendo el dinero a amigos o familia, etc. En el primer modelo que se ha definido, el objetivo es financiar la empresa mediante la financiación por los propios trabajadores y ahorrar costes colaborando con empresas asentadas en el sector. Sin embargo, en caso de que esta opción no sea posible, se ha decidido como alternativa buscar inversión externa mediante aceleradoras de *Startups*. Esta decisión se debe a que este tipo financiación se ajusta perfectamente a las características de la empresa[10]: trabajadores jóvenes recién egresados, un proyecto ilusionante con gran innovación tecnológica, necesidad de mentorización en la gestión del negocio y gran posibilidad de crecimiento.

Conseguir la financiación de una aceleradora de *Startups* es muy complicado, sin embargo en España existen numerosas aceleradoras[11] que buscan proyectos novedosos, para las que la oferta propuesta (que une redes sociales e inteligencia artificial) puede resultar muy interesante.

En este modelo se considera presentar el proyecto a la aceleradora *Wayra*, ya que ofrece un mercado global que permitiría crecer más allá de España. Está financiada por *Telefónica*, una de las grandes empresas del Ibex 35 de España.

La aceleradora *Wayra* ofrece el capital inversor que se requiere, entre 50.000 euros y 250.000 euros. Además *Wayra* daría acceso a la oferta de servicios de *Telefónica*, como el servicio de alojamiento en la nube que permitiría reducir los gastos mensuales (puede verse la figura A.17 del Apéndice A como referencia).

3.5. Plan de marketing

El mercado de empresas de venta de interacciones en redes sociales está plagado de páginas que ofrecen todo tipo de servicios, sin embargo la opción de interacciones conversacionales de gran calidad que generen debate no está extendida en el mercado internacional. Por ello, la opción para crecer más rápido parece ser ofrecer el servicio a estas empresas que ya están colocadas en el mercado y que tienen un buen banco de clientes, pero que no ofrecen este servicio.

Un ejemplo es la empresa *COMPRASOCIALMEDIA* (véase la figura A.1 en el Apéndice A), la empresa dueña de compra-seguidores.com, una de las empresas más completas según *Encarni Arcoya* de *ActualidadeCommerce*[1] para la compra de todo tipo de interacciones en *Twitter*, *Facebook* y otras redes sociales.

El objetivo es realizar una oferta de servicios a estas empresas ya colocadas en el mercado ofreciéndoles un servicio novedoso de gran calidad, permitiendo una inmersión rápida en el

mercado desde el inicio, evitando así la necesidad de una inversión de capital inicial en marketing.

Se realizará un acercamiento inicial mediante el contacto indicado en sus webs online, buscando una reunión cara a cara para realizar una oferta en la que las empresas ya asentadas en el mercado ofrezcan este servicio a sus clientes a cambio de una comisión fija por cliente. Esto les permitirá completar su oferta ofreciendo un servicio del que no disponían, así como un ingreso extra. A la empresa se le facilitará una base inicial de clientes a partir de la cual crecer de forma sostenible.

Esta opción conlleva varios riesgos que se desarrollarán en el apartado 3.6, por lo que alternativamente en caso de que esta estrategia de marketing no surta efecto y no se obtengan los resultados esperados o se deseche por tener suficiente capital inversor inicial, se contemplará la opción de anunciarse en *Google Ads*. En este caso, será necesario ampliar la inversión inicial en unos 26.000 euros adicionales para garantizar la financiación del marketing los dos primeros años, lo que retrasaría el comienzo de la recuperación de la inversión hasta el mes 26, contando con el mismo crecimiento esperado hasta el mes 24(puede verse en la gráfica de la figura A.15 del Apéndice A).

La inversión calculada en este caso en marketing se estima en 1.000 euros mensuales basándose en las estimaciones realizadas por Javier Mir, experto de *xplora* empresa líder en marketing web, que establece[4] un coste aproximado de 900 euros al mes. Tomando un margen de 100 euros que se cree necesario para asegurarse estar en las previsiones por encima de la media nacional de inversión en marketing digital[14]. Toda la gestión del marketing se dejaría a la empresa *xplora* anteriormente mencionada, ya que no se dispone de un experto en marketing y es la opción que permitiría un crecimiento más rápido.

En caso de requerirse esta inversión inicial en marketing se requeriría la opción desarrollada en el modelo de negocio (véase sección 3.4.2) con una inversión inicial de capital externa.

3.6. Análisis de debilidades, amenazas, fortalezas y oportunidades

En este plan de negocio se han realizado varios modelos con el objetivo de cubrir distintos escenarios que podrían surgir. En base a estos escenarios, se va a realizar un análisis de los principales elementos del plan: sus debilidades, las amenazas que pueden surgir, las mayores fortalezas del modelo y las oportunidades.

3.6.1. Debilidad

Se ha planteado en el plan de negocio un posible escenario en el que este producto se ofrece a compañías ya asentadas en el mercado, por lo que no se incluye ningún gasto en publicidad. Sin embargo, esta estrategia para reducir la necesidad de un gran capital inicial es una de las grandes debilidades. Surge al utilizarla la opción de que ninguna de estas empresas esté dispuesta a aceptar los términos y condiciones propuestos o simplemente no

les interese ofrecer esta opción de reventa de servicios. En este caso ya se ha planteado como solución utilizar capital propio o una aceleradora de *Startups* para cubrir esta necesidad financiera. Esta opción como ya se ha visto llevaría a modificar el balance estimado.

3.6.2. Mayor amenaza: mes con balance negativo

Dentro de las mayores amenazas para esta empresa está la incapacidad de obtener clientes fieles que compren los grandes paquetes y que ayuden con un gran volumen de ingresos. Para ilustrarlo se muestra un mes con pocos clientes, en el que para el cálculo de los ingresos se estima que una empresa interesada en este servicio contratará de media un paquete *PRO* y 1000 iteraciones extra del paquete *Premium* al mes (véase el cuadro 3.1). Sin embargo, no se cuenta con más volumen de grandes clientes más allá de este. Este escenario, reflejado en la figura A.18 del Apéndice A, ilustra como si esto se prolongase a lo largo del tiempo se acumularían grandes pérdidas, estimadas en este caso en 6.313 euros cada mes, y que pondrían en riesgo la supervivencia a largo plazo de la empresa.

3.6.3. Fortalezas, amenazas y oportunidades de negocio

Estas tres aristas de este análisis de negocio giran en torno al mismo punto, que es la escasa oferta actual de interacciones conversacionales automáticas a gran escala. Es claramente la gran fortaleza y oportunidad que ofrece, permitiendo entrar en un mercado con poca competencia en este ámbito, que unido al creciente interés y prospectos de mercado de la inteligencia artificial[3] hace muy atractiva la oferta de la empresa.

Esta falta de oferta sin embargo supone también la mayor amenaza para los prospectos de negocio, ya que los clientes ya asentados pueden no llegar a ver interés en el nuevo servicio que se ofrezca al estar acostumbrados a trabajar con otras opciones de interacción ya conocidas y contratadas previamente. Para evitar esto es importante que el producto sea atractivo, especialmente debe hacerse incapié en la calidad de las técnicas de inteligencia artificial utilizadas.

La última de las grandes amenazas son las empresas que actualmente copan el mercado en cuanto a subida de tasa de interacción en redes sociales, a quienes incluso se les propone como una de las opciones del plan de marketing del capítulo 3.5 que sean vendedores intermediarios de estos servicios. Estas empresas una vez se empiece a vender el servicio podrían decidir desarrollar sus propias soluciones para ofrecer a sus clientes. Para evitar esto es muy importante que el producto que se ofrezca sea tanto revolucionario como de gran calidad, de forma que estas empresas no puedan desarrollar el mismo producto. Es importante también ir creando tanto una base de clientes fieles que conozcan este producto y lo compren regularmente como una reputación dentro del mercado.

3.7. Conclusiones del plan de negocio

En el lanzamiento de este proyecto se contemplan varias opciones de financiación y dos estrategias diferentes de marketing, que permitirán afrontar los posibles contratiempos que puedan surgir. Se analizan además las mayores debilidades del negocio, permitiendo adelantarse a las posibles amenazas que puedan afectar a la empresa.

La estrategia de marketing unida al plan de negocio, permiten estimar un escenario que cubra todos los costes que se esperan del funcionamiento de la empresa.

Este proyecto se lanza en un contexto en el que los servicios ofrecidos para la interacción conversacional en redes sociales son muy elementales y muchos tienen precios prohibitivos para la mayoría de empresas, influencers y creadores de contenido. En él se propone la creación de un entorno completo que permita ofrecer respuestas e interacciones a clientes a un coste asequible para la mayoría. El servicio permitirá, a partir de unos parámetros establecidos por el cliente, responder de manera inteligente y coherente a las interacciones que este indique. Además, se crearán respuestas cruzadas entre los propios perfiles que interactuarán además con la cuenta indicada por cliente. Las respuestas serán personalizables por el cliente, pero sin ser necesario que el cliente supervise ni explicite las respuestas, bastará con que este indique la actitud que quiere que tomen los distintos perfiles que participarán en las conversaciones. Adicionalmente el cliente podrá modular una serie adicional de parámetros opcionales que le permitirán obtener los resultados más adaptados a sus necesidades, en caso de requerirlo. Toda la interacción de los distintos perfiles estará gobernada por una inteligencia artificial desarrollada por el equipo del proyecto, que permitirá la automatización de las respuestas y permitirá abaratar los costes de cada respuesta que se publique.

Capítulo 4

Propuesta y diseño

A continuación se explican las distintas herramientas que se utilizan para acabar creando y utilizando un bot. El objetivo de los bots es simular una personalidad humana, con distintos gustos y caracteristicas, capaz de interactuar en una red social con otros usuarios.

Como se explicó anteriormente en la introducción, el objetivo de esta aplicación es crear una red de bots que opinen y discutan entre sí. Se busca que a veces tengan opiniones positivas, o negativas, que molesten o que apoyen a un usuario. Todo esto es posible gracias a las herramientas que describiremos a continuación. Se busca que a partir de un texto que se reciba como contexto (generalmente será el tweet del usuario al que se quiere responder) y algunos otros parámetros, generar un mensaje para que lo publique un bot como respuesta. Sin embargo, para poder crear ese mensaje, primero es necesario tener un bot creado con una personalidad, y que obviamente haya aprendido a escribir como un humano siguiendo ciertas técnicas de Machine Learning y con un corpus de texto ya escrito por personas como base.

4.1. Propuesta de interfaz para el usuario

La aplicación dispone una UI amigable que permite al usuario realizar varias acciones tanto para crear como para gestionar las trollnets a su elección. Se dispone un menú lateral que distribuye las vistas generales a las que se puede acceder. La Figura 4.1 muestra lo descrito aquí. Para el proyecto en concepto de prueba, las únicas vistas operativas son 'Nueva trollnet' y 'Galería'. Por defecto, al inicio de la aplicación, se carga la vista 'Galeria', que muestra la lista de trollnets. Al tratarse de la primera vez, no se habrá creado todavía ninguna trollnet, así que ésta aparecerá vacía. Se deberá por tanto acceder a la vista 'Nueva trollnet' a través del menú lateral para crear una nueva trollnet.

En la vista 'Nueva trollnet', se disponen una serie de diferentes componentes, cada uno con un título y unos selectores o campos diferentes, que permitirán configurar una serie de parámetros de la trollnet y sus bots. La figura 4.2 muestra lo descrito aquí. Sin entrar en mayor detalle (estos parámetros se explicarán más adelante), se encuentran los siguientes componentes:

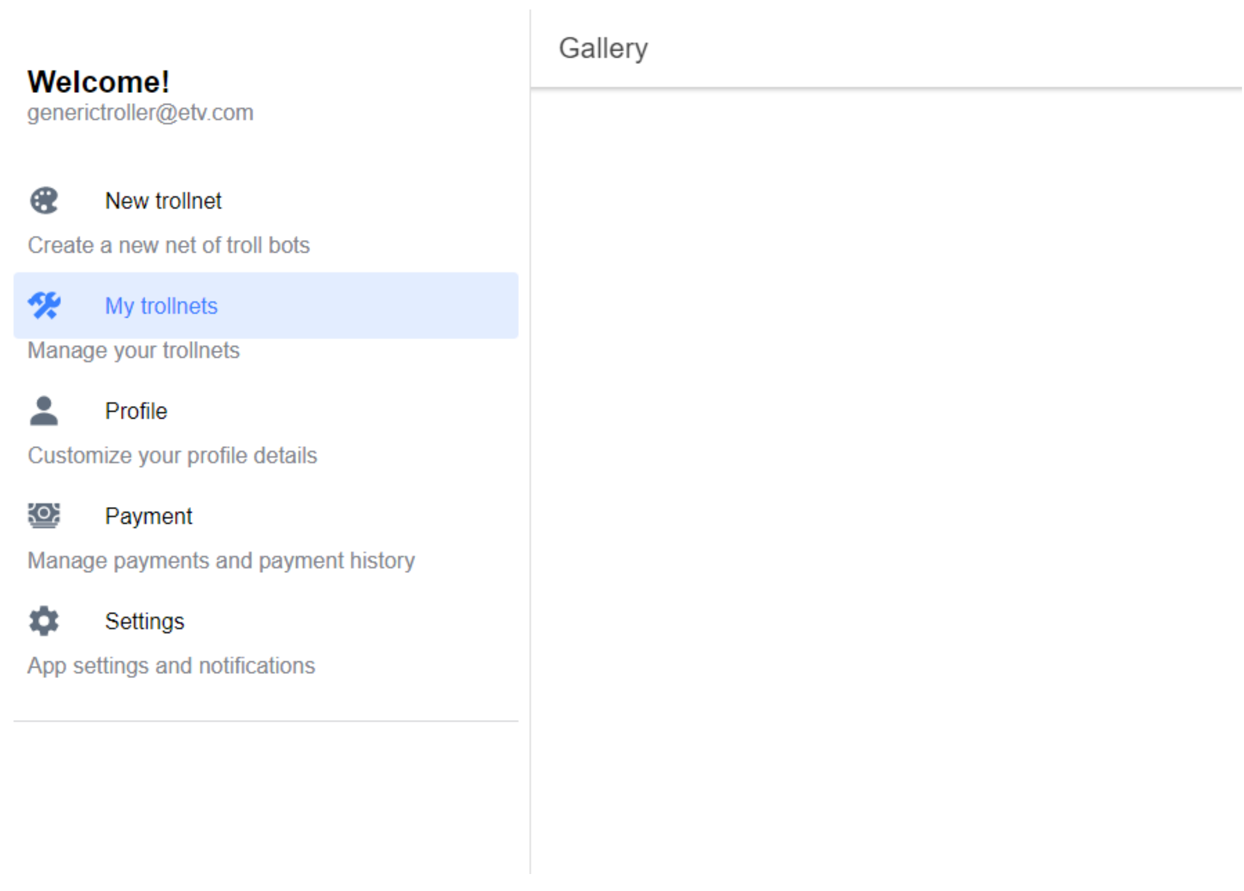


Figura 4.1: *Vista inicial de la galería vacía*

1. Un selector horizontal doble para ajustar el intervalo de edad de los bots.
2. Un selector de múltiples opciones para elegir los posibles géneros de los bots.
3. Un selector de múltiples opciones para elegir las posibles etnias de los bots.
4. Un selector horizontal doble para ajustar el intervalo del nivel de formalidad de los bots en sus mensajes.
5. Un selector horizontal doble para ajustar el intervalo de positivismo / negativismo de los bots en sus mensajes.
6. Un selector horizontal para ajustar la extensión de los mensajes de los bots (en un grado de 1 a 5).
7. Un selector horizontal para ajustar el tamaño de la trollnet (número de bots de 5 en 5 hasta 50 bots por trollnet).
8. Un campo de texto para introducir las palabras que serán los keywords de la trollnet.

9. Otro campo de texto para los likes de la trollnet.

10. Otro campo de texto para los dislikes de la trollnet.

Además, en la parte superior se dispone un título por defecto para la trollnet, junto a un botón que al hacer clic abrirá un popup que permitirá editar el nombre de la trollnet.

En la parte inferior, se encuentran 3 posibles acciones: 'Limpiar trollnet' para limpiar el borrador de trollnet que se haya podido configurar; 'Trollnet por defecto' para usar una plantilla de ejemplo que rellene todos los campos con valores; 'Crear trollnet' para lanzar el proceso de creación de la trollnet con los parámetros configurados en ese instante.

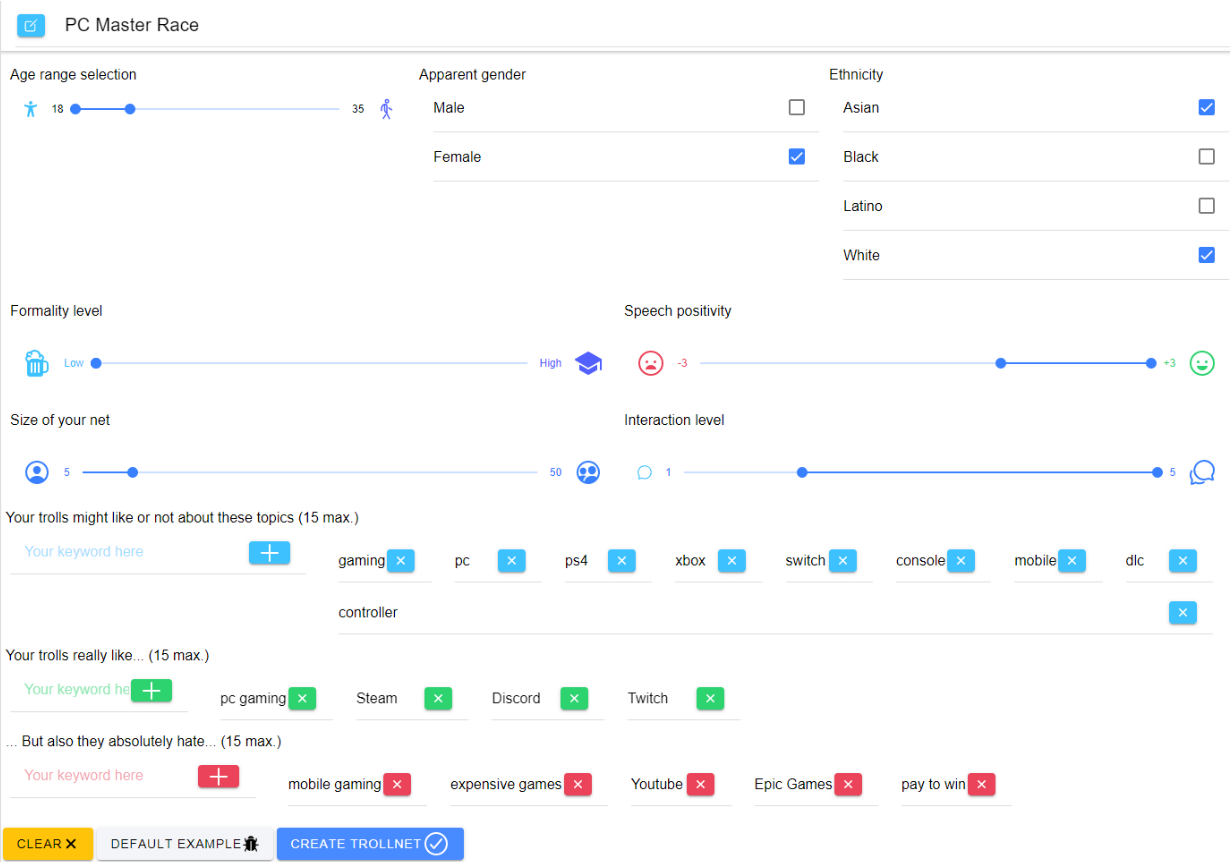


Figura 4.2: *Vista de creación de nueva trollnet*

Al rellenar la configuración con los valores deseados y presionar 'Crear trollnet', aparecerá un pequeño diálogo de confirmación avisando de que el proceso va a empezar y no se puede revertir. Una vez se confirme, se mandará la orden y los datos necesarios al backend en una petición; si todo funciona bien, se mostrará un pequeño aviso de confirmación de que ha funcionado correctamente, y si se navega a la vista de 'Galería' en el menú lateral, se puede apreciar que ahora aparece un pequeño recuadro correspondiente a nuestra nueva

trollnet recién creada, y que las barras de estado de creación y entrenamiento van avanzando progresivamente, como se observa en la figura 4.3.

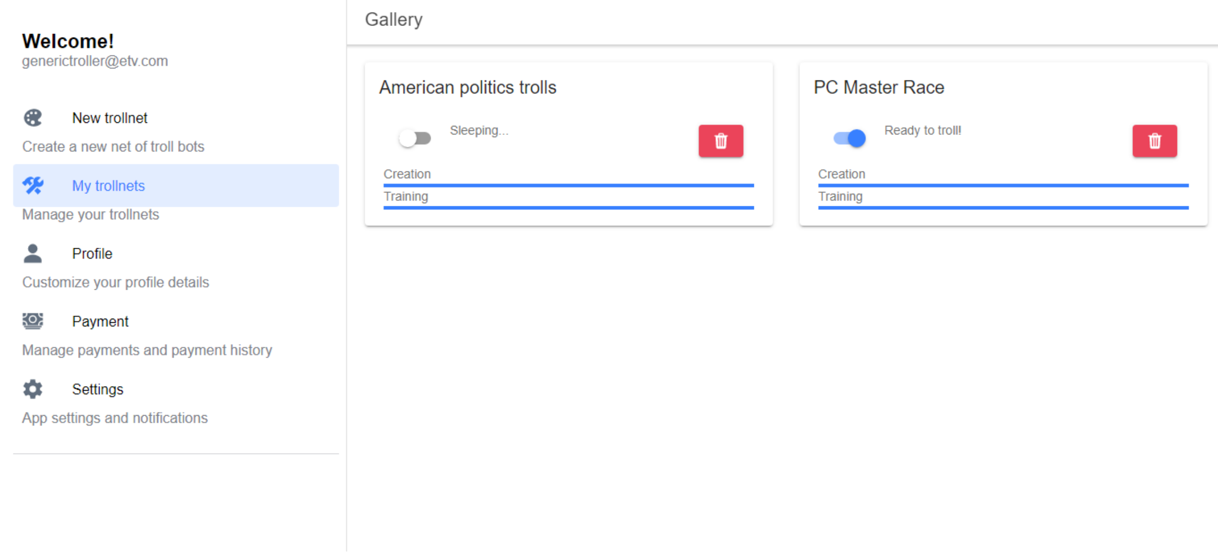


Figura 4.3: *Vista de galería con trollnets creadas*

En esta vista, cada uno de los componentes mostrados corresponde a la vista previa de una trollnet, y podemos ver la información básica que identifica a dicha trollnet así como su estado actual (como se observa en la figura 4.4):

1. El nombre que se le atribuyó en la parte superior de la caja,
2. El estado de avance en su creación mediante una barra de progreso horizontal junto a una etiqueta 'Creación',
3. El estado de avance en su entrenamiento mediante una barra de progreso horizontal junto a una etiqueta 'Entrenamiento'.

Además, encontraremos un botón rojo con una papelera para eliminar esta trollnet junto a todos sus bots, y un interruptor que, de activarlo, abrirá un cuadro de diálogo donde podemos introducir el nombre de la cuenta objetivo de Twitter (como se observa en la figura 4.5). Así, cuando introduzcamos el nombre del perfil, se mandará una orden al backend para que escuche dicha cuenta, y tan pronto dicha cuenta publique un nuevo tweet, nuestros bots empezarán a generar una conversación entorno a ese tweet.

4.2. Modelos de generación de texto

Un modelo de generación de texto (o simplemente *modelo*), permite generar texto *aleatorio* entrenado a partir de un corpus específico. Es aleatorio en el sentido de que no se

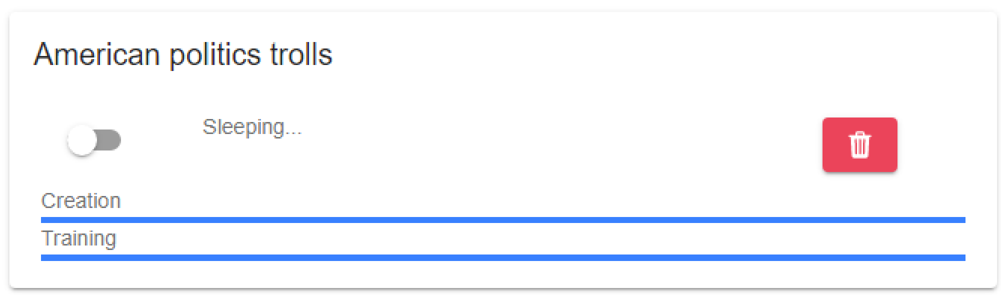


Figura 4.4: *Vista de estado de la trollnet*

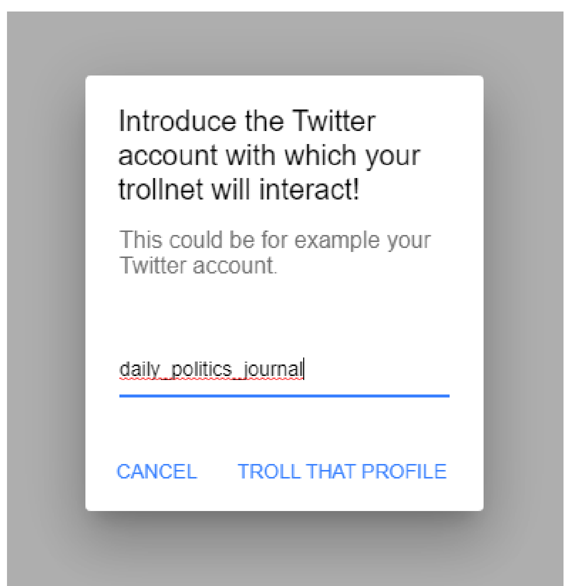


Figura 4.5: *Popup para generar conversación contra una cuenta objetivo de Twitter*

controla qué va a decir, pero se puede afirmar que este texto está construido de manera similar a los textos que se le hayan proporcionado en el corpus.

4.2.1. Implementación

Estos modelos se crean a partir de GPT-2 (Generative Pre-trained Transformer), un sistema basado en Transformers. Este sistema es totalmente Open Source y ha sido desarrollado por OpenAI.

El paquete utilizado, *gpt-2-simple*, implementa varias funcionalidades, que según se describen en GitHub:

> *This package incorporates and makes minimal low-level changes to:*

- *Model management from OpenAI's official GPT-2 repo (MIT License).*
- *Model finetuning from Neil Shepperd's fork of GPT-2 (MIT License).*
- *Text generation output management from textgenrnn (MIT License/also created by me).*

Sobre este paquete se construyen distintos módulos, que facilitan y adaptan a la aplicación su uso, `responseGeneration.py` y `models.py`.

4.2.2. models.py

Este módulo permite crear una interfaz con el paquete, que en este caso facilite la configuración del entrenamiento de un modelo. Consiste en un único método:

trainModel(corpusPath, nameOfModel, num_iterations, _model_version)

siendo sus parámetros los siguientes:

- corpusPath: ruta donde se aloja el corpus.
- nameOfModel: nombre que se le quiere dar al modelo.
- num_iterations: número de iteraciones que aplicará el entrenamiento.
- _model_version: versión del modelo base.

Además, este módulo define la clase Model, que contiene todos los datos necesarios para poder utilizar los modelos entrenados más tarde. Estos datos también permiten que el bot pueda hacer una selección adecuada de modelo 4.4.3.

Keywords de un corpus

Una vez se tiene un modelo entrenado, es necesario un mecanismo que describa el corpus con el cual se ha entrenado este, para permitir seleccionar un modelo que se ajuste a lo deseado cuando existen multitud.

Un corpus no es más que, al fin y al cabo, un montón de palabras que de alguna manera tienen relación. Entonces si se seleccionan las n palabras más descriptivas o significativas del corpus, aparece una manera de describir de *qué* va a hablar el modelo. Estas palabras deben conseguir distinguirse del resto de corpus, por tanto el primer paso debe ser eliminar las palabras que no aportan información sobre el contenido, estas son las llamadas *stopwords*.

Una vez eliminadas estas palabras, hay que establecer una jerarquía sobre las restantes, para poder clasificar las n más descriptivas. Esta jerarquía sobre las palabras, sobretodo cuando se habla de un modelo que genera texto de la misma manera que el corpus está redactado, es lógico que sea el número de veces que aparece la palabra en el corpus. Por ejemplo, si en un corpus aparece muchas veces la palabra *política*, se puede pensar que es bastante probable que el modelo la introduzca en los textos que genere. Consiguientemente,

lo único que se debe hacer es un recuento de todas las palabras (ya filtradas sin *stopwords*), y seleccionar las n que mas aparecen, y este es el mecanismo que sigue un modelo para generar sus *keywords*.

La implementación del conteo se realiza en C++, debido a que es una tarea potencialmente costosa y se requiere eficiencia, por lo que evitar la naturaleza interpretada de Python, y su gestión automática de memoria es recomendable, además C++ dispone de estructuras de datos optimizadas que serán útiles.

El algoritmo es el siguiente, se leen todas las palabras en el texto y se cuentan haciendo uso de un *unordered_map*, que permite que el coste del conteo sea $O(n)$, aprovechando que el coste mínimo posible para leer las palabras es el mismo. Además, se guardan en una lista no ordenada las claves del mapa, sin repetir. Una vez realizado el conteo, se ordenan las palabras mediante el uso de una cola de prioridad, implementada como *Fibonacci Heap*. Se recorren todas las claves guardadas en la lista, y se insertan en la cola con coste total en el peor caso de $O(n)$. Por último se lee m veces la cabeza de cola (siendo m el número de palabras que interesa hallar, i.e. las 100 primeras), con un coste de $m \cdot \log n$. Esto da un coste final de $n + n + m \cdot \log n$.

Se podría discutir que una mejor solución utiliza inserción ordenada, pero a continuación se describe por qué a partir de $m = 100$ para tamaños de corpus de unas 30000 palabras y en adelante, esta es una peor solución.

Una solución utilizando inserción ordenada, puede ser la siguiente: A medida que se leen las palabras ($O(n)$), se insertan en una lista enlazada ordenada el elemento. Utilizando búsqueda binaria, se tiene un coste de $O(\log n)$ para elementos nuevos. En el caso de un elemento ya presente en la lista (i.e. una palabra que ya se ha encontrado anteriormente), el coste para insertar es como mínimo $O(\log n)$ de nuevo: si se guardasen las posiciones de las palabras en la lista en un *HashMap*, el borrado del elemento sin actualizar en la lista y la correspondiente actualización de punteros tiene un coste de $O(1)$, más su inserción en la lista en la nueva posición $O(\log n)$. Despreciando la lectura con coste $O(m)$ posterior, el coste total sería $O(n \cdot \log n)$, un coste mayor al encontrado con el método anterior.

Una posible mejora a este método, es adaptar el tamaño de la lista a $O(m)$, en vez de $O(n)$. De este modo, solo se guardan en la lista las palabras que más tarde serán leídas, haciendo uso de un *HashMap* de manera auxiliar que guarda los contadores de todas las palabras. Esto hace que el coste de una inserción en el peor caso (el caso en que esta palabra está entre las m primeras), sea de $O(\log m)$, si se sigue el mecanismo anterior, lo que da un coste total de $O(n \cdot \log m)$. Aunque se obtiene una mejora muy grande, y un coste muy similar al primer método, este sigue siendo superior a medida que n y m se hacen grandes.

Models Descriptor List

Para que la aplicación pueda guardar los modelos que ya ha entrenado, con todos los datos necesarios para describirlos, se utiliza la *Models Descriptor List*, que se implementa como una lista de Python que contiene descriptores de modelo (modelos en JSON). Esta lista a su vez se convierte a JSON para que la aplicación pueda guardarla facilmente.

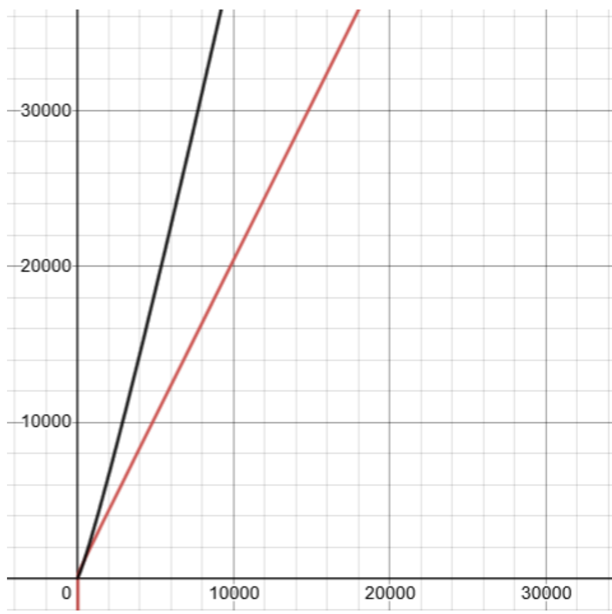


Figura 4.6: *Negro:* $n \cdot \log n$ *Rojo:* $n + n + m \cdot \log n$ *para* $m = 106$

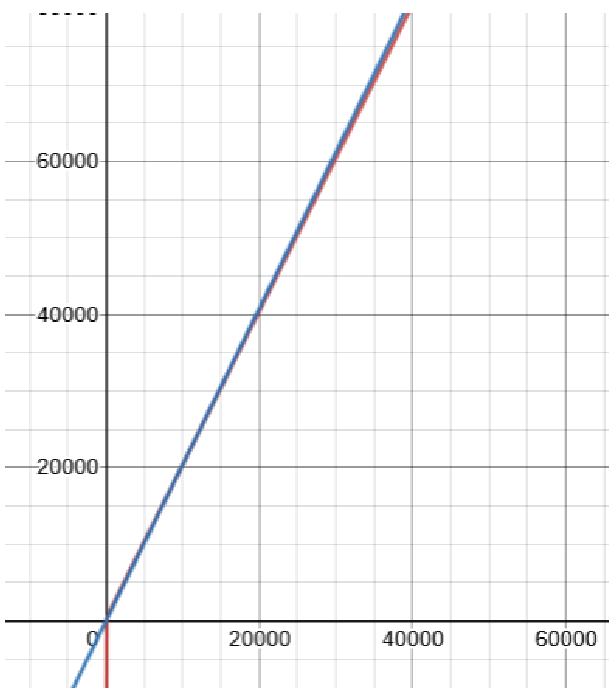


Figura 4.7: *Azul:* $n \cdot \log m$ *Rojo:* $n + n + m \cdot \log n$ *para* $m = 110$

4.2.3. responseGeneration.py

Este módulo se encarga de proporcionar una interfaz con el paquete gpt-2-simple, que permita parametrizar y filtrar la salida que se genera. Esta interfaz consiste en un método principal:

> generateResponse(model, posFactor, slangFactor, keyWords, nchars, number_of_responses, prefix)

Estos parametros se pueden dividir en aquellos que se le proporcionan al modelo, y aquellos que se utilizan en el filtrado. Los parametros dirigidos al modelo son:

- number_of_responses: número de *textos candidatos* que se generan.
- prefix: prefijo a partir del cual el modelo generará el texto.

Por otra parte, los parámetros dirigidos al filtrado son:

- posFactor: factor de positividad.
- slangFactor: factor de informalidad.
- keyWords: palabras clave.
- nchars: número máximo de caracteres que debe tener la salida.

Filtrado de *textos candidatos*

El modelo se ocupa de generar un número de respuestas, llamados *textos candidatos*. Todos ellos pasan por el *pipeline de filtrado*, formado por una serie de *funciones de filtrado*, que se aplican en orden. Estas pueden ser *funciones de puntuación*, las cuales puntúan de 0 a 10 al texto candidato según su *criterio de filtrado*, o bien *funciones de modificación*, que alteran el contenido de estos *textos candidatos*, limitando los caracteres o añadiendo expresiones informales. El *texto candidato* que más puntuación consiga será la salida.

Todas las funciones de filtrado deben cumplir una serie de estándares:

- La entrada es una lista de duplas. Cada dupla está formada por el texto en primer lugar (valor modificado en las *funciones de modificación*) y la puntuación en segundo lugar (valor modificado en las *funciones de puntuación*).
- La salida es una lista de duplas de igual manera.

Es recomendable que a esta puntuación se le *añada* y no *asigne*, ya que podría machacar puntuaciones aplicadas por otros filtros. Además mantener las puntuaciones de 0 a 10 permite que no haya filtros "*overpowered*", que valoren en exceso una característica del texto, haciendo que las puntuaciones del resto de filtros sean insignificantes.

Podemos encontrar las siguientes *funciones de puntuación*, las cuales únicamente alteran la puntuación de los *textos candidatos* añadiéndoles una puntuación de 0 a 10 cada una.

- *Positividad:* Esta función simplemente hace uso del parámetro *posFactor* para dar mayor puntuación a los textos que tengan una polaridad similar a la del factor dado. Es decir, si el factor es un valor positivo, los textos con un sentimiento igual de positivo recibirán una mayor puntuación que los que tengan un sentimiento demasiado postivo, menos positivo, o directamente un sentimiento negativo. En el mejor de los casos, si ambos valores coinciden, ese texto recibirá 10 puntos.

- *Keywords:* esta función hace uso del parámetro *keywords*, que en el fondo no es más que una lista de palabras clave. Dicho filtro comprueba cuántas de las palabras clave aparecen en cada texto, y luego lo divide entre el número total de palabras clave y lo multiplica por 10. De esta forma se consigue que el filtro siempre otorgue valores entre 0 y 10, cumpliendo así los estándares indicados anteriormente.

Por otro lado las *funciones de modificación*, que utilizan distintos parámetros para modificar ligeramente el texto resultante.

- *Formalidad:* se usa para alterar el texto de forma que parezca escrito con más cuidado, si el valor del parámetro es bajo, o escrito de la forma más vulgar de internet, si el valor es alto. En la práctica, se considerarán los textos originales lo más formales posibles, debido a como funcionan los modelos, y a la dificultad que supondría volverlos más formales. Es por ello que los valores bajos del parámetro de informalidad simplemente harán que el texto se modifique menos. Esto se debe a que el filtro toma ciertas medidas para hacer el texto más vulgar, cada una de ellas se ejecuta con una probabilidad que se vuelve más alta según sea más alto el factor.

- *Formato:* este filtro formatea el texto de modo que la salida cumpla una longitud máxima determinada. Además elimina anomalías como URLs incompletas, dobles saltos de linea y otras posibles salidas del modelo.

Las medidas de informalización empleadas son las siguientes:

- Abreviar ciertas expresiones en versiones más cortas y muy comunes en el habla de internet. Para ello se hace uso del archivo `abbreviations.py`, donde se ha creado un diccionario con las expresiones completas como claves, y un par como valor. El par consiste en la expresión abreviada y la probabilidad individual de que se reemplace esa expresión. Estas probabilidades se han predefinido manualmente basadas en la experiencia del equipo usando internet. Por ejemplo, se puede cambiar *I don't know* por *idk*.

- Añadir ciertas palabras o los conocidos *emoticonos* al inicio y/o al final de los textos, basados de nuevo en el archivo de abreviaciones, donde se tienen dos diccionarios con expresiones iniciales y finales y sus respectivas probabilidades de aparición. Por ejemplo, se puede añadir :) al final del texto.

- Eliminar ciertos símbolos de puntuación, en particular el punto, la coma y el apóstrofe (este último se debe a que los textos son en inglés).

- Cambiar las letras mayúsculas por minúsculas con cierta probabilidad. De nuevo, esta probabilidad depende del factor de informalidad.

- Añadir pequeños errores de escritura alterando el orden de algunos pares de letras adyacentes en el texto, con una probabilidad muy baja y que sólo ocurre en factores de informalidad superiores a 5.

Cabe destacar que todas estas medidas se deben al hecho de que el usuario medio de internet es perezoso y tiende a cometer errores, o acortar frases con tal de enviar su mensaje antes. Por lo que el objetivo es hacer el texto más humano y realista.

Este tipo de construcción permite que el escalado y configuración del filtrado sea realmente simple, si se quiere añadir una función de filtrado basta con que cumpla los estándares y se introduzca su llamada en el pipeline.

4.3. Análisis de sentimiento

A continuación se describe una herramienta utilizada por distintos elementos de la implementación. Esta herramienta permite analizar textos para sacar información que pueda ser útil en distintos ámbitos, como puede ser el filtrado de textos que se explicará más adelante. Para este propósito se construye funcionalidad sobre el paquete de Python Spacy, que implementa pipelines con distintos propósitos para analizar texto, y conseguir distintas informaciones de este. De la herramienta de análisis de sentimiento se pueden sacar las siguientes propiedades sobre un texto:

- Nombres propios del texto.

- Sustantivos del texto.

- Adjetivos utilizados en el texto.

- El sujeto de las frases.

- El objeto de las frases.

- Puntuación de polaridad del sentimiento, 1 muy positivo y -1 muy negativo.

- Expresiones que describen la polaridad del texto, con su respectiva puntuación.

- Puntuación de subjetividad del texto, 1 muy positivo y -1 muy negativo.

El texto se separa por frases, y se ordena de mayor polaridad absoluta a menor.

La implementación se caracteriza por dos clases Python: `Classifier` y `Properties`. Cuando se crea una instancia de la clase `Classifier`, se puede utilizar para analizar distintos textos a través del método `classify(text)`. Este método recibe un texto `text` como string y nos devuelve una lista de instancias de la clase `Properties`. Las instancias de este objeto reúnen los atributos analizados por el `Classifier` descritos más arriba.

4.4. Bots

Ahora se debe conseguir crear un ”bot”, entendiendo como bot una entidad capaz de responder o formular un texto, que tenga el mayor sentido posible, dado un contexto (ya sea un comentario en Facebook, un tweet, etc). Además, estos bots deben tener (o simular tener) una personalidad, es decir, que hablen de una serie de temáticas en específico, con actitudes negativas o positivas, que utilicen ciertas expresiones...

Para ello se disponen de las herramientas descritas anteriormente, el *analisis de sentimiento* (véase sección 4.3) y los *modelos de generación de texto* (véase sección 4.2).

En principio, se podría entrenar simplemente un modelo, con un corpus suficientemente grande, que reuniera todos estos comportamientos que se quieren simular en el bot. A priori parece que podría funcionar, pero no es una solución muy práctica ni eficiente, ya que apenas se puede controlar *qué* va a decir el modelo, y podría responder algo que no tiene sentido en el contexto. Por ejemplo, dado un modelo entrenado con los tweets de *Jimmy*, que es un chico de 19 años al que le encanta el fútbol y la música clásica. Si se quiere generar una respuesta, simulando la parte músical de la personalidad de Jimmy, para responder a otro usuario, no tendría mucho sentido que el bot respondiera hablando de Cristiano Ronaldo.

Por otro lado, está el *analisis de sentimiento*, que podría ser útil en esta situación. Con los datos que extrae el análisis, se podría solicitar al modelo que genere multiples respuestas, analizar cada una de ellas, y realizar un filtrado de las que más se ajusten a la salida esperada. Ahora se puede esperar que la salida se ajuste más a lo deseado, sin dejar tanto al azar, aunque aún es mejorarable.

El siguiente paso lógico, es separar los distintos aspectos de la personalidad de *Jimmy* en distintos modelos. Entonces por un lado está el *Jimmy* que habla de fútbol, y el que habla de música. Entonces si el bot consigue discernir, dado un contexto, que ”aspecto de su personalidad” debe seleccionar, con mayor seguridad responderá algo coherente, y que simule un comportamiento humano, que es en última instancia el objetivo. Esta idea, aunque en teoría suena bien, tiene algunas limitaciones prácticas, y es que conseguir corpus suficientemente grandes de cada uno de los aspectos de la personalidad de *Jimmy*, no es una tarea fácil, ni a veces posible.

Una solución posible a todos estos problemas, es buscar un equilibrio entre todas las ideas anteriores. Se puede entrenar un modelo con un corpus *gigante* en el que solo se hable de fútbol, otro gigante en el que se hable de música, etc. Ahora si se quiere que *Jimmy* hable de música, basta con seleccionar el modelo entrenado en música, y si se le suma al filtrado, hacer que *Jimmy* odie a *Tchaikovsky*, pero hable bien de *Brahms* parece una tarea más asequible.

4.4.1. Implementación

Para condensar en atributos las características que se le quieran dar a un determinado bot, este guardará:

1. Edad.

2. Nivel de educación (Desescolarizado, educación obligatoria, estudios superiores).
3. Lista de gustos *(likes)*.
4. Lista de disgustos *(dislikes)*.

Siguiendo las ideas expuestas anteriormente, los bots serán capaces de aprender de los modelos que se ajusten a su personalidad, es decir, asociarse a este modelo para más tarde utilizarlo para generar una respuesta. Basta con que el bot sepa seleccionar el modelo adecuado en el contexto dado. Este contexto será normalmente un texto corto, de no más de alrededor de 280 caracteres (límite de *Twitter*).

4.4.2. Aprendizaje de modelos

El escenario de partida es el siguiente, existe un bot, con una serie de gustos y disgustos y se dispone de una serie de modelos, cada uno de ellos con sus *keywords* (véase sección 4.2.2) asociadas. Además existe una base de datos de relaciones que se verán en la sección 4.7.

El objetivo es hacer que el bot sea capaz de asociarse a los modelos que se ajusten a sus gustos, simulando que el bot "sabe" de esa temática. Por ejemplo, si un bot en sus gustos tiene *Kendrick Lamar*, y se dispone de un modelo entrenado en *hip hop*, lo ideal es que este bot sea capaz de asociarse a ese modelo.

Con todas estas herramientas, el mecanismo que se sigue es el siguiente, el bot recorre todos sus gustos y disgustos, asi como las *keywords* de cada modelo, buscando las relaciones que existen entre estos conceptos. Estas relaciones tienen una puntuación asociada, de -1 a 1, que muestra la intensidad (ya sea positiva o negativa) de las relaciones. En el mejor de los casos, todos los likes y dislikes tendrán una fuerte relación con todas las keywords, esto hará que la puntuación final fuese (likes + dislikes) * keywords. Si se mide de manera relativa a este valor las puntuaciones halladas, se puede establecer un umbral porcentual que se quiere superar para considerar que un modelo encaja.

4.4.3. Selección de modelo para generar una respuesta

Para esta tarea se utiliza un mecanismo muy similar al anterior, en este caso se comprueba la relación entre los sustantivos comunes y propios que existen en el contexto y sentimentAnalysis encuentra, y las keywords de cada uno de los modelos. En el caso en el que un modelo encajase a la perfección con el contexto, todos los sustantivos tendrían una relación fuerte con cada una de las keywords del corpus. A partir de esto, se calculan los grados de relación de manera relativa a ese número. El modelo que mayor grado de relación consiga es el seleccionado.

4.4.4. Creación e interacción con un Bot

Para interactuar con los bots, se encapsula la funcionalidad en la clase BotInstance. Sus métodos se describen a continuación:

- `constructor(age, level_of_education, loadModels, likes, dislikes)`: se crea una instancia del bot con o sin modelos asociados.
- `learn(modelDescriptorList)`: dada la lista de descriptores de modelo, el bot asocia los modelos que se ajusten a su lista de likes y dislikes.
- `generateResponse(context, filterParams)`: a partir de un texto de partida `context`, el Bot genera una respuesta. Esta respuesta se puede filtrar bajo una serie de opciones que se configuran tal y como se explica en la siguiente sección.

4.5. Filter Parameters

Debido a la naturaleza incontrolable y aleatoria de la generación de texto, se utilizan una serie de parámetros que permiten filtrar el texto que se genera para conseguir ajustar el resultado buscado. Estos parámetros son:

1. *keywords*: una lista de palabras que se valoran positivamente a la hora de seleccionar una respuesta. Cuantas más de estas palabras aparezcan en un texto candidato, más puntuación conseguirá. Por defecto, no hay keywords.
2. *positivity factor*: un número, comprendido entre -1 y 1, siendo 1 muy positivo y -1 muy negativo. Este número se compara con la puntuación que le da el análisis de sentimiento a un texto candidato, y cuanto más cercanos sean estos valores, mayor puntuación conseguirá el texto candidato. Por defecto, no se aplica.
3. *number of chars*: un entero, mayor que 0. Este número sólo indica el largo máximo del texto, medido en caracteres. Este parametro trunca el texto en la ultima frase acabada en '.' que cumpla que el largo del texto es menor o igual que el number of chars introducido. Por defecto, sin límite.
4. *number of responses*: un entero, mayor que 0. Especifica el número de textos candidatos que el generador producirá. Cuantos más textos candidatos se generen, más probable es conseguir una respuesta que se ajuste mejor a lo buscado, pero el tiempo de computación también aumentará.
5. *slang factor*: un entero, entre 0 y 10. El slang se podría denominar en castellano como jerga. Como su nombre indica, se refiere al lenguaje informal, en este caso el usado en internet. Cuanto mayor sea el número, más probabilidad hay de que el texto se modifique para tener un toque más casual. Por defecto, no se aplica.

Todos estos parámetros se encapsulan en una clase **FilterParams**, que es la que se le proporciona al bot a la hora de generar una respuesta.

4.5.1. Filter Parameters en los scripts

Para especificar los parámetros de filtrado en los comandos, se debe seguir la siguiente sintaxis, especificada en *filterParams.txt*:

- *-k keyword1,keyword2*: keywords (default None).
- *-pf float*: positivity factor (default None).
- *-nc int*: number of chars (default -1, "i.e." no limit).
- *-nor int*: number of responses of the model (default 10).
- *-sf float*: slang factor (default None, must be between 0 and 10).

4.6. Inferencia de parametros de filtrado

Partiendo de tener bots que consiguen generar una respuesta lógica dados unos parametros de filtrado, para lograr que los bots sean independientes y conseguir adaptabilidad a diversas situaciones, se debe lograr que los parametros de filtrado se calculen de manera automatica a partir del *contexto* antes de generar una respuesta. Este *contexto* será en este caso un post en una red social, es decir, un texto pequeño (alrededor de 280 caracteres de media) que habla de algo muy concreto.

Para cada uno de los parametros de filtrado, se utiliza una función distinta. Se describen a continuacion:

4.6.1. Inferencia de *Positivity Factor*

Encontrar el valor adecuado para este parametro es una tarea bastante compleja, ya que automatizar un método de "comprensión" del contexto y fusionarlo con la personalidad del bot para que este responda con un factor de positividad acorde es algo dificil. Partiendo de la herramienta de analisis de contexto (véase sección 4.3), que separa el texto en frases ordenadas por mayor positividad absoluta (es decir, la frase que mayor intensidad de sentimiento presenta, se puede asumir que esta frase en general describe bien el carácter y de lo que se habla), se intenta "entender" de la entidad o tema del que se habla. Como se dispone de una base de datos de relaciones (véase sección 4.7), si se encuentra un "objeto" que esté emparejado se podrá deducir el resultado. Por tanto, se analizan las frases en busqueda de una palabra (o grupo de palabras) que será utilizada como "resumen" del contexto dado. Se busca este resumen en el siguiente orden, buscando en el siguiente grupo de palabras si el anterior estaba vacio:

1. Objetos de las frases.
2. Nombres propios.
3. Sustantivos.

Una vez se tiene la palabra resumen, se intenta emparejar con una palabra que esté en los likes o dislikes del bot. Si esta palabra se encuentra de manera directa en la lista de likes o dislikes del bot, se devuelve 1 o -1 respectivamente, si no, se utiliza el buscador de relaciones de la base de datos de relaciones. Si el factor conseguido se encuentra a traves de un like, se devuelve la relación encontrada directamente, y si se encuentra a traves de un dislike, esta relación se multiplica por -1 (siguiendo el razonamiento de ejemplo, si alguien odia el reggaeton, y el reggaeton y la bachata tienen una relacion asociada de 1, se puede suponer que su relacion con la bachata también será de -1).

4.6.2. Inferencia de *keywords*

A partir del contexto dado, se deben conseguir una serie de palabras que tendrían sentido en una respuesta y harían que fuese coherente. Lo más sencillo, es que el bot responda con los mismos conceptos de los cuales se habla en el contexto dado, asi que simplemente se añaden a la lista de *keywords* los objetos de las frases, los nombres propios, y los sustantivos utilizados.

Además de esto, se cuenta con la lista de *likes* y *dislikes* (véase sección 4.4.1), que no es mas que una lista de palabras que describen lo que le gusta y lo que no. Si entonces se demuestra que esas palabras tienen relaciones con las que hay en el contexto, también se pueden añadir. Para ello se hace uso de la base de datos de relaciones descrita en la sección 4.7, y se buscan relaciones de mayor puntuación absoluta que 0.7 (valor arbitrario, con una puntuación mayor que 0.7 se puede considerar que existe una relación fuerte), entre la lista de likes y dislikes, y los objetos, nombres propios y sustantivos que aparecen en el contexto.

4.6.3. Inferencia de *Slang Factor*

Al igual que ocurre con el *Positivity Factor*, inferir un buen valor del factor de informalidad no es tarea sencilla. Sin embargo, se dispone de una ventaja, y es que el factor de informalidad está definido de forma muy específica, con valores y abreviaciones escritas a mano por el equipo del proyecto, por lo que se puede aplicar cierta "ingeniería inversa" sobre el *filtro de informalidad* (véase sección 4.2.3) para saber lo informal que es el contexto. Con esta idea ya está casi todo hecho. Simplemente basta con buscar en el texto todas esas abreviaciones y emoticonos que se consideran lenguaje vulgar, y añadir puntos por cada aparición. Esos puntos comenzarán en 0 (y se mantendrán así si no hay expresiones de las buscadas) y podrán aumentar hasta 10 si se encuentra gran cantidad de ellas. De esta forma se obtiene directamente el parámetro buscado, y respetando el rango de valores que espera.

4.7. Base de datos de relaciones

Para que un bot sepa como debe interactuar con el mundo, debe conocer que cosas están relacionadas y de qué manera, y asi poder deducir como responder. Con este objetivo se

crea la *base de datos de relaciones*, que guarda de una manera muy simple las relaciones entre conceptos. Por ejemplo, para describir la relación que tiene el *Real Madrid* con el *FC Barcelona*, basta con guardar (*Real Madrid*, *FC Barcelona*, -1), y con ello se deja declarado que existe relación entre ambos conceptos, siendo de hecho una relación negativa.

4.7.1. Implementación

La implementacion de esta base de datos se realiza sobre Prolog, aprovechando que la manera de representar las relaciones es directamente sintaxis de Prolog, y se pueden definir reglas de consulta que ayuden a construir algunas funciones facilmente sobre estas relaciones de base. Como se ha descrito antes, para guardar las relaciones basta con escribir sobre los hechos *relacion(real_madrid,fc_barcelona,-1)*.

Como se puede esperar, es muy complicado tener las relaciones de manera directa entre todas las entidades del mundo, por ello las *relaciones complejas* son útiles. Las *relaciones complejas* no son mas que el emparejamiento de dos entidades utilizando entidades intermedias. Por ejemplo, para encontrar la relación entre a y c, si se conoce que a tiene relación con b, y b tiene relación con c, se puede deducir una relación entre a y c, y este proceso podemos hacerlo tan complejo como se quiera. Se construyen reglas siguiendo el razonamiento: *el amigo de mi amigo es mi amigo, y el amigo de mi enemigo es mi enemigo*. Simplemente se busca el emparejamiento menos complejo entre dos entidades, y aplicando un factor de perdida de amistad. Este factor se va restando entre relaciones complejas, partiendo de la puntuación hayada en la relación simple de la que se parte, siempre que esta sea positiva y sin pasar de 0 (no tiene sentido pensar que por mas entidades que haya entre dos objetos, estas se van a odiar más, simplemente sentirán mayor indiferencia).

4.7.2. factsGeneration.py

A pesar de que el sistema de *relaciones complejas* es una gran herramienta para poder relacionar la mayoría de términos sin necesidad de tener una base de datos muy extensa, se siguen necesitando algunos hechos o relaciones iniciales de los que partir, sobre los cuales formar las susodichas relaciones. Debido a la naturaleza de las reglas, estos hechos iniciales tendrán gran influencia en las relaciones intermedias que se puedan originar, por lo que es importante que sean precisos, y que abarquen un rango de temas suficientemente amplio. Es por ello que crearlos manualmente no es viable y se necesita, por tanto, una forma de generar esos hechos automáticamente. En este punto entra en juego **factsGeneration.py**.

Este módulo se ejecutará de forma automática cada vez que se realice el entrenamiento de un modelo tal y como se explica en la sección 4.2. Tomará como entrada el mismo corpus de texto que se usa para entrenar el modelo y lo dividirá en sentencias usando la herramienta de análisis de sentimiento (sección 4.3). Este otro módulo es de especial utilidad, pues también es capaz de detectar y devolver los nombres propios que aparecen en cada frase. Estos se usarán como los términos principales sobre los que se van a establecer relaciones, ya que son más objetivos y suelen representar el mismo objeto o idea independientemente del contexto. Y eso no es todo, puesto que también se hará uso de la polaridad o ”sentimiento” devuelto

por el análisis para cada frase, y así poder saber si las palabras que aparecen en ella tienen una relación positiva o negativa.

A partir de aquí, basta con recorrer los nombres propios distintos que aparecen en cada frase, y asignar a cada par único el valor de positividad de la frase en la que se han encontrado. De forma que si, por ejemplo, aparecen las palabras *Madrid* y *Barcelona* en la misma proposición, y esta tiene un valor negativo, se puede intuir que la relación entre ambos términos también lo es.

Sin embargo, un par de nombres propios puede aparecer en múltiples ocasiones a lo largo de un corpus grande en la misma oración. Es por ello que a cada par de valores distinto se le asigna una lista a la que se le va añadiendo la polaridad de las frases en las que se encuentran. Una vez recorrido todo el corpus, simplemente se hace la media aritmética de esas polaridades para cada par, y así se obtiene una única que servirá como valor final de la relación entre esos términos. Como último paso, se comprueba la base de datos de hechos y se asegura de reemplazar las relaciones de pares de términos que ya existieran con sus nuevos valores, manteniendo las de pares que no han aparecido en el corpus, dando de esta forma prioridad al corpus entrenado más recientemente.

4.8. Unificación de entidades

Un problema añadido tanto a la selección de modelo de un bot 4.4.2 como al guardado de relaciones en la base de datos es que a veces las palabras, a pesar de estar describiendo o hablando de lo mismo, se escriben de formas distintas. Además, hay que tener en cuenta las posibles erratas. Por ejemplo, *fc barcelona*, *barça* y *barsa* representan el mismo concepto, pero se puede escribir de maneras diversas. Por todo esto, se hace necesario un sistema de *unificación* de entidades y conceptos.

Bajo este escenario, si existiera una base de datos pública con una entrada única para cada uno de los conceptos y entidades del mundo, y hubiera una manera de llegar a esas entradas de diversas maneras, entonces la tarea está hecha. Esa base de datos que cumple todas las premisas es *Wikipedia*. Y la manera de llegar a estas entradas en la base de datos, se puede hacer a traves de un servicio que comunmente se usa en la vida diaria de las personas, este es los *Web Browsers*.

Este es el mecanismo que se sigue para realizar la *Unificación de entidades*, se busca el objeto que se quiera, escrito de la manera que se quiera (con erratas incluidas), en el *Web Browser* junto a la palabra *Wikipedia*. Esto en lineas generales dirigirá a la misma entrada en la base de datos, aunque las palabras no sean las mismas. Una vez en la base de datos se coge la cabecera de la entrada, y se convierte en el concepto unificado.

4.9. Scripts

A continuación se describe el mecanismo mediante el cual la aplicación principal hace uso de los Bots (véase sección 4.4) y toda la funcionalidad que encapsulan. Para ello se utiliza

el archivo **scripts.py**, el cual se ejecuta con distintos parámetros a modo de "comandos". Esta lista de comandos está descrita en el archivo *commands.txt*, y es la siguiente:

- *create age level_of_education "likes" "dislikes"*: crea un Bot sin modelos asociados y con los datos proporcionados. Se devuelve un JSON con los datos del Bot.

- *trainModel name pathCorpus "modelDescriptorList" (-n numIterations)* : crea un nuevo modelo (véase sección 4.2), especificando su nombre en *name*, la ruta del corpus en *pathCorpus*, y opcionalmente, el número de iteraciones. Se devuelve la *modelDescriptorList* actualizada con el nuevo modelo añadido. Además, expande la base de datos de relaciones (véase sección 4.7) con los nuevos términos encontrados en el corpus.

- *trainBot "jsonBot" "modelDescriptorList"*: el bot selecciona los modelos de la *modelDescriptorList* que mas se ajustan a su personalidad (vease sección 4.4.3) y los deja asociados. Se devuelve el JSON del bot actualizado.

- *getResponse "jsonBot" "context" filterParams*: solicita una generación de texto al bot *jsonBot*, con un contexto *context*, y una serie de parametros de filtrado (vease sección 4.5.1).

- *setupBaseModel*: instala el modelo base.

Además se cuenta con las siguientes opciones generales:

- *–ascii-in*: Indica que todos los JSON de entrada están codificados en ASCII crudo.

- *–ascii-out*: Indica que todos los JSON de salida están codificados en ASCII crudo.

- *–outfile "filepath"*: Escribe la salida del programa en el archivo especificado en la ruta *filepath*

4.10. Generador de corpus

Un obstáculo encontrado a la hora de entrenar los bots fue la falta de corpus. Encontrar grandes cantidades de comentarios humanos sobre un mismo tema no es tarea sencilla, por lo que fue necesario crear un script que, gracias a una cuenta de desarrollador de Twitter, obtenga multitud de tweets sobre una palabra o hashtag especificados.

El script en cuestión se denomina `twitterCrawler.py`, y hace uso de la librería de Python tweepy que, como indica el nombre, permite la conexión con Twitter gracias a unas claves (consumer key y access token) que se pueden obtener gracias a la cuenta de desarrollador. Una vez conectada con esas claves, se especifica en el propio script el texto a buscar, el nombre del corpus de salida, y el script empieza a recibir tweets que contengan esa palabra.

Se ha adaptado de forma que filtre los tweets recibidos y no añada los que contengan links que podrían ser considerados spam, además de causar posibles problemas en el futuro

aprendizaje sobre ese corpus. Así como los retweets, los cuales se ignoran pues solo causan que haya una gran cantidad de mensajes idénticos en el corpus. Este código se ejecuta hasta que se hayan recibido 3000-5000 tweets sobre el tema especificado, y se puede volver a ejecutar con un nuevo tema para el mismo corpus y así ampliarlo más.

4.11. Setup e instalación de entorno Python

Lo primero que se debe hacer es configurar el entorno de ejecución. Para ello hay que instalar todos los requerimientos descritos en `requirements.txt`. Específicamente, se necesita la versión 3.7.0 (64 bits) de Python e instalar los siguientes paquetes: tensorflow==1.15.2, regex, requests, tqdm, numpy, toposort, spacy, spacytextblob y pyswip. Además, se debe instalar cualquier versión de SwiProlog.

A continuación se debe ejecutar el siguiente comando en la consola de Python:

```
spacy download en\_core\_web\_sm
```

Una vez hecho esto hay que instalar un modelo base para la generación de texto, esto se hace a través del script `setupBaseModel`. Simplemente se ejecuta el archivo `scripts.py` con el argumento `setupBaseModel`.

Capítulo 5

Arquitectura e Implementación

En este capítulo se expone y explica la estructura del código utilizado, así como los flujos y relaciones entre los diferentes módulos que lo componen, para la creación de la solución tecnológica aplicada para la propuesta de negocio de El Trol Versátil.

5.1. Arquitectura de la aplicación frontend

En esta sección se expondrá el stack tecnológico sobre el que se apoya la solución de la aplicación con la que interacciona el usuario, el frontend, los diferentes elementos que la compone explicando el cometido de las diferentes partes de este código, así como los patrones que las une.

5.1.1. Consideraciones previas en el desarrollo de una APP

Con el fin de salvar ciertos obstáculos elementales que no están directamente relacionados con la finalidad del cliente web, para el desarrollo de la aplicación frontend se ha utilizado un par de frameworks comúnmente visto juntos. En primer lugar se dispone del framework Open Source Ionic (en su versión 6), un Open Source que permite construir aplicaciones híbridas multiplataforma, para poder exportar la aplicación a plataformas con sistemas operativos como Android, o iOS a través de Cordova, o a su vez exportarlo a un cliente web alojable en cualquier servidor http/https. Es de agradecer que Ionic aporta potentes herramientas para maquetar la interfaz de forma responsive, lo que ayuda a ajustarse a cualquier cliente web desde dispositivos de distintos tamaños, ahorrando esfuerzo a la hora de crear aplicaciones móviles evitando invertir importantes esfuerzos en alcanzar aspectos estéticos y de usabilidad, permitiendo invertir ese esfuerzo en otros aspectos funcionales propios del proyecto. En segundo lugar, puesto que Ionic está basado en Angular, éste es el segundo framework utilizado (en su versión 11) fundalmentalmente por su focus en una arquitectura orientada a los componentes reutilizables a lo largo de diferentes módulos de la aplicación. TypeScript es elemental desde hace varias versiones de este framework para programar su lógica de negocio, aportando así más limpieza de código, tipado estricto

y herramientas de desarrollo entorno a ello (a menudo habilitadas gracias a IDEs como Visual Studio Code) que en último fin redunda en una mayor escalabilidad. Así, para la lógica de negocio se programará en TypeScript, para los estilos se utilizará SASS y para la maquetación el HTML, que además es precompilado por Ionic y Angular para aprovechar sus herramientas.

5.1.2. Estructura básica

La estructura básica de la aplicación frontend puede ser resumida en:

1. Componentes - si se piensa en segmentar o desestructurar una vista o sección de la aplicación, se encontrarán tantos de estos como elementos (potencialmente) repetibles (conocidos como *component*); servirán para aislar diferentes comportamientos que se vayan a requerir en una vista, y se puede replicar un mismo componente tantas veces como se desee en cualquier parte de la aplicación con diferentes parámetros a cada vez.

2. Vistas o secciones - habitualmente se encuentra un elemento de este tipo por cada vista de la aplicación (conocidos como *view* o *template*); sirven para cargar, presentar y distribuir la información requerida por dicha vista en la aplicación, y permitir la interacción del usuario con dicha vista, transformando sus acciones en ejecuciones de flujos lógicos que pasan por otros módulos de la aplicación.

3. Módulos de manipulación y gestión de datos - en general se tiene uno por cada estructura de datos de la aplicación (conocidos como *store*); su objetivo es el de proporcionar vías para otros módulos para obtener, almacenar o modificar elementos del tipo de datos específico de dicha store; ésta es la responsable última de obtenerlos, manipularlos o almacenarlos por cualquier medio existente.

4. Servicios para implementar APIs - pueden servir para interactuar con APIs externas o contra bases de datos locales (conocidos como *api provider/service* en el primer caso y *database service* en el segundo); generalmente se tiene uno (de cada tipo) por cada estructura de datos que maneje la aplicación; su cometido es ejecutar las llamadas a sendos servicios con los parámetros requeridos (bien sean APIs REST remotas o bases de datos locales) y controlar las respuestas retornadas.

5. Servicios de utilidades - sirven de herramientas utilizables en cualquier sitio de la aplicación, de forma tan independiente que potencialmente son reutilizables incluso entre aplicaciones que tuvieran características o funcionalidades similares (como si de una librería externa se tratase), (conocidos como *service*); aportarán utilidades puntuales con un nivel de abstracción suficiente como para que el üsuario"no necesite conocer su funcionamiento a bajo nivel, interactuando contra métodos explícitos, sencillos y convenientes.

En el punto actual de la aplicación, sólo se encuentra un tipo de estructura existente, ésta es la red o grupo de trolls, que será conocida como *trollnet*. Por lo general, se observará que las menciones a diferentes módulos son en su traducción al inglés, lo que conviene por facilidad a la hora de manejarse en el flujo del código programado.

5.1.3. Patrón del flujo de datos

A continuación se explican los patrones aplicados en el código de la parte frontend, donde se verá que se ha intentado cumplir los principios de separación de responsabilidades y capas de abstracción.

Las vistas existentes en la aplicación presentan datos de un tipo de estructura u otro. El template mostrará la información requerida para la funcionalidad de esta vista, adaptándola o filtrándola según se desee exponer; el template obtiene esa información pidiéndosela a la store. A partir de ese momento se crea una capa de abstracción conveniente ya que, por un lado, no es necesario que el template se preocupe de dónde obtener esa información, delegando esa responsabilidad en la store, de forma que permite que el código de la vista sea más limpio puesto que sólo debe preocuparse por cómo mostrar los datos o atender a la interacción del usuario; por otro, se podrá reaprovechar la misma lógica que ofrece la store para este fin en cualquier vista.

La figura B.1 del Apéndice B es un buen ejemplo para explicar esto. Se puede ver como el template de la vista *My Trollnets* muestra una lista de todas las trollnets existentes. Tras adquirir el array de trollnets a partir de la store de trollnets, el template se encarga de mostrar un componente por cada trollnet en la lista de trollnets. Puesto que no se conoce la longitud de la lista de antemano, y puede cambiar en cualquier momento, la implementación necesita ajustarse a esa dinámica; Angular trae las herramientas necesarias en este contexto: la directiva *ngFor* permite generar tantos elementos como existan en el array de trollnets (independientemente de su tipo), pudiendo instanciar así un elemento visual por cada trollnet, cada uno con la información específica de dicha trollnet.

Así mismo, la figura permite observar que los servicios son usados tanto por lógicas asociadas a las vistas como por las stores (e incluso por otros servicios también). El LoadingService, como ejemplo de estos servicios, ofrece un spinner de carga junto a un mensaje como feedback para el usuario de que un proceso está cargando. Este servicio independiente es quien se encarga de aportar y ejecutar el código necesario para que dicho spinner se muestre; sin embargo no es quien conoce el estado de la vista, por tanto quien sabe si los datos del template han sido pedidos y están en curso de ser recibidos será quien deba mandar la orden para mostrar el spinner, y ese será el módulo que invoque al LoadingService. De la misma forma, este mismo módulo se encargará de ordenar ocultar el spinner cuando los datos hayan sido finalmente recibidos.

La figura B.2 del Apéndice B muestra el patrón que se utiliza para la obtención de datos. El flujo más crucial del frontend pasa por la store. Cada vez que una vista, componente u otro módulo desee interacción con un elemento del tipo manejado por la store, por ejemplo cada vez que se pida a una trollnet que genere una conversación entorno a un hilo, se pedirá la información de esa trollnet o se lanzará esa actuación a través de la trollnet store.

En ese momento, la store será quien decida de dónde obtener la información requerida de esa trollnet, o a qué externo (API o base de datos local) lanzar la petición para generar la conversación. La store se encarga de hacer las veces de manejador/distribuidor de la información del tipo que maneja.

En pro de la separación de responsabilidades, conviene aquí un módulo api provider, un servicio que al estar diseñado específicamente para manejar el mismo tipo de datos que la store, extiende las capacidades de la store permitiendo interactuar con la API remota de este tipo. De este modo, si la trollnet store desea obtener la lista más actualizada real de trollnets existentes y su estado, usará el método público del trollnet service `getAllTrollnets`; cuando el usuario del frontend quiera añadir una nueva trollnet, así mismo hará uso del método público `createTrollnet` del trollnet service para lanzarle dicha petición al servidor. Dependiendo de cada petición http lanzada desde este servicio a la API remota, se incluyen diferentes características propias de este protocolo, como el tipo CRUD, los headers, así como características definitorias de la API remota, como la url que atenderá la petición y el contenido del cuerpo de dicha petición; estos detalles intrínsecos a cada petición, al crearse esta nueva capa de abstracción (y no simplemente introducir este código también en la store) permite que este servicio se comporte como una caja negra de cara a la store, que no requiere esos detalles para su finalidad de manipular y distribuir la información, y así simplificar el desarrollo, depuración y escalabilidad de cada módulo por separado por la claridad que supone esta separación de responsabilidades. El requisito que esta capacidad implica es que todos los servicios que contactan con la API remota (se encontrará un por cada endpoint existente en la API remota, en general uno por cada tipo de datos importante), deben heredar de la clase ApiGenericProvider, implementando cada método CRUD existente en la API remota que se quiera aprovechar. La figura B.3 del Apéndice B expone, para todas las APIs creadas, el diagrama de clases correspondiente.

En paralelo se encontrará también un database service que manejará la misma estructura de datos que la store, que extenderá (de la misma forma independiente y abstracta que el servicio proveedor de la API remota) la capacidad de la store de comunicarse con la base de datos local cuando quiera recuperar o modificar una entrada del tipo de datos específico. Hay varias bases de datos existentes posibles al alcance de un proyecto web como éste; se hará uso de la base de datos IndexedDB, propia del motor web del dispositivo (así como de LocalStorage, como backup). IndexedDB es una base de datos No-SQL, orientada a almacenar datos de entradas de gran tamaño, y no bloquea la entrada-salida del hilo principal. Esto encaja muy bien para tratar con objetos JSON, con posiblemente estructuras y tamaños diferentes entre sí. Se observará que cada database service está creado apuntando específicamente a una tabla y ninguna más, con el mismo nombre del tipo de datos manejado por este database service y su store asociada, reforzando así la separación de responsabilidades y la mantenibilidad y escalabilidad del código. Esta separación es conveniente ya que da la posibilidad de intercambiar el servicio de base de datos utilizado, la codificación o características particulares que requiera cada uno sin afectar al flujo de la store, que delega esa responsabilidad en el database service, y se limita en cualquier caso a interactuar con sus métodos públicos. El database service garantizará la persistencia, inmutabilidad y accesibilidad en cualquier momento a los datos que se le confían. A diferencia de los servicios proveedores

de la API externa, no se observará que los database service hereden de un mismo módulo pero todos hacen uso del servicio StorageService independiente para crear utilizar la herramienta Storage de Ionic y Angular, que simplifica mucho la interacción a bajo nivel con las BBDD mencionadas. Se expone la clase que sigue TrollnetDatabaseService en la figura B.4 como ejemplo que abarca los métodos más comunes de interacción con su base de datos. Se encontrarán los diagramas para el resto de database services en el apartado 5.1.4.

5.1.4. Inicialización de la aplicación frontend

Cuando se inicia la aplicación, se inicializan los frameworks de Ionic y Angular, y se lanza el módulo base que importa todos los demás módulos implementados antes de ser utilizados por primera vez, `app.module.ts`. En otras características, en este momento es cuando se utiliza el `AppRoutingModule`, módulo donde se especifican qué rutas (en la práctica, generalmente una por cada vista) existen y qué modulo debe cargarse en cada una de ellas.

Después, se crea el `app.component.ts` que compone la base visual de la aplicación: se encontrará la lógica correspondiente a un menú lateral con una serie de elementos con un título y descripción que, según sea el seleccionado, se cargará a la derecha el módulo que corresponde a la ruta que invoca.

Si se desease añadir funcionalidades a la aplicación, como soporte a sesiones de usuarios, ciertos flujos deberían llevarse a cabo en este momento. Por ejemplo, si se debiera detectar si la aplicación tiene un token válido para mantener iniciada la sesión de un usuario, debería hacerse a esta altura, para decidir a posteriori si se muestra una vista de inicio de sesión o registro, o si por el contrario se puede proceder a mostrar vistas y funcionalidades de la aplicación a las que este usuario sí tenga acceso.

5.1.5. Diagramas de clases de las Stores y Models

La trollnet store gestiona todo lo relacionado con la lista de trollnets disponibles, por tanto tiene una importancia que merece una explicación más detallada (véase figura B.5 del Apéndice B). Durante la inicialización de la aplicación, se llama al método privado `_synchronizeData` para hacer uso del TrollnetService que permite obtener la lista de todas las trollnets mediante `getAllTrollnets`. Como norma genérica aplicable a cualquiera de las stores, cada vez que se interactúe con la API remota a través de cualquiera de sus métodos, se suele llevar a cabo un flujo común y conveniente: se recibir la información del trollnetService contra la API, se guarda en la trollnetDB con `_saveTrollnetsIntoDB`, el cuál llama a su vez al método `refreshList`, que extrae el valor recién actualizado de la trollnetDB y actualizar tanto el array privado `_currentTrollnets` como el observable `_currentTrollnetsObservable` con su nuevo valor.

Cabe destacar que este último paso es muy importante para el flujo de datos en esta aplicación: se usa el patrón Observer/Subscribe, uno de los patrones reconocidos más útiles de los que hacemos uso en la aplicación frontend. Esta funcionalidad nos viene a través de la librería RXJs y permite que cualquier flujo que desee antes o después acceder a la

lista de trollnets, se suscriba al observable _currentTrollnetsObservable mediante el método trollnetsChange de la trollnet store. Así, cada vez que la trollnet store actualice este observable, se notificará dicho cambio dinámicamente a todos los suscriptores de dicho observable en ese instante, enviándoles el nuevo valor para que ejecuten el flujo que necesiten en ese instante.

Una vez este proceso inicial dota a la aplicación del estado más reciente de las trollnets, con su método _startPollingStatus se inicia un proceso de actualización periódica: cada 10 segundos, y mientras la aplicación frontend esté en funcionamiento, se llamará a _pollStatus, el cual reúne en una lista los identificadores de aquellas trollnets no completamente entrenadas todavía (discriminando a través de su parámetro trainingStatus) si, las hay, le pasa dicha lista al método getUntrainedStatus del TrollnetService para obtener el último estado de creación y de entrenamiento de cada una de las redes y así ir actualizando periódicamente el progreso de cada una de ellas. De nuevo, se produce el proceso encadenado descrito anteriormente con _saveTrollnetsIntoDB y refreshList.

La trollnet store expone también los métodos createNewTrollnet, deleteTrollnet, renameTrollnet, activateTrollnet y deactivateTrollnet. Todos ellos son autodescriptivos, interactuan con la API para accionar su cometido y actualizan el frontend según se requiera en cada caso. Si cualquier parte del flujo fallase (ya sea en frontend o backend), se rechazan las partes del flujo que pudieran conllevar datos inconsistentes entre sí y el pretexto es que el backend siempre tiene la información fiable, que se obtendrá de nuevo en cualquier caso con la siguiente petición _synchronizeData. Por tanto, es similar a una transacción, garantizando que los datos de la aplicación son coherentes.

Se observará que los diagramas de clase, con el fin de permitir los mecanismos necesarios para cada caso de uso existente, demuestran modelos para cada tipo de datos que son claros y concisos (véase figura B.6 del Apéndice B).

1. Para la clase TrollnetModel, se requiere un id como identificador único de la trollnet, un booleano isActive para describir cuándo la red está activada, un número entero creationStatus que determina el porcentaje de avance en la creación de la trollnet y sus bots (0 al comienzo de la creación, 100 cuando está completado y -1 si se ha abortado por un fallo), un número entero trainingStatus que determina el porcentaje de avance en el entrenamiento de sus bots (0 al comienzo del entrenamiento, 100 cuando está completado, -1 si se ha abortado por un fallo), una fecha lastTrained de cuándo se entrenó a sus bots por última vez, un array de strings botList que contiene todos los identificadores de bots asociados a la trollnet, un campo properties que contiene un objeto que sigue el modelo TrollnetDraftModel (descrito a continuación) y que contiene todas las propiedades que caracterizan a una trollnet y un campo de tipo any error para almacenar el último error que pudo ocurrir en el proceso de creación o de entrenamiento de la trollnet, con fines correctivos.

2. Para la clase TrollnetDraftModel, se requieren una serie de campos que describirán las características tanto de la red en sí como de los bots de dicha red. Así, se requiere un customName como nombre personalizado de dicha trollnet, un genders que contendrá

los géneros sexuales posibles, el `ageInterval` para el intervalo de edades posibles, un campo `ethnicity` las etnias posibles, un objeto `culturalLevel` para el intervalo de niveles de educación/niveles culturales posibles, un campo `moodLevel` para el intervalo de positividad o negatividad posible, un campo `keywords` para contener una lista de palabras clave genéricas sobre las que aprenderán los bots, dos campos `likes` y `dislikes` que contendrán listas de palabras clave sobre las que aprenderán los bots y además asociarán a sentimientos de aprecio o rechazo respectivamente, un campo `netsize` que describirá el tamaño exacto de la red en número de bots, y por último el campo `interactionLevel` que será útil para calcular el tamaño de las respuestas que aporten los bots. donde almacenar los cálculos generales extraidos de sus sensores asociados a la room. Salvo el `customName` y el `netsize`, el resto de características usará o bien el modelo `MultipleStringsModel` o bien el `NumberIntervalModel`, descritos a continuación.

3. La clase `MultipleStringsModel` es una clase auxiliar que aporta utilidad, tiene un campo `values` que contiene un array de strings.

4. La clase `NumberIntervalModel` también es una clase auxiliar de utilidad, ésta tiene un campo `values` que contiene un objeto con 2 propiedades: `lower` y `upper`, conteniendo dos valores enteros que limitan inferior y superiormente un intervalo de números enteros.

5.1.6. Diagramas de clases de los Services

A continuación, se describen los servicios a través de sus diagramas (véase figura B.7 del Apéndice B).

El `alert.service.ts` y el `toast.service.ts` son servicios que aprovechan las funcionalidades de Ionic y Angular de *alertas* y *toasts* respectivamente. En sus constructores se encuentra la configuración por defecto necesaria para generarlos según se prefiera y exponen métodos sencillos para usarlos con diferentes características, `showAlert` y `showToast` respectivamente.

El `loading.service.ts` es otro servicio de la misma categoría que ofrece el típico spinner de carga con mensaje adjunto como feedback al usuario de que un proceso está teniendo lugar y le invita a esperar. Está implementado de forma que sea único por toda la aplicación, y que el módulo que lo puede disipar (con `dismiss`) sea distinto al que puede instanciarlo (con el método expuesto `show`) si se determina que tiene mayor autoridad para hacerlo. También se puede modificar su mensaje adjunto sobre la marcha con `setContent` para poder modificarlo sin tener que volver a instanciarlo.

El servicio `storage.service.ts` se utiliza a lo largo de toda la aplicación desde cada database service existente para crear almacenamientos particulares para cada uno a través de `create`. Una vez existe dicha base de datos, desde cada database service se interactua con esa base de datos a través de `get`, `set`, `getAll`, `remove` y `removeAll`. Internamente hace uso del servicio de `Storage` de Ionic y Angular, aprovechando las bases de datos ofrecidas

por el motor del navegador, en la mayoría de casos siendo preferentemente IndexedDB y como backup LocalStorage. Se ha añadido una funcionalidad adicional de encriptación de los datos para mayor seguridad con la librería `crypto-js`, encriptando y desencriptando cada entrada al almacenarla o recuperar respectivamente.

5.2. Arquitectura del servidor de NodeJS

En esta sección se expondrá el stack tecnológico sobre el que se apoya el backend, la infraestructura que vertebra y comunica el frontend con las ejecuciones de scripts de Python (tanto para los procesos relacionados con los bots como para los relacionados con las redes sociales en los que publican). Se explican los diferentes elementos que lo componen, así como la finalidad de cada uno y los patrones que los relaciona entre sí.

5.2.1. Consideraciones previas

Para desarrollar el proyecto que contiene al servidor se ha elegido el MEAN stack (por sus siglas MongoDB, ExpressJS, Angular y NodeJS). Esto viene justificado por los beneficios que aportan sus herramientas: MongoDB, por ser una base de datos no relacional, permite un desarrollo fluido, escalable y no ajustado a diseños arquitectónicos rígidos, permitiendo almacenar objetos JSON sin modelos fijos que puedan ir variando a lo largo del desarrollo permitiendo convivencia entre datos nuevos y antiguos con diferentes estructuras; ExpressJS facilita enormemente el trabajo de implementar una API REST en NodeJS sin los obstáculos que podría presentar éste a priori (tratamiento de peticiones http con herramientas poco intuitivas); estas elecciones se alinean con Angular en el frontend ya que se desarrolla en un mismo idioma o estilo (JS y TS); y NodeJS aporta la existencia de infinitas librerías de sencillo uso, presentes a lo largo de toda la comunidad de usuarios, altamente probadas y actualizadas, dando muchas herramientas que allanan el terreno para centrarse en las implicaciones directas del desarrollo, y cuyos beneficios se enumerarán más adelante.

5.2.2. Estructura básica

En términos generales, se puede describir el proyecto backend compuesto por:

1. Enrutadores, se encontrará uno por cada endpoint de la API REST (se conocerán como *routers*); cada petición http recibida en el servidor se encaminará a través de ellos dependiendo de su tipo CRUD (GET, POST, PUT o DELETE) y la estructura de la URL para ser atendida por el método específico que debe procesar la petición en cuestión.

2. Controladores, se encontrará al menos uno por cada endpoint de la API REST por lo general habrá uno por cada nombre en la API REST (se conocerán como *controllers*);

en su contexto, se procesan las peticiones que se hayan encaminado hacia ellos, manipulando los datos de la estructura de datos asociada y también interactuando con otros controllers dependiendo de la complejidad de la operación requerida.

3. Modelos, que igual que en el frontend, se encontrará al menos uno por cada estructura de datos existente (se conocerán como *models*); establecerán la estructura JSON mínima que debe tener un objeto para ser almacenado en una colección de MongoDB. Así mismo se utilizará para generar objetos de ese tipo así como para comprobar que un objeto pertenece a ese tipo gracias a las herramientas de MongoDB.

4. Servicios, que de forma similar a la aplicación frontend, expondrán cómodas APIs para atacar a diferentes servicios como la base de datos de MongoDB, el sistema de archivos del servidor, o la ejecución controlada y sincronizada de scripts de python (se conocerán como *providers* o *services*); su grado de independencia de la aplicación es tan alto que podrían ser copiados y utilizados tal cual en otro proyecto (salvando ligeras modificaciones que están adaptadas a cada proyecto por comodidad). Su existencia separada del resto del código es justificada por la capa de abstracción adicional para delimitar la responsabilidad del módulo que lo invoca y reducir su complejidad, dejando detalles de implementación del servicio en el propio servicio, ocultando su complejidad al invocador.

5. Módulos de soporte, también independientes y por lo general se ocupan de operaciones tediosas de cálculo o manipulación de estructuras de datos que puedan distraer la atención si estuvieran presentes en el código del módulo invocador (se conocerán como *helpers*); exponen sus métodos a cualquier módulo de la aplicación (por lo general a controllers).

6. Archivos de configuración y de constantes, para centralizar determinados datos relevantes de configuración, para su fácil acceso y modificación.

Para entender con más facilidad la alineación del proyecto backend con el frontend y su esquema, se utilizará una similar para cada tipo de datos.

5.2.3. Flujo de enrutado de las peticiones HTTP

Como ya se explicó anteriormente, se utiliza ExpressJS para enrutar las peticiones que se dirijan desde cualquier origen a la aplicación frontend. Cuando se inicializa el servidor, a través de ExpressJS se genera un servidor que atiende todas las peticiones recibidas en el puerto (por defecto el 3000) e IP local que se configure mediante el método `listen` de su objeto `app` en el `server.js`.

A posteriori, se configura el enrutador con el método `setupRouting` del módulo `router.js`. Éste es responsable de, mediante el método `use` del objeto `app` importar todos los routers creados y disponer cada uno de ellos de forma vinculada a cada endpoint asociado. De esta forma, cuando se reciba una petición con la ruta `/api/trollnet`, se redirigirá dicha petición para ser atendida por el router `trollnetRouter` con

`use(ROUTER_CONFIG.EP_GLOBAL.TROLLNET, trollnetRouter)`

Se puede observar esta primera etapa de encaminamiento de peticiones en la figura B.8 del Apéndice B.

Tras encaminar una petición al router que debe tratarla, entra en juego la segunda etapa de encaminamiento, en la que dicho router importará el controller asociado a su tipo de datos para encaminar así cada petición según su tipo y su contenido al método del controller que deba atenderla. Gracias a ExpressJS se puede registrar la función del controller a ejecutar para determinadas características, como el tipo CRUD y las características de su url. De esta forma, suponiendo que el `trollnet.router.js` atiende una petición PUT en la url `/api/trollnet/rename/:id`, mediante el método `put` se podrá encaminar dicha petición a ser atendida por el método `renameTrollnet` del `trollnet.controller.js`. La figura B.9 del Apéndice B muestra este segundo nivel de encaminamiento para cada enrutador, junto a una breve descripción.

Para el endpoint `/api/trollnet`, se asocian 5 entradas atendidas por el `trollnet.router.js`:

1. En la ruta `/api/trollnet/`, las peticiones GET serán atendidas por el método `getAllTrollnets`.
2. En la ruta `/api/trollnet/`, las peticiones POST serán atendidas por el método `addTrollnet`.
3. En la ruta `/api/trollnet/:id`, las peticiones GET serán atendidas por el método `getTrollnetById`.
4. En la ruta `/api/trollnet/:id`, las peticiones DELETE serán atendidas por el método `deleteTrollnet`.
5. En la ruta `/api/trollnet/status`, las peticiones POST serán atendidas por el método `getUntrainedTrollnetsStatus`.
6. En la ruta `/api/trollnet/rename/:id`, las peticiones PUT serán atendidas por el método `renameTrollnet`.
7. En la ruta `/api/trollnet/activate/:id`, las peticiones PUT serán atendidas por el método `activateTrollnet`.
8. En la ruta `/api/trollnet/deactivate/:id`, las peticiones PUT serán atendidas por el método `deactivateTrollnet`.

Las peticiones que no encajen con algún segmento del primer nivel y del segundo nivel serán automáticamente rechazadas. Se controla de esta manera muy severamente el camino de cualquier petición que atienda el servidor, constituyendo una medida de seguridad importante sobre interacciones externas con esta aplicación, ya que se conoce y controla cualquier posible entrada al servidor por un único camino específico para cada petición y así minimizar brechas de seguridad.

5.2.4. Diagramas de clases de los Controllers, Helpers y Models

El `core.controller.js` es un módulo central cuyo cometido es inicializar los procesos demonio de otros controladores una vez la conexión con la MongoDB se ha llevado a cabo, para sincronizar las acciones de dichos controladores con la información existente en la base de datos una vez se ha iniciado el servidor NodeJS y éste haya conectado su cliente de Mongo con la instancia de MongoDB. Al ejecutarse su método público `setupCoreMonitors`, este módulo lanza `_releaseDaemons` iniciando los *daemons* de cada controlador (`startTrollnetDaemon` del `trollnet.controller.js`, `startCorpusDaemon` del `corpus.controller.js` y `startModelDaemon` del `model.controller.js`). Con esto, finaliza la preparación inicial del servidor.

El `trollnet.controller.js` debe manipular todo entorno a las trollnets, y por ello es quien ejecuta `startTrollnetDaemon` además de exponer todos sus métodos para atender a la API REST. El `startTrollnetDaemon` inicia un proceso de comprobación de redes pendientes por entrenar. Para ello, la operación sigue los pasos aquí descritos:

1. Al ser lanzado cuando el servidor está listo, se quiere comprobar qué trollnets están guardadas en la base de datos como creadas pero no totalmente entrenadas, y para ello se accede a la MongoDB con su atributo `creationStatus` con valor 100 y su atributo `trainingStatus` con valor inferior a 100. Una vez tiene la lista, llama a `_checkUntrainedTrollnets` con ella.

2. `_checkUntrainedTrollnets` une la lista obtenida en la base de datos a cualquier posible trollnet que se hubiera creado entretanto, e inicia el proceso recursivo `_followTrollnetTraining` para entrenar las redes pendientes de una en una.

3. `_followTrollnetTraining` en cada nivel (por ser recursivo) irá llamando a `_teachTrollnet` con una de las trollnets pendientes. Lo primero que hace este método es actualizar en la MongoDB el trainingStatus de esa trollnet con valor 1. Acto seguido, extrae los datos de esa trollnet, de entre los cuales, la lista de bots e invoca al método `teachBotArray` del `botController` con ella. Su cometido es entrenar a cada uno de los bots de uno en uno, pero esto se explicará en el apartado del controller de bots.

4. Si devuelve algún fallo, se actualizará en la base de datos la red con trainingStatus -1 así como almacenado el error para ser revisada la causa; si retorna exitosamente, se actualizará en la base de datos la red con trainingStatus 100, y será indicador de que la red ha sido entrenada exitosamente.

5. Al acabar, `_followTrollnetTraining` vuelve a invocarse a sí mismo para ejecutarse de nuevo con la siguiente trollnet, hasta que no queden más por entrenar. En ese momento, vuelve a `_checkUntrainedTrollnets` que se encarga de configurar un temporizador para volver a repetir estos mismos pasos más tarde, en caso de que se hayan creado nuevas trollnets.

El método `getAllTrollnets` extrae de la MongoDB la lista de trollnets existentes, y si funciona y las hay resuelve la petición http asociada con un código 200 y la lista de trollnets

en el body de la respuesta; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El método `getTrollnetById` extrae de la MongoDB la trollnet con el campo id igual al parámetro id en la URL de la petición http que lo invocó, y si funciona resuelve la petición http con un código 200 y la trollnet correspondiente en el body de la respuesta; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El método `addTrollnet` se encarga de inicializar una nueva trollnet en la DB, notificárselo de vuelta al frontend y empezar el proceso de creación de los bots asociados, así como de vincularles las cuentas disponibles para publicar sus mensajes. Se puede desgranar este proceso con detalle:

1. Se crea un objeto del tipo Trollnet con un id generado aleatoriamente, el campo creationStatus con valor 1, el campo trainingStatus a 0, el campo isActive a false, un array vacío en el campo botList y el campo properties contendrá el body de la petición http. A continuación se procede a salvar este objeto en la colección de Trollnets de la MongoDB para inicializar la trollnet.

2. Si falla, se devuelve un código 500 junto a un mensaje de error; lo habitual es que funcione, de forma que se resuelva la petición http con un código 200 y la nueva trollnet creada en el body de la respuesta, además de llamar al método `_createTrollnetBots`.

3. `_createTrollnetBots` invoca primero al método `createBotArray` del `botController`. Su cometido es crear una lista de bots que irán asociados a esta trollnet con las características del campo properties, pero esto se explicará en el apartado del controller de bots. Una vez ese método finalice, si ha fallado, se llamará a `updateCreationStatus` de `trollnet.helper.js` con valor de fallo (-1) y el error asociado para almacenarlo. Si el proceso se completa correctamente, devuelve un array de identificadores de bots que irán almacenados en el campo botList de esta nueva trollnet, y se actualiza la trollnet en la base de datos con dicha lista.

4. El último requisito para completar la creación de la trollnet es que se asigne una cuenta de Twitter a cada bot de la trollnet; por ello se invoca con la lista de bots al método `linkAccountsToBots` del servicio `PythonService`. Con su resultado, se llamará a `updateCreationStatus` de `trollnet.helper.js` con valor 100 si ha funcionado, o en caso de fallo, con valor de -1 y el error asociado para almacenarlo.

5. Si funciona, por último se apunta esta trollnet como red creada pero sin entrenar en el método `_addUntrainedTrollnet`.

El método `deleteTrollnet` elimina de la MongoDB la trollnet con el campo id igual al parámetro id en la URL de la petición http asociada, y además, llama al método `deleteBotArray` del `botController` que se encargará de eliminar las instancias relacionadas con cada uno de los bots de la trollnet a eliminar. Si todo se completa correctamente, resuelve la petición http con un código 200 y un mensaje de éxito; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El método `getUntrainedTrollnetsStatus` recibe como parámetro una lista de ids de trollnets teóricamente sin entrenar, extrae de la MongoDB la lista de trollnets correspondientes a esos ids, y si funciona correctamente, genera un mapa a partir de ellas, apuntando sus parámetros id, creationStatus y trainingStatus, y resuelve la petición http con un código 200 y dicho mapa; si falla, devuelve un código 500 junto a un mensaje de error.

El método `renameTrollnet` modifica en la MongoDB la trollnet con el campo id igual al parámetro id en la URL de la petición http asociada, sustituyendo el valor de su campo customName dentro del campo properties por el proporcionado en la petición http, y si funciona, resuelve la petición http con un código 200 y un mensaje de éxito; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El método `activateTrollnet` modifica en la MongoDB la trollnet con el campo id igual al parámetro id en la URL de la petición http asociada, actualizando el valor de su campo isActive a true, y si funciona, resuelve la petición http con un código 200 y un mensaje de éxito, y además lanza el método `_prepareTrollnetConversation` con el parámetro targetAccount especificado por el usuario en el frontend para generar una conversación con esa trollnet sobre ese perfil de Twitter; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El método `_prepareTrollnetConversation`, primero, invoca a `_listenToTwitterAccount` con la cuenta de Twitter a escuchar, el cuál llama al homónimo del `PythonService`. Cuando éste retorna, vuelve con el identificador del mensaje y el contenido del mismo. Aquí, recupera de la MongoDB qué trollnet va a generar la conversación, y llama al método `createConversation` del `threadController` con el identificador de la trollnet, el contenido del mensaje al que responder, el identificador de ese mensaje y con la lista de bots de esa trollnet.

El método `deactivateTrollnet` modifica en la MongoDB la trollnet con el campo id igual al parámetro id en la URL de la petición http asociada, actualizando el valor de su campo isActive a false, y si funciona, resuelve la petición http con un código 200 y un mensaje de éxito; de lo contrario, devuelve un código 500 junto a un mensaje de error.

El `bot.controller.js` debe manipular todo entorno a los bots. Expone métodos tanto para crear una lista de bots como para entrenarlos, además de un método para que generen un mensaje entorno a un tema. Este controlador no tiene métodos que atiendan a la API REST puesto que de momento no existe enrutador para ello ni funcionalidades proyectadas entorno a modificar directamente un bot. El `createBotArray` inicia un proceso de creación de bots a partir de una serie de parámetros. Para ello, extrae el valor netsize de los parámetros e invoca a `_followBotCreation` con ello. Éste es un método recursivo que a cada nivel de profundidad creará un nuevo bot y, cuando haya finalizado totalmente, devolverá un array con la lista de identificadores de los bots creados. Para cada nivel, la operación sigue los pasos aquí descritos:

1. Se invoca a `_getRandomParams` con el objetivo de parametrizar aleatoriamente las características del bot que se va a crear dentro de los límites establecidos para cada parámetro de la trollnet. Se explicará el proceso de `_getRandomParams` más adelante en este mismo apartado.

2. Se invoca `_createBot` con los parámetros generados para el bot. Este método en primera instancia se encarga de llamar al script de Python que genera un bot a partir de algunos de los parámetros especificados gracias al método `createBot` del servicio `Python` (explicado en la sección 5.2.5). Si la generación del bot es exitosa, entonces crea un objeto del tipo Bot con un id generado aleatoriamente, un botName por defecto, un campo creationDate con la fecha del instante en que ha sido creado, un campo properties con sus parámetros generados previamente, y además un campo AIObject que contiene lo devuelto por `createBot` del servicio `Python`.

3. Cuando ha terminado, almacena este objeto Bot en la colección de bots de la MongoDB, y al volver a `_followBotCreation` almacenará el identificador del bot en una lista. En este momento, actualizará el indicador de progreso de creación con el método `updateCreationStatus` del `TrollnetHelper`, y luego comprobará si ese era el último bot por crear. Si no lo es, se invocará a sí mismo para crear un nuevo bot; si era el último requerido, finalizará toda la operación devolviendo la lista de identificadores de los bots creados.

El `teachBotArray` inicia un proceso para enseñar a una lista de bots acerca de los temas que les interesan. Para ello, se siguen los pasos aquí descritos:

1. Se invoca al `getModelDescriptorListFromDB` del `ModelController` para obtener la *modelDescriptorList*, un objeto del que se hablará más adelante, requerido para los siguientes pasos de esta operación.

2. Se invocará entonces a `_followBotTeaching`, un método recursivo que a cada nivel de profundidad enseñará a un bot de la lista. Para ello, invocará al método `_teachBot`.

3. `_teachBot` se encarga primero de extraer los datos del bot de la colección de bots de la MongoDB con el identificador del bot. Después, de dichos datos, extraerá el campo AIObject y junto al modelDescriptorList, invocará con ello al método `teachBot` del servicio `Python`. Se explicará en la sección 5.2.5 el cometido de este método que, si ocurre con éxito, devuelve el objeto del campo AIObject actualizado, y se actualizará ese campo en su entrada de la MongoDB, momento a partir del cual ese bot estará preparado para "hablar.ªcerca de los temas que se le ha enseñado.

4. Después, `_followBotTeaching` actualizará el indicador de progreso de enseñanza con el método `updateTrainingStatus` del `TrollnetHelper`, y luego comprobará si ése era el último bot por enseñar. Si no lo es, se invocará a sí mismo para enseñar al siguiente bot; si era el último por enseñar, finalizará toda la operación y se lo notificará al proceso invocador.

El `deleteBotArray` elimina de la base de datos de Bots y de la de BotUsers aquellas entradas con el identificador `id` y el `botname` respectivamente en la lista de identificadores de bots que le llega por parámetro.

El `answerThread` es un método que a partir del identificador de un bot y un mensaje al que responder (identificador de mensaje + contenido), generará y publicará una respuesta del bot especificado al hilo en cuestión. Sigue los siguientes pasos:

1. Con `_getBotFromDB` obtendrá la instancia de dicho bot a partir de la base de datos; después, obtendrá los parámetros de filtrado específicos del bot con `_getFilterParams` a partir del parámetro interactionLevel de sus properties.

2. con esta información procederá a invocar el método `answerThread` del servicio `Python` (explicado en la sección 5.2.5), que recoge la propiedad AIObject del bot, el contenido del mensaje al que responder y los parámetros de filtrado mencionados.

3. Cuando se retorna de esa ejecución con la respuesta generada, se invoca al método `publishMessage` del servicio `Python` (explicado en la sección 5.2.5), que recoge tanto el identificador del mensaje al que responder, la respuesta generada y el identificador del bot que debe publicar el mensaje.

4. Esta ejecución retorna con el identificador del mensaje publicado, para que pueda retornarse al proceso invocador con esta información (junto con la respuesta generada), con el fin de ser aprovechada (si es necesario) por el siguiente interlocutor en la conversación.

El `corpus.controller.js` se encarga de manejar el procesamiento de archivos de corpus. Así, cuando se invoca su demonio `startCorpusDaemon`, a través del método `onNewFileAdded` del servicio `FileSystem` (explicado en la sección 5.2.5) configurará un listener con un callback que se ejecutará cuando el servidor encuentre un nuevo archivo de corpus en la carpeta especificada: `C:\Proyectos\etv-tfg\ETV-models-and-bots\ETV\rawCorpus`. Cuando esto ocurra, el callback llamará al método `_createCorpus` con la ruta del nuevo archivo. El `_createCorpus` se encarga de guardar en la colección corpus de la MongoDB una nueva entrada de tipo Corpus, con un campo id generado aleatoriamente, el nombre del archivo en un campo fileName y un flag isNewCorpus para señalar si este corpus es nuevo o si ha sido procesado ya posteriormente. El `getAllNewCorpuses` se encarga recuperar de la colección corpus de la MongoDB todas aquellas entradas que tengan el flag isNewCorpus a true. El `markAsUsed` se encarga de, a partir de un identificador de corpus, actualizar en la base de datos su flag isNewCorpus a false, para indicar que éste ya ha sido procesado.

El `model.controller.js` se encarga de manipular todo entorno a modelos entrenados o sin entrenar. Así, cuando se invoca su demonio `startModelDaemon`, se inicia un proceso de comprobación de corpus pendientes por ser procesados para obtener modelos a partir de ellos. Para ello, la operación sigue los pasos aquí descritos:

1. Se invoca el método `_checkPendingCorpuses` que se encarga de comprobar si hay corpus nuevos, pendientes sin procesar. Para ello, invoca a `getAllNewCorpuses` del controller `CorpusController`. Si los hay, se inicia el proceso recursivo `_followCorpusTraining` para entrenar los modelos a partir de ellos. Si no los hay, programa un temporizador para volver a comprobarlo más tarde.

2. `_followCorpusTraining` en cada nivel (por ser recursivo) irá llamando a `_trainModel` con uno de los corpus pendientes.

3. Lo primero que hace `_trainModel` es actualizar en la MongoDB el status de ese corpus con valor PROCESSING. Acto seguido, se llama a `getModelDescriptorListFromDB` para obtener la *modelDescriptorList*, que es una lista que se actualiza cada vez que se entrena un modelo y se debe almacenar para ir aumentando la lista de modelos operativos. Cuando tiene esta información, lo siguiente que hace es generar un identificador aleatorio para el modelo que se creará e invoca al método `trainModel` del servicio `Python` con la información recabada y un parámetro que indica el número de iteraciones que se debe realizar en el entrenamiento, por defecto 100 iteraciones (a mayor número, más fiable es el modelo).

4. Si devuelve algún fallo, se aborta y se actualiza en la base de datos el corpus con status FAILED para ser revisada la causa; si retorna exitosamente, se actualiza en la colección pythonFiles de MongoDB el valor actualizado de la modelDescriptorList y se actualiza también el corpus con status READY, y será indicador de que el modelo ha sido entrenado exitosamente.

5. Al acabar, `_followCorpusTraining` se encarga de las actualizaciones del status del corpus mencionadas, y vuelve a invocarse a sí mismo para ejecutarse de nuevo con el siguiente corpus pendiente, hasta que no queden más por procesar. En ese momento, vuelve a `_checkUntrainedTrollnets` que se encarga de configurar un temporizador para volver a repetir estos mismos pasos más tarde, en caso de que se hayan encontrado nuevos corpus pendientes.

El `thread.controller.js` se encarga de la generación de conversaciones. `createConversation` es el método público para empezar una conversación de una lista de bots entorno a un hilo. De cara a simular una conversación real, se han adoptado ciertas licencias (que no eran requisitos ni son necesarios) sólo para simplificar determinados procesos y decisiones:

1. El tamaño de la conversación es el número de bots en la lista al cuadrado. Queda de cara a expandir este proyecto ofrecer la opción al usuario del producto de configurar el número de mensajes que desee.

2. Para calcular si un bot debe responder al hilo general o a algún otro bot, el método `_shouldReplyOrJustAnswer` compara un valor aleatorio entre 0 y 1 contra la raíz cuadrada del cociente entre el número de respuestas dadas y el tamaño total de la conversación. Si es inferior, responderá a otro bot. Al principio de la conversación será muy raro que un bot responda a otro; cuanto más avance la conversación, más probable es que los comentarios sean respuestas a otros bots.

Así, Para generar la conversación, se inicia el proceso recursivo `_followConversation` de forma que en cada nivel se generará una respuesta al hilo. Para obtener cada respuesta, se siguen los pasos aquí descritos:

1. Se escoge un bot de la lista al azar con `_getRandomDifferentMember`, que evita al último que respondió para no tener repeticiones inmediatas.

2. Se determina si el bot debe responder al hilo general o no con `_shouldReplyOrJustAnswer`. Si debe responder a otro miembro, con `_getRandomDifferentMember` se obtiene de entre la lista de bots que han hablado, un miembro diferente al bot que va a responder ahora, y con `_getLastMessageFrom` se obtiene el último mensaje de ese miembro y se establece como mensaje a responder. Si no se debe responder a ningún otro miembro, se establece el hilo original como mensaje a responder y el destinatario queda vacío.

3. Se invoca el método `answerThread` del controller `botController` con el identificador del bot que debe generar una respuesta y el mensaje al que responder (contenido + identificador del mensaje). Este método está explicado más arriba en la descripción de dicho controlador.

4. Una vez se obtiene la respuesta, se almacena en un objeto que contiene el emisor, el destinatario y el contenido del mensaje, y dicho objeto se introduce al final de un array que contiene todas las respuestas de esta conversación. Por último, `_saveLastTalker` guarda cuál ha sido el último bot en hablar, para que conste en la lista de bots que han hablado usada más arriba.

5. Si quedan más mensajes por generar, `_followConversation` vuelve a invocarse a sí mismo para seguir el proceso hasta el final, momento en el cual se vuelve al proceso invocador `createConversation` y éste se encarga de almacenar la conversación en la colección de conversations de la MongoDB, junto al identificador de la trollnet, la fecha de creación y el topic.

El `trollnet.helper.js` contiene métodos auxiliares relacionados con la gestión del tipo de dato `trollnet`:

1. El método `updateCreationStatus` permite actualizar directamente en la MongoDB el parámetro creationStatus y el parámetro error.

2. El método `updateTrainingStatus`, permite actualizar directamente en la MongoDB el parámetro trainingStatus y el parámetro error.

El `utils.helper.js` contiene varios métodos utilizados en diferentes sitios de la aplicación, por lo general simples cálculos o algoritmos útiles:

1. El método `generateId` se utiliza en casi todos los scripts que van a almacenar algo en la base de datos, para generar un nombre aleatorio.

2. El método `randomIntegerInInterval`, como su nombre indica, sirve para obtener un entero aleatorio dentro de un intervalo de enteros delimitado por los parámetros, un entero mínimo y un entero máximo.

3. El método `getIntervalPortion` sirve para obtener la proporción de un valor dentro de un intervalo.

4. El método `randomElementFromArray` sirve para obtener un elemento aleatorio de entre los elementos de un array.

5. El método `splitRandomlyByCoef` sirve para separar aleatoriamente los elementos de un array en dos arrays diferentes y según una proporción determinada, de forma que pueda hablar más en un array que en el otro, dependiendo de la proporción especificada por parámetro.

6. El método `_shuffleArray` es utilizado en el `splitRandomlyByCoef` para reordenar aleatoriamente el array antes de separarlo.

7. El método `stringToAscii` sirve para codificar todos los caracteres de un string a codificación ASCII.

8. El método `asciiToString` sirve para decodificar un texto en ASCII a su texto original.

Los modelos de datos manejados en el servidor se han diseñado como modelos para ser usados por MongoDB. Por eso, se utiliza los métodos `Schema` y `model` de la librería de npm `mongoose` para crearlos y registrarlos. Para los modelos Trollnet y TrollnetDraft, puesto que se comparten con el frontend, se sigue el mismo esquema con el fin de reforzar la mantenibilidad de código, y allanar el terreno al comunicar cliente y servidor. Encontramos otros modelos como:

1. `Bot` utilizado para almacenar información de bots creados. Tendrán un string identificador `id`, una fecha de creación `creationDate`, un string como nombre `botName`, un objeto de propiedades que le describen `properties`, el objeto descriptor `AIObject` que genera y utiliza Python en su creación y enseñanza respectivamente y una fecha `lastTrained` para indicar cuándo se entrenó por última vez.

2. `BotUser` utilizado para almacenar la asociación de un bot a una cuenta de Twitter. Tendrán un string identificador del bot `botname`, y la cuenta asociada en un string `twitteruser`.

3. `Corpus` contendrá información acerca de un archivo de corpus. Tiene un string identificador `id`, un string de su ruta de localización `fileName` y un booleano `isNewCorpus` indicando si es un corpus procesado o no.

4. `Conversation` servirá para crear y almacenar conversaciones, descritas por la fecha de creación `creationDate`, el identificador de la trollnet que generó la conversación `trollnetId`, el hilo original utilizado para generarla `topic` y el array de mensajes (con emisor, destinatario y contenido) generados en el campo `conversation`.

5. `PythonFiles` se utiliza para almacenar aquellos objetos que los scripts de python requieran frecuentemente. Por el momento, se almacena la model descriptor list en el campo `modelDescriptorList`, además de una fecha de creación en `creationDate` y un valor de versión `version` como campos auxiliares útiles para el desarrollo y control de versiones de datos.

5.2.5. Diagramas de clases de los Services

El servicio `mongo.db.js`, gracias a la librería de npm `mongoose`, permite establecer una conexión entre el cliente utilizado en la aplicación servidor y la base de datos de MongoDB. El servicio creado expone públicamente un método `_connect` para este fin, y una vez se le proporciona la URL de MongoDB y las opciones deseadas para la conexión (constantes extraídas del archivo de configuración `server.config.js`), avisará de si la conexión ha tenido éxito para poder operar contra la base de datos con garantía de funcionamiento y disponibilidad, así como de si se producen futuros errores en las operaciones contra la MongoDB.

El servicio `python.service.js` tiene la capacidad de ejecutar scripts de python y atender a su respuesta. Se debe especificar la ruta del sistema de archivos del servidor donde está el ejecutable de Python que se debe usar; así como la ruta del script que se quiere lanzar. En el caso de este proyecto, se ejecutará desde dos carpetas distintas, por razones de distribución del trabajo en el proyecto. Por un lado el script `scripts.py` situado en `C:/Proyectos/etv-tfg/ETV-models-and-bots/ETV`; por otro, los siguientes scripts `assign_accounts_to_bots.py`, `listener_twitter.py` y `script_publisher.py` situados en `C:/Proyectos/etv-tfg/bot_publisher`. Los parámetros de los métodos deben ir en formato texto; y además, en el caso de algunos parámetros de alguno de los métodos (cuando contienen objetos JSON pasados a texto), es requisito del script que se codifique la entrada a ASCII, y así mismo se decodifique cuando la ejecución de los scripts de Python retornen valores de las mismas características. Además, para todos ellos, es necesario filtrar la salida del script pues puede devolver un array con varios elementos de log de información, pero en todos los casos la respuesta que interesa es el último elemento del array y esa es la que habrá que procesar. Este servicio expone los siguientes métodos:

1. `createBot`: ejecuta el script `scripts.py` con la orden `create` y varios parámetros de un bot como la edad, nivel de educación, likes y dislikes para devolver un objeto que describe a un bot.

2. `trainModel`: ejecuta el script `scripts.py` con la orden `trainModel` y varios parámetros como el identificador de un modelo, el nombre del archivo, la modelDescriptorList explicada en la sección 5.2.4 y el número de iteraciones para refinar el modelo. Devuelve la modelDescriptorList actualizada con la información asociada al último modelo entrenado.

3. `teachBot`: ejecuta el script `scripts.py` con la orden `trainBot`, el objeto descriptor del bot y la modelDescriptorList.

4. `answerThread`: ejecuta el script `scripts.py` con la orden `getResponse`, el objeto descriptor del bot, el mensaje a responder, los parámetros de filtrado y devuelve el mensaje generado por el bot.

5. `linkAccountsToBots`: ejecuta el script `assign_accounts_to_bots.py` con una lista de identificadores de bots como único parámetro. Devuelve la lista de los bots a los que se les ha asignado una cuenta.

6. `listenToTwitterAccount`: ejecuta el script `listener_twitter.py` con un nombre de perfil de Twitter. El script sigue ejecutándose hasta que dicha cuenta publique un mensaje, en cuyo momento retornará con el identificador de ese mensaje y su contenido. Ambos datos serán lo que devuelva finalmente este método al proceso invocador.

7. `publishMessage`: ejecuta el script `script_publisher.py` con el identificador del mensaje al que responder, el contenido de la respuesta que se va a publicar y el identificador del bot que la va a publicar. Una vez se ha publicado, devuelve el identificador del mensaje que se publicó.

El servicio `filesystem.service.js`, gracias a la librería `chokidar` y al módulo de NodeJS `fs`, es capaz de detectar qué archivos hay en la ruta que se desee y atender a cambios en dichos archivos o carpetas. Acepta también expresiones regulares o strings que determinen rutas o nombres de archivo para los cuales se desee pasar por alto la detección. Para su usabilidad desde módulos que lo quieran invocar, expone los métodos `scanFolder` para detectar los elementos existentes en una determinada ruta y `onNewFileAdded` para detectar genéricamente los cambios que ocurran en una ruta.

5.3. Arquitectura de los scripts de Python para la utilización de Twitter

La interacción del servidor central con la red social Twitter se ha centralizado en una serie de scripts de Python agrupados por funcionalidades: publicar respuestas de bots en Twitter, recoger publicaciones de los clientes, asginar usuarios disponibles a los bots y autorizar a la APP de Twitter a publicar en nombre de los usuarios.

Para la integración de Twitter con Python, se utilizará la API oficial de Twiter junto con la biblioteca de Python *tweepy*, que facilita el la utilización de la misma. Así mismo, se ha utilizado la base de datos central para almacenar la información de las cuentas de Twitter y de las asignaciones de usuarios a bots.

5.3.1. Consideraciones previas para la integración de Twitter en Python

Para utilizar la API oficial de Twitter es necesario cumplir una serie de requisitos[15], el más importante es disponer de una cuenta de desarrollador de Twitter. Una cuenta de

dessarrollador se obtiene completando un complejo formulario sobre el uso que se hará de la misma, tras la aprobación de Twitter. Una vez se dispone de una cuenta de desarrollador, se debe crear una APP de Twitter. Al crear esta APP se generarán las claves de autorización que serán necesarias para interactuar desde Python con la API de Twitter. Sin embargo, aún no se podrá publicar en nombre de los usuarios que se quiera.

Para publicar en Twitter utilizando la APP creada deben en todo momento cumplirse las políticas de Twitter para desarrolladores[13] y además cada usuario debe autorizar a la aplicación para que publique en su nombre. Esta autorización solo debe realizarse una vez y nunca caduca, por lo que una vez hecho se dispondrá de la cuenta para publicar y responder en nombre de los bots. Para la integración se ha desarrollado un script específico que automáticamente almacena los *tokens* generados en la base de datos principal. Puede verse el diagrama del flujo en la figura B.10 del Apéndice B.

5.3.2. Estructura básica y patrón de datos

El core de la interacción está dividido en dos scripts de Python principales: el script dedicado a realizar publicaciones en Twitter y el script dedicado a permanecer en espera para recoger las publicaciones que realicen los clientes. Para la interacción con Twitter, el servidor central llama a los distintos scripts disponibles, que centalizan la interacción. Estos scripts interaccionan con Twitter utilizando su API oficial. Para el almacenamiento de las claves y tokens de autorización, de los usuarios disponibles y otros elementos de interés se utiliza la base de datos central.

Puede verse un diagrama simplificado en la figura B.11 del Apéndice B.

5.3.3. Arquitectura de la base de datos de la interacción con Twitter

La base de datos de MongoDB que utilizan los scripts de Python para la interacción con Twitter está formada por dos *Collections*: *bot users* y *twitter accounts*. Puede verse la estructura básica en la figura B.12 del Apéndice B.

La *Collection bot users* almacena todos los bots y la cuenta de usuario de Twitter que se le ha asignado a cada uno. La *Collection twitter accounts* almacena todas las cuentas de Twitter disponibles para utilizar por el servidor central junto con los token de autorización para que la APP publique por ellos.

Capítulo 6

Casos de Estudio

En el presente capítulo se propone un entorno ficticio en el que aplicar nuestra solución para que el lector lo utilice de escenario en el que figurar las explicaciones de los siguientes casos de uso que prosiguen. Para cada uno de los casos de uso, que vendrán acompañados de diagramas explicativos, se dividirá su proceso en tantas fases como separaciones haya entre el frontend web, el servidor web y los scripts de Python, en un intento de visualizar la intervención de cada una de estas capas en cada caso de uso.

6.1. Aplicando la solución en un entorno propuesto

Para atraer la atención en sus redes sociales, el recién contratado Social Media Manager de un incipiente diario online de política busca una forma de ampliar el alcance de sus noticias en las redes, y busca una forma de aumentar la interacción de los usuarios en sus perfiles de Twitter, Instagram, Facebook, etc. En la era moderna, toma mucha importancia la cantidad y contenido llamativo (no tanto la calidad o veracidad del mismo) para que los motores de búsqueda y de sugerencias de las redes sociales destaquen ciertos perfiles por encima de otros.

Por ello, este empleado que recién ha conocido las capacidades del producto de El Trol Versátil, desea probar qué tal funciona en su caso, esto es: necesita usuarios (aunque sean bots) que se adhieran rápidamente en sus publicaciones, y a ser posible que generen debate y polémica por los comentarios que añadan. Puede bastarle con una veintena de bots de diferentes edades y perfiles sociales, que puedan hablar de temas relacionados con la política (con más o menos inclinaciones políticas en diferentes ámbitos) y que den sus opiniones (más o menos controvertidas) en cada publicación que él haga.

Nuestra solución le permite conseguir su propósito: necesita acceder a nuestra aplicación web, crear una red de trolls con esas características, y una vez nuestro servicio haya creado y entrenado esa red, tendrá a su alcance que los bots generen conversaciones sobre cada una de las publicaciones del perfil en Twitter de su diario, o sobre las publicaciones de cualquier otro perfil que introduzca.

A continuación, se profundizará en cada uno de los casos de uso que intervienen en este

escenario.

6.2. Caso de uso: Crear y entrenar una nueva trollnet

El usuario previamente ha accedido a la aplicación web y ha seleccionado la pantalla de 'New trollnet'. A continuación se describe el proceso por el que el usuario elige las características de la trollnet que desea crear y procede aceptar la creación, que lleva a cabo el servidor para crear tanto la trollnet como cada uno de los bots que van asociados a ella, así como lo que ejecutan los scripts de Python que crean cada uno de los bots. Cuando ese proceso finaliza, a través de otro script de Python se llevará a cabo una vinculación uno a uno de cada bot de la trollnet con una de las cuentas disponibles (previamente creadas por los administradores de este producto) del servicio que vamos a utilizar para publicar los mensajes, Twitter; tras esto, empezará a entrenar la trollnet, en un proceso similar a la creación, entrenando uno por uno los bots apoyándose en los scripts de Python para dicho entrenamiento. Una vez esto finalice, se guardará la trollnet y cada uno de los bots actualizados en la MongoDB. En paralelo, el frontend irá solicitando el estado del progreso al backend y será informado del porcentaje de compleción del mismo, y se le mostrará al usuario.

La figura B.13 del Apéndice B servirá de apoyo para la explicación de este caso de uso.

6.2.1. Fase 1: Frontend Web - procesar el input del usuario

1. Una vez el usuario haya accedido a la pestaña 'New trollnet', observará en el lado derecho de la aplicación una serie de componentes parametrizables para moldear las características de la trollnet que desea crear. Cada vez que modifique uno de estos parámetros, el componente que lo rige modificará esa característica en la plantilla para la trollnet. Así, tiene a su alcance la modificación de las siguientes propiedades:

    a) El nombre de la trollnet, modificará la propiedad *customName*. Este parámetro es, de momento, sólo estético, pero le permitirá identificar la trollnet de entre su lista de trollnets.

    b) El rango de edades, a través del componente `age-selector.component.ts`, modificará la propiedad *ageInterval*. Este parámetro es, de momento, sólo estético.

    c) Los géneros posibles, a través del componente `gender-selector.component.ts`, modificará la propiedad *genders*. Este parámetro es, de momento, sólo estético.

    d) Las etnias posibles, a través del componente `ethnicity-selector.component.ts`, modificará la propiedad *ethnicity*. Este parámetro es, de momento, sólo estético.

    e) El rango de educación o formalidad del habla, que se da a través del componente `cultural-level.component.ts`, modificará la propiedad *culturalLevel*. Este parámetro influye en el nivel de formalidad de la respuesta de los bots.

f) El rango del nivel de positividad del bot en el habla, a través del componente `mood-selector.component.ts`, modificará la propiedad *moodLevel*. Este parámetro influye en la proporción relativa entre *likes* y *dislikes* (explicados más adelante), de forma que cuanto más bajo sea el nivel de positividad, más tenderá a quejarse de aquello que no le gusta, y al contrario, cuaanto más alto, más tenderá a hablar de aquello que le gusta.

g) El tamaño de la trollnet en número de bots, que se puede fijar a través del componente `net-size.component.ts`, modificará la propiedad *netsize*. Pueden ser valores entre 5 y 50, a inervalos de 5.

h) El intervalo del nivel de interacción, definido por medio del componente llamado `interaction-level.component.ts`, modificará la propiedad `interactionLevel`. Esto influirá en el tamaño de las respuestas del bot.

i) La inclusión de una serie de palabras clave, utilizando el componente llamado `keywords.component.ts`, modificará la propiedad *keywords*. Los bots aprenderán diferentes combinaciones de estos temas, y dependiendo del *moodLevel* se distribuirán como *likes* o *likes* del bot.

j) La inclusión de una serie de gustos o *likes*, a través del componente `keywords.component.ts`, modificará la propiedad *likes*. Los bots expresarán más afección por estos temas que por otros, teniendo cada bot una combinación diferente de estos temas.

k) De formal similar, la inclusión de una serie de odios o *dislikes*, a través del componente `keywords.component.ts`, modificará la propiedad *dislikes*. Los bots expresarán más desafección por estos temas que por otros, teniendo cada bot una combinación diferente de estos temas.

2. Al modificar los parámetros a su alcance, cada componente modificará su propiedad en la plantilla almacenada en la lógica de esta pantalla, `create.page.ts`, de tipo `TrollnetDraftModel`. Cuando esté listo, seleccionará el botón 'CREATE TROLLNET' en la parte inferior de la pantalla, de forma que el flujo alcanzará a la `trollnet.store.ts` a través de `createNewTrollnet` con la plantilla como parámetro.

3. A su vez, la store redirigirá el flujo a `trollnetService` para acceder a través de él a la API remota del backend con una petición POST en el endpoint `/api/trollnet`, con la plantilla de la trollnet como payload.

Los últimos pasos de la fase están ligados entre sí con promesas (Objetos **Promise** del ECMASCRIPT 6). Cuando se reciba la respuesta del servidor en la fase 2 del caso de uso, las promesas irán resolviéndose sucesivamente, y se observará en el frontend una trollnet creada. En la fase 9 de este caso de uso se explica lo pertinente a observar el progreso de creación y entrenamiento de la trollnet.

6.2.2. Fase 2: Servidor Web - iniciar la creación de la trollnet

1. En el servidor, el router `router.js` procesará la petición en el endpoint `/api/trollnet` y la redirigirá hacia el `trollnet.router.js`. Al detectar que es una petición tipo POST, la redirige al método `addTrollnet` del `trollnet.controller.js`.

2. Éste invocará al `createBotArray` del `bot.controller.js` para iniciar la creación de los bots asociados a la trollnet, almacenará en la MongoDB un nuevo objeto Trollnet con la información de esta trollnet y resolverá la petición del frontend como éxito, confirmándole que se inicia el proceso de creación de la trollnet.

3. El `bot.controller.js` iniciará una operación recursiva a través de `_followBotCreation`, que se llamará a sí misma tantas veces como bots se deba crear. Ese método se encarga de, en primer lugar, calcular mediante `_getRandomParams` una configuración de parámetros aleatoria dentro de los intervalos específicos establecidos por los parámetros de la trollnet, elegidos por el usuario. Esa configuración será la asignada al bot que se esté creando en ese punto. En segundo lugar, llamará a `_createBot`.

4. `_createBot` usará el servicio `PythonService` con los parámetros adecuados para la creación del bot, el cual se encargará de realizar la llamada al script de Python con los parámetros en el formato adecuado, y de recoger y procesar su respuesta en la fase 4 de este caso de uso.

6.2.3. Fase 3: Ejecución Python - creación de un bot

Para esta fase, véase figura B.14 del Apéndice B.

1. Cuando Scripts.py recibe un comando a ejecutar, primero analiza los parametros en busca de opciones generales especificadas (véase sección 4.9)), si se encuentran se levantan los flags pertinentes. Después se identifica el comando ”create” y se hace una validación de los parámetros introducidos.

2. Se solicita la creación de una instancia de Bot con parámetros introducidos y se transforma a JSON.

3. Se devuelve el resultado por la salida específicada.

6.2.4. Fase 4: Servidor Web - control y fin de la creación de la trollnet

1. Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca del objeto bot creado y lo devolverá a `bot.controller.js`, que se encargará de almacenar la respuesta dentro del campo AIObject de este nuevo bot creado en la MongoDB.

2. En este momento, se calcula la proporción de bots creados sobre los totales por crear y se actualiza la trollnet en la MongoDB con el campo creationStatus a dicho porcentaje (sobre 100).

3. Si faltan más bots por ser creados, `_followBotCreation` será invocado de nuevo para volver a realizar los pasos previos. Si no, el flujo será devuelto al invocador en `trollnet.controller.js` que, si no registra ningún fallo del proceso, actualizará la trollnet en la MongoDB con la lista de identificadores de los bots.

4. Se requerirá un último paso antes de dar por finalizada la creación: vincular cuentas de Twitter a cada uno de los bots. Para ello, se invoca al `linkAccountsToBots` del `PythonService`, que invocará al script de Python con la lista de bots que deben recibir la asignación de una cuenta.

6.2.5. Fase 5: Ejecución Python - vincular cuentas a los bots

1. El script de Python recibirá una lista de nombres de bots. Esta lista debe contener todos los nombres de bots para los que se quiere que se les asigne un usuario disponible de Twitter para publicar los tuits que se generen para él.

2. Una vez recibida la lista se ejecutará `assign_accounts`, que primero cargará de la base de datos ETV la lista de usuarios de Twitter disponibles y después asignará a cada bot un usuario de Twitter mientras queden.

3. Finalmente se almacenará en la base de datos ETV las asignaciones que se han realizado y devolverá al script un *json* con la lista de los bots a los que se les ha asignado un usuario de Twitter y otra lista con los que no se les haya podido asignar. El script remitirá esta información y finalizará. Puede verse el flujo de vinculación en la figura B.15 del apéndice B.

6.2.6. Fase 6: Servidor Web - iniciar proceso de entrenamiento de la trollnet

1. Cuando finalice el script de Python, `PythonService` notificará la compleción al controlador `trollnet.controller.js` que, si ha funcionado bien, guardará en la MongoDB la trollnet con el campo creationStatus a 100. Además, apuntará la trollnet a la lista volátil de trollnets sin entrenar con `_addUntrainedTrollnet`.

2. El demonio del `trollnet.controller.js` que realiza chequeos periódicamente con `_checkUntrainedTrollnets` detectará una nueva trollnet por entrenar, e iniciará la operación recursiva `_followTrollnetTraining`. Según el mismo principio que sigue `_followBotCreation`, que se llamará a sí misma tantas veces como trollnets se deba entrenar. A cada nivel de la misma, llamará a `_teachTrollnet`, que actualizará primero el trainingStatus de la trollnet en la MongoDB a 1; después llamará a `teachBotArray` del `botController` con la lista de bots de la trollnet.

3. `teachBotArray` se encarga de iniciar el entrenamiento de los bots asociados a la trollnet, pero para ello primero debe extraer un dato importante para los scripts de Python, la model descriptor list, y lo hace llamando al método `getModelDescriptorListFromDB` del `ModelController`. Cuando lo tiene, invoca la operación recursiva `_followBotTeaching`, que se llamará a sí misma tantas veces como bots se deba entrenar.

4. En cada nivel, invocará a `_teachBot` que extrae la información del bot de la MongoDB, y junto a la model descriptor list, los utiliza como parámetros en la llamada al método `teachBot` del servicio `PythonService`, el cual se encargará de realizar la llamada al script de Python con los parámetros en el formato adecuado, y de recoger y procesar su respuesta en la fase 8 de este caso de uso.

6.2.7. Fase 7: Ejecución Python - entrenamiento de un bot

Para esta fase, véase figura B.16 del Apéndice B.

1. Cuando Scripts.py recibe un comando a ejecutar, primero analiza los parametros en busca de opciones generales especificadas 4.9, si se encuentran se levantan los flags pertinentes. Después se identifica el comando "trainBotz se hace una validación de los parámetros introducidos.

2. Se carga la instancia del bot.

3. Se carga la instancia de la MDL.

4. Se llama a learn en el bot cargado con la MDL cargada.

5. El bot consulta en la BD las relaciones entre sus gustos y las keywords de cada modelo en la MDL.

6. Asocia los modelos que encajan.

7. Se devuelve instancia del bot actualizada.

8. scripts.py devuelve al bot actualizado.

6.2.8. Fase 8: Servidor Web - control y fin del entrenamiento de la trollnet

1. Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca del objeto del bot entrenado y lo devolverá a `bot.controller.js`, que se encargará de actualizar el campo AIObject del bot con la respuesta en la MongoDB.

2. En este momento, se calcula la proporción de bots entrenados sobre los totales por entrenar y se actualiza la trollnet en la MongoDB con el campo trainingStatus a dicho porcentaje (sobre 100).

3. Si faltan más bots por ser entrenados, `_followBotTeaching` será invocado de nuevo para volver a realizar los pasos previos. Si no, el flujo será devuelto al invocador en `trollnet.controller.js`, que se encargará de actualizar la trollnet en la base de datos modificando su trainingStatus a 100. Si algo hubiera fallado en el entrenamiento de alguno de los bots, se actualizaría con trainingStatus -1, y se almacenaría en el mismo objeto logs del error impreso en el campo error.

6.2.9. Fase 9: Frontend Web - chequeo periódico del progreso

`trollnet.store.ts` se encarga de realizar periódicamente una petición al backend para actualizar el estado de las trollnets existentes que no han finalizado su entrenamiento, apoyándose en el `trollnetService`, el cual hace la petición GET a `/api/trollnet/status`. Con cada una de las llamadas, se recibe para cada trollnet 'sin terminar' el valor de su progreso de creación y de entrenamiento, y si el usuario observa la vista 'My trollnets', observará 2 barras de progreso que indican dicho porcentaje de avance; cuando esté completamente creada y entrenada, se habilitará un botón selector para poder generar conversaciones con ella, que veremos en el caso de uso de la sección 6.4.

6.3. Caso de uso: Entrenar un modelo

Este caso de uso aplica a los usuarios administradores del servidor, que se encargarán periódicamente de que el volumen de modelos existentes aumente para ofrecer un mejor servicio a los clientes. Para entrenar un modelo, los administradores deberán encargarse manualmente de conseguir archivos de texto que conoceremos como *corpus*. Una vez los tengan, si los introducen en la carpeta designada a su almacenaje, el código del servidor detectará su presencia y procederá a marcarlos como archivos a procesar para generar un modelo a partir de cada uno de ellos, los cuáles son controlados por un proceso que periódicamente comprueba estos archivos y lanza la generación de cada uno de ellos. En las siguientes fases describiremos en detalle estos pasos.

La figura B.17 del Apéndice B servirá de apoyo para la explicación de este caso de uso.

6.3.1. Fase 1: Servidor Web - detección del corpus y comienzo del proceso

1. El `corpus.controller.js` se encarga con su demonio de tener un flujo que a través del `fileSystemService` detecta cuando se añade un archivo a la carpeta donde se guarda esta corpus: `C:\Proyectos\etv-tfg\ETV-models-and-bots\ETV\rawCorpus`. Al detectar un nuevo archivo, se procede a almacenar su ruta en la colección de corpus de la MongoDB con un flag que indica que es nuevo (no ha sido procesado todavía).

2. Paralelamente, el `model.controller.js` comprueba periódicamente con `_checkPendingCorpuses` a través del método `getAllNewCorpuses` del `corpus.controller.js` si hay nuevos corpus por procesar y, si los hay, se ejecuta la operación recursiva
`_followCorpusTraining` que se llamará a sí misma tantas veces como corpus se deban procesar para obtener modelos a partir de ellos.

3. En cada nivel, invocará a `_trainModel` con la ruta de un corpus sin procesar. Primero extrae la model descriptor list, llamando a `getModelDescriptorListFromDB` del `ModelController`. Tras esto, llama al método `trainModel` del servicio `PythonService` con la model descriptor list, la ruta del corpus y un valor del número de iteraciones para refinar el modelo. El servicio de Python se encargará de realizar la llamada al script de Python con los parámetros en el formato adecuado, y de recoger y procesar su respuesta en la fase 3 de este caso de uso.

6.3.2. Fase 2: Ejecución Python - entrenamiento de un modelo a partir de un corpus

Para esta fase, véase figura B.18 del Apéndice B.

1. Cuando Scripts.py recibe un comando a ejecutar, primero analiza los parametros en busca de opciones generales especificadas 4.9, si se encuentran se levantan los flags pertinentes. Después se identifica el comando "trainModelz se hace una validación de los parámetros introducidos.
2. Se solicita a models.py entrenar un nuevo modelo.
3. models.py lanza un nuevo entrenamiento con el corpus dado en GPT2.
4. models.py lanza un conteo para extraer keywords del corpus a wordCounter.
5. models.py construye la instancia del modelo con los datos conseguidos y se devuelve esta instancia.
6. scripts.py añade el nuevo modelo a la MDL.
7. scripts.py lanza una lectura de nuevas relaciones para la BD.
8. factsGeneration.py lee el corpus y analiza en busca de nuevas relaciones.
9. factsGeneration.py añade las nuevas relaciones a la BD.
10. sripts.py devuelve la MDL actualizada.

6.3.3. Fase 3: Servidor Web - actualización de la MDL y control del proceso

1. Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca de la model descriptor list actualizada y la devolverá a `model.controller.js` que, si el entrenamiento funcionó bien, se encargará de actualizar en MongoDB la model descriptor list en la base de datos con la misma actualizada que viene como respuesta del script además de marcar el corpus en la MongoDB como ya procesado a través del método `markAsUsed` del servicio `corpus.controller.js` con la ruta del corpus.

2. Si faltan más corpus por procesar, `_followCorpusTraining` será invocado de nuevo para volver a realizar los pasos previos. Si no, la función recursiva de entrenamientos de modelos llegará a su fin y dichos modelos estarán disponibles para futuros entrenamientos de bots.

6.4. Caso de uso: Generar una conversación

El usuario previamente ha creado una trollnet y ha esperado a que ésta esté completamente creada y entrenada. En este momento, activará una trollnet desde su vista de 'My trollnets', introducirá el nombre de un perfil de Twitter y, a través de un script de Python, el flujo quedará a la espera de que dicho perfil de Twitter haga una nueva publicación. Cuando eso ocurra, la trollnet empezará a generar una conversación sobre dicha publicación, ejecutando un script de Python para generar cada mensaje de la misma. Además, otro script de Python permitirá publicar cada mensaje en el hilo de Twitter, respondiendo a la publicación principal o a cada usuario según sea el caso. En este caso de uso describiremos en detalle el proceso que ocurre para que una trollnet genere una conversación sobre dicha publicación.

La figura B.19 del Apéndice B servirá de apoyo para la explicación de este caso de uso.

6.4.1. Fase 1: Frontend Web - procesar el input del usuario

1. Una vez el usuario haya accedido a la pestaña 'My trollnets', observará que la trollnet que creó tiene un interruptor de dos posiciones. Lo encontrará en posición apagado, y si lo enciende, se le pedirá que introduzca el perfil de Twitter objetivo, sobre el que la trollnet generará una conversación cuando dicho perfil realice su siguiente publicación. Una vez el usuario introduzca el nombre de perfil deseado, y presione 'Troll that profile', el componente `trollnet-preview.component.ts` que recoge esta acción contactará con la `trollnet.store.ts` a través de `activateTrollnet` con el perfil especificado.

2. La store redirigirá el flujo a `trollnetService` para acceder a través de él a la API remota del backend con una petición PUT en el endpoint `/api/trollnet/activate`, con el id de la trollnet y el perfil objetivo.

6.4.2. Fase 2: Servidor Web - preparación inicial y petición de escucha

1. En el servidor, el router `router.js` procesará la petición en el endpoint `/api/trollnet` y la redirigirá hacia el `trollnet.router.js`. Al detectar que el endpoint final es `/api/trollnet/activate` y que es una petición de tipo PUT, la redirige al método `activateTrollnet` del `trollnet.controller.js`.

2. Éste actualizará en la MongoDB el estado de la trollnet como activado, se lo notificará al frontend y lanzará su método `_prepareTrollnetConversation` con el id de la trollnet y el nombre de la cuenta de Twitter introducido por el usuario, que vendrá en el campo targetAccount. A su vez, se invocará `listenToTwitterAccount` del `PythonService` con la cuenta de Twitter para observar cuándo esa cuenta publicará un mensaje. Este servicio se encargará de realizar la llamada al script de Python con los parámetros en el formato adecuado, y de recoger y procesar su respuesta en la fase 4 de este caso de uso.

6.4.3. Fase 3: Ejecución Python - escuchar la próxima publicación de la cuenta objetivo

La ejecución creará un *Stream* que permanecerá a la espera de la publicación de un tuit por parte del usuario, para remitirlo en cuanto este sea publicado(puede verse el flujo en la figura B.20 del apéndice B).

1. Lo primero que se hará es construir un objeto *MyStreamListener* de la biblioteca *tweepy* de Python, que nos permitirá comunicarnos con la API de Twitter y abrir un *Stream* de escucha a la cuenta del usuario. Para construir este objeto se utilizarán las claves de autenticación de la aplicación de Twitter ETV y los token de autorización del usuario de Twitter que hemos creado expresamente para ejecutar este script.

2. A continuación el script de Python utilizará el nombre de usuario recibido y utilizará la API de Twitter para obtener su ID de Twitter, código único que identifica a cada cuenta dentro de la red social.

3. Con este ID y el *Stream* creado se ejecutará `stream.filter`, que permanecerá escucuchando a la cuenta de Twitter del usuario hasta que publique algún tuit.

4. Una vez el usuario realice la publicación, el *Stream* recibirá toda la información del tuit. Con esta información el script remitirá tanto el ID como el texto completo del tuit. El ID del tuit lo identifica unívocamente entre todos los tuits de Twitter, y facilitará localizarlo en el futuro para interactuar con él.

6.4.4. Fase 4: Servidor Web - inicio del proceso de conversación

1. Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca del identificador del mensaje publicado en la cuenta observada y su contenido y los devolverá a `trollnet.controller.js`. En este momento, se extraerá de la MongoDB la trollnet que participará y se llamará a `createConversation` del `thread.controller.js` con el identificador de la trollnet, su lista de bots, el mensaje publicado en la cuenta observada y su identificador.

2. El `createConversation` iniciará una operación recursiva a través de `_followConversation`, que se llamará a sí misma tantas veces como sea necesario generar mensajes para la conversación. Este método está explicado en la sección 5.2.4. Se habrá seleccionado un bot para que, la primera vez, comente el mensaje principal.

3. Se pedirá un mensaje al bot a través de `answerThread` del `botController`. Éste, en primer lugar, extrae la info del bot en cuestión de la MongoDB y calcula los parámetros de filtrado dependiendo de su nivel de interacción (esto es, un valor de respuestas posibles que aumentará el grado de ajuste de su respuesta al contexto, y otro valor para limitar el tamaño de la respuesta). En segundo lugar, usa el `answerThread` del servicio `PythonService` con la información del bot, los parámetros de filtrado, y el mensaje a responder; este servicio se encargará de realizar la llamada al script de Python con los parámetros en el formato adecuado, y de recoger y procesar su respuesta en la fase 6 de este caso de uso.

6.4.5. Fase 5: Ejecución Python - generación de respuesta

Para esta fase, véase figura B.21 del Apéndice B.

1. Cuando Scripts.py recibe un comando a ejecutar, primero analiza los parametros en busca de opciones generales especificadas 4.9, si se encuentran se levantan los flags pertinentes. Después se identifica el comando "generateResponsez se hace una validación de los parámetros introducidos.

2. Se carga la instancia del bot.

3. Se llama a generateResponse del bot.

4. El bot selecciona el modelo con el contexto introducido.

5. Solicita inferencia de parametros al Inferencer.

6.4.6. Fase 6: Servidor Web - recoger respuesta y lanzar flujo de publicación

Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca de la respuesta real y la devolverá a `bot.controller.js` que, si la respuesta se generó bien,

se encargará de llamar al `publishMessage` del servicio `PythonService` con el identificador del mensaje a responder, el mensaje que publicar y el bot que lo publicará.

6.4.7. Fase 7: Ejecución Python - publicación de respuesta en Twitter

En la ejecución completa se publicará una respuesta en la red social Twitter(puede verse el diagrama del flujo completo en la figura B.22 del apéndice B).

1. El script `script_publisher.py` crea la clase de Python `Publisher` y ejecuta su función `post_response` con los parámetros recibidos: el ID del tuit al que responder, el texto a publicar y el nombre del bot que publicará el tuit.

2. La función `post_response` cargará de la base de datos el usuario de Twitter que le corresponde al nombre de bot recibido y a continuación cargará también sus tokens de autorización de la APP de Twitter. Con estos tokens creará el objeto `tweepy.API` ya autorizado para que la APP publique en Twitter en nombre del usuario.

3. A continuación se ejecutará una llamada a la API de Twitter con la función `update_status`, que publicará una respuesta en Twitter al tuit que se corresponde con el ID recibido, utilizando el usuario cargado de la base de datos y con el texto que se ha recibido.

4. Una vez se publique el tuit en Twitter se devolverá el ID de este nuevo tuit.

6.4.8. Fase 8: Servidor Web - control y fin del proceso

1. Cuando finalice el script de Python, `PythonService` recogerá su output, filtrará en busca del identificador del mensaje publicado, y lo retornará a `bot.controller.js` que devolverá tanto este identificador como su contenido en texto al proceso invocador en `thread.controller.js` para proseguir, el cual registrará esta intervención del bot si ha sido exitosa.

2. Si faltan más mensajes por generar, `_followConversation` será invocado de nuevo para volver a realizar las fases previas 5, 6 y 7. Si no, la función recursiva de generación de mensajes llegará a su fin y la conversación estará almacenada en MongoDB.

Capítulo 7

Conclusiones

En este capítulo se tratan aquellas dificultades encontradas durante el desarrollo de este TFG y se comentan brevemente algunos aspectos que podrían mejorarse en el futuro.

7.1. Inteligencia de un bot

Cuando se habla de un bot se tratan muchas partes distintas del sistema que, trabajando de manera conjunta, se abstraen en forma de bot. Cada una de ellas está bien definida y tiene su problematica asociada, aunque en conclusión es la idea de estas piezas trabajando de manera conjunta la que hace que el sistema sea como es. Esta organización y distribución de roles propone un producto con un potencial muy grande, ya que un hipotetico sistema en el que estas piezas estuvieran altamente refinadas, lograría crear personalidades, algo mucho más allá del alcance de los productos en la competencia. La abstracción y separación de estos elementos facilita también el desarrollo y refactorización si así fuera necesario, logrando focalizar en una tarea muy clara cada una de estas piezas. Este ha sido uno de los problemas que se encontró al plantear el sistema, bajo la premisa de crear un bot con un comportamiento individual y humano, separar este objetivo en distintos engranajes más simples (y programables).

A pesar de que este objetivo se ha conseguido llevar a cabo, las distintas piezas que lo forman plantean otros retos, que aunque mejor definidos, carecen de simpleza. A continuación se plantean los diversos problemas y posibles soluciones que se encuentran en cada módulo.

7.1.1. Generación de texto

Generador

La implementación de este rol se ha llevado a cabo con GPT-2, a pesar de que GPT-3 existe desde que el desarrollo comenzó. Esto se debe a que GPT-3 permite el finetuning bajo pago, dependiendo de sus servidores. La libreria de GPT-2 utilizada permite que se haga el finetuning alla donde se quiera alojar. El finetuning es un pilar fundamental del

sistema para alcanzar los bots con personalidad, y esta barrera económica es la que hace que se descarte GPT-3. Con mayor presupuesto disponible, se pasaría a utilizar GPT-3. Esto no solo incrementaria de forma considerable la calidad de las respuestas, sino que además descargaría requerimientos por parte del equipo, ya que el finetuning se realiza en los servidores de OpenAI.

Por otro lado, el tiempo que se tarda en entrenar un modelo es un aspecto a mejorar con urgencia, ya que para modelos relativamente pequeños, se han encontrado tiempos de espera de 4 a 8 horas. Con corpus más grandes este tiempo crece de manera pronunciada. Una de las opciones, es preparar una máquina dedicada a tiempo completo a este trabajo, y que además los entrenamientos corran sobre una GPU. Esto último requeriría un ajuste en las librerías a instalar y el código, pero lo realmente costoso es poseer y mantener una máquina con estas características.

Otro problema hayado es la obtención de corpus. Actualmente los corpus se obtienen manualmente con un script que recopila tweets de twitter a partir de una o varias palabras clave. El propio script puede mejorar un poco, ya que actualmente hay que ejecutarlo varias veces por corpus si se quieren incluir varias palabras clave. Aunque, aun así, está lejos de ser una solución perfecta, pues el script hace llamadas a la API de Twitter, las cuales tienen un límite cada 15 minutos, por lo que el script a menudo se detiene y tarda bastante en obtener los corpus. Por tanto, o bien se podría buscar una alternativa para la obtención de corpus que no involucre recopilar mensajes de Twitter, o bien adaptar mejor el script en torno a la limitación de los 15 minutos y aumentar el nivel de automatización del mismo, que por ahora no llegó a ser muy necesario.

Pipeline de filtrado

El pipeline de filtrado es una herramienta con alto potencial de desarrollo y fácil ampliación. La extensión con otras características podría permitir un filtrado por personalidad más fino, en el que se pudieran tener en cuenta otros factores que definen una personalidad, como la etnia, la edad, etc. También se podrían añadir otras funciones modificadoras que estilizasen en mayor medida los textos resultantes, ajustandose así más a distintos perfiles de personalidad. Otro factor a explotar en este elemento es el orden de las funciones de filtrado, el orden en el que se ejecuten podría cambiar el resultado, ya que si se ejecutan funciones modificadoras las siguientes funciones evaluadoras leerán esos cambios. Esto con un volumen de funciones alto podría tener un amplio impacto, pudiendo llegar a merecer la pena soportar una interfaz que permita manjear este orden, ya sea de manera automática o para el usuario.

7.1.2. Base de datos de relaciones

Aunque en la implementación se ha optado por utilizar Prolog, bajo unas reglas relativamente sencillas, se podría decidir proseguir el desarrollo en otra dirección, o en esta misma ampliando la lógica de estas reglas. El objetivo de esta base de datos es condensar en un único parametro (un número de -1 a 1) como están relacionados dos conceptos en el mundo

que conocemos, por tanto cualquier servicio que cumpliera esta condición sería suficiente. Un posible camino, es utilizar otra red neuronal, que reciba como entrada estos dos conceptos y devuelva el número acorde. Esto conllevaría su dificultad asociada, ya que no solo hay que conseguir esta tecnología, si no que además hay que alimentarla con un catálogo de corpus acomodado a este requerimiento, y la automatización de la propia creación del corpus sería no trivial.

7.1.3. Inferencer

Esta herramienta, que se apoya fuertemente en la base de datos de relaciones y la cual configura como se aplica el pipeline de filtrado, tiene un ambito de desarrollo tan grande como el propio pipeline de filtrado (o incluso más). A su vez cada parámetro tiene su propio cálculo independiente, con tanto desarrollo y complejidad como se le quiera dar. Las dificultades encontradas se asocian a cada uno de los parámetros de filtrado. Los algoritmos utilizados podrían mejorarse ampliamente, aunque el buen funcionamiento de esta pieza depende en gran medida de la base de datos de relaciones. Una posible mejora (y aunque se comente en este apartado podría extrapolarse a otras partes) es hacer que el bot tenga en cuenta no solo el contexto inmediato, si no crear un contexto más complejo compuesto por los tweets que ese mismo bot haya escrito ese día (lo que podría ir definiendo de qué humor está), todos los tweets del hilo en el que se responda e incluso datos del usuario al que se responde como la descripción del perfil. Con todos estos datos se podrían refinar en gran medida los parámetros que se calculan, aunque esto tiene una problemática asociada, y es el crecimiento en ámbito y complejidad de los algoritmos, que cada vez deban tener mas factores en cuenta, haciendo que acaben siendo insostenibles de cara a la depuración y refinación. Es de vital importancia que cada una de los parámetros (al igual que su cálculo) esté bien definido y acotado.

7.1.4. Análisis de sentimiento

El desarrollo de este módulo se ha basado en las demandas requeridas por el resto del sistema, haciendo de *wrapper* de la librería utilizada *Spacy*. Uno de los problemas (o ventajas, según como se mire) es la cantidad de cosas que calcula, lo cual lo hace muy lento. Esto se ve reflejado claramente en el filtrado de textos generados o en la generación de relaciones en la base de datos. Aunque para esta aplicación, no se necesitan bajos tiempos de respuesta, si se suman los costes de los distintos módulos sin control podría llegar a ser excesivo. Por ello *Spacy* tiene aún un gran potencial por ser explotado, pero podría ser demasiado potente para lo necesario en el sistema, se debe considerar (en una supuesta continuación del desarrollo) si utilizar herramientas más ligeras o sacar mayor partido a las funcionalidades que propone.

7.1.5. Conclusiones de la sección

El hecho de que todas estas herramientas estén bien definidas y delimitadas, podría permitir que una empresa asignara cada una de ellas a un grupo de personas distinto, que

se especialicen en ese campo y puedan llevar el desarrollo a niveles más altos.

7.2. Conclusiones generales

En este TFG se ha desarrollado un servicio completo de generación de interacciones automáticas inteligentes en *Twitter*. En concreto se ha realizado: el análisis del mercado actual, el diseño del plan de negocio, el desarrollo completo de la interfaz de usuario, la herramienta para el entrenamiento de los distintos algoritmos de generación de bots, el servidor central para la configuración, generación de las redes de bots especializados y la sincronización de las diferentes ejecuciones de Python, la implementación de la base de datos central y el módulo para publicación en la red social *Twitter*. Por tanto, se ofrece un servicio que permite al cliente configurar una red de bots con las características deseadas para interactuar con perfiles de *Twitter* mediante una interfaz amigable. El servidor coordinará a los bots de la red para participar e interactuar con los mensajes publicados por la cuenta indicada. El usuario decidirá qué bots responden al mensaje principal y qué bots participan en conversaciones derivadas y se responden entre sí. Los perfiles operativos de *Twitter* se almacenan en la base de datos central, donde se habrán creado en el momento en el que el cliente haya contratado el servicio. El proceso de inclusión de usuarios en la base de datos requiere autorizar previamente a nuestra aplicación a publicar mediante otro de los módulos desarrollados.

En el análisis del mercado y el plan de negocio se ha elaborado una estrategia para la implementación de una empresa que desarrolle y ofrezca el servicio. Si bien para encontrar financiación no se ha obtenido una opción segura y definitiva, se han ofrecido distintas opciones de las cuales se han analizado y ofrecido soluciones para los principales problemas que podrían surgir. Mediante el plan de marketing se han propuesto soluciones para la salida rápida al mercado, con un bajo interés y coste de financiación.

La interfaz web de usuario permite al cliente configurar las redes de bots que ineractuarán con sus perfiles de *Twitter* de forma fácil y rápida. Esta opción de configurarse su propia red también facilitará al equipo la creación de redes propias en caso de que el cliente no desee configurarlas él mismo. La gran cantidad de opciones parametrizables en la web permite la creación de bots con personalidades complejas y muy cercanas a la caracterización de cualquier usuario real.

Los mensajes y respuestas que se generan son de gran calidad y originalidad, siendo prácticamente indistinguibles de los publicados por un usuario real. Esto es el resultado de un proceso complejo de recopilación de conjuntos de datos, desarrollo de algoritmos apropiados y largos entrenamientos de redes de bots. Si bien el entrenamiento se ha visto que es costoso en tiempo y cómputo, esto no afecta a la experiencia del usuario ya que solo se realiza para la creación de las redes de bots especializados. Una vez se crea la red de bots con las distintas características, la interacción real con *Twitter* ofrece una gran esperiencia de usuario. La interacción con los mensajes publicados se produce prácticamente en tiempo real, con el delay que se esperaría para la interacción de un usuario real.

7.2.1. Líneas de trabajo futuro

El desarrollo se ha centrado en el uso de *Twitter* pero la modularidad de la arquitectura permite integrar el servicio con otras redes sociales, una mejora que ampliaría el mercado en el que se podría ofrecer el servicio.

La publicación actualmente se realiza exclusivamente en inglés, por lo que uno de los siguientes pasos a dar es desarrollar el servicio en castellano, lo que aumentaría el número potencial de clientes.

La generación de perfiles reales junto con la autorización a nuestra aplicación para publicar en *Twitter* no ha podido automatizarse al completo debido a las estrictas normas de uso de *Twitter*. Si bien se ha desarrollado un script para facilitar al máximo esta tarea, en un futuro sería interesante desarrollar una interfaz gráfica web para facilitarla aún más.

El alto tiempo de entrenamiento para los algoritmos se debe al coste en cómputo y memoria de estos. Este tiempo actualmente es demasiado alto debido a la falta de medios técnicos por el alto coste de grandes equipos de cómputo. Si bien como primera mejora ya se ha empleado una GPU, se vuelve necesario para los siguientes pasos utilizar equipos más potentes.

Capítulo 7

Conclusions

In this chapter are described the difficulties found during the development, and some features that could be improved in the future.

7.1. Bot's intelligence

When talking about a bot, lots of different parts of the system are involved, which work together to abstract themselves as a bot. Each one of them is well defined and has its problems associated, but in the end the idea of these pieces working together is what makes the system be as it is. This role distribution proposes a product with lots of potential. A hypothetical system with these pieces greatly refined could create personalities, something that is far ahead from the competitors in the market. This splitted model also allows the development or refactorization in case it was needed, setting clearly the objective of each piece. This was one of the main problems found, having the objective of creating individual and "human-behavioural"bots, splitting the system into small codable modules.

Despite this was actually achieved, the different parts bring new problems that are not simple although they are better defined. This will be described and discussed in the following sections.

7.1.1. Text generation

Generator

Despite of GPT-3 being on the market since the beginning of the development, the implementation was made with GPT-2. This was caused by the fact that GPT-3 only allows fine-tuning through real payments and in their servers. The GPT-2 library allows fine tuning in any place. Fine tuning is a key element of the system to reach the bots with personality goal, and this economic breach is what makes GPT-3 not to be considered.

With a bigger budget access, GPT-3 would be the choice. The usage of this tool would increase significantly the quality of the responses, as well as it would ignore team equipment limitations, due to hosting the fine-tuning on OpenAI servers.

Another thing to improve urgently is the time required to train a model, since right now this takes from 4 to 8 hours using a relatively small sized corpus. This time would strongly increase with bigger corpus. One option to solve this would be having a machine dedicated strictly to this purpose, with a GPU to run the training on. The GPU usage requires some adjustments in the libraries and in the code, but the biggest issue comes from having and maintaining a machine with those characteristics.

Last but not least, the other problem found was the attainment of a new corpus. This task is made right now manually with a script that collect tweets from Twitter using some keywords. This script could be improved, due to the need of running it multiple times for every corpus if it is desired to use different keywords. The script is far from being a good solution, since it calls the Twitter API every 15 min (a time limit established by Twitter) so most of the time the script is waiting. A solution could be gathering tweets from other sources, or enhancing it around the 15 minute limitation, using better automation that wasn't necessary for the moment.

Filter pipeline

The filter pipeline is a tool with lots of room for improvement, and easy development. This, along with other characteristics could allow better filtering, that could consider other features of a human, such as age, or other modifiers that would customize more the responses based on the profile of the bot. Another thing to consider is the order in which the functions are applied. This could change the result because, if some modifiers are executed before the evaluators, the final punctuation could be different since this changes are later read and evaluated. With a big volume of functions the order could have a big impact on the final result, leading to the consideration of having an interface to manage this, automatically or by the user.

7.1.2. Relationships database

In the implementation it was decided to use Prolog, with some simple rules. The development could follow this path improving the rules, or just take a completely different path. The objective of the database is to condense the relationship between two entities in a single parameter (a number from -1 to 1), therefore any service that provides this could be useful. An option is to use a neural network, that takes these two elements as the input and returns the number. To achieve this, not only would we need the technology, but also for it to be fed with the corresponding corpus, and the automation and adjustment of the corpus would not be trivial.

7.1.3. Inferencer

This tool, that leans strongly on the relationships database and configures how the filter pipeline is applied, has a development scope as big as the filter pipeline. Each parameter has its own algorithm, that could be developed and improved as much as desired. The obstacles

that came across are associated with each one of the parameters. The algorithms can be improved, but the good performance of this element depends strongly on the relationship database. A possible improvement (and this can be applied to other parts of the system) is to make the bot take in consideration more things apart from the immediate context, such as the tweets the bot has written, other tweets from the thread or even the profile description of the user it is responding to. With all this data the parameters could be greatly improved, but this has an associated problem, the scope growth and algorithm complexity, making the inference not viable to debug and enhance. It is mandatory that each one of the parameters (and their calculation) is well defined.

7.1.4. Sentiment analysis

This module has been developed around the requirements demanded by the rest of the system, working as a *wrapper* for *Spacy*, the dependency used. One of the disadvantages (or advantages, depending on how it is seen) is the amount of processing that it does, which makes it slow. This is clearly shown in the filtering of generated texts or in the generation of relationships inside of the database. Even though this application does not really require low response times, the result of adding the cost of all modules without control could end up being excessive. That is why *Spacy* still has room for improvement. It could, however, be too much for what the system needs, so using lighter tools should be considered (in case the development is continued), or improving the usage of the options that it provides.

7.1.5. Section conclusions

The fact that all these tools are properly defined and delimited could let a company easily assign a specialized group of people to each one, allowing a higher level development.

7.2. Main conclusions

In this Final Degree Project, a service for the generation of automatic interactions on *Twitter* has been developed. The following steps were carried out: the analysis of the current market, the design of the business plan, the development of the user interface, the tool for training the different bot algorithms, the central server for configuration, generation of the botnets and synchronization of different Python executions, the implementation of the central database and the module for publication in the social network *Twitter*. Therefore, a service is offered that allows the client to configure a botnet with the desired characteristics to interact with *Twitter* profiles through a friendly interface. The server will coordinate the bots of the network to participate and interact with the messages published by the account that has been selected. The user will decide which bots respond to the main message and which bots participate in secondary conversations and interact between each other. The operational profiles of *Twitter* are stored in the central database, where they have been created when the client has contracted the service. The process of adding users into the

database requires to previously authorize our application to publish through another of the developed modules.

In the market analysis and the business plan, a feasible strategy has been developed for the real implementation of a company that creates and offers the service. Although a fully safe and definitive strategy does not exist to find the initial funding, different options have been offered. These options have been analyzed and the main problems that could arise have been solved. Through the marketing plan, different solutions have been proposed for a quick integration in the market of services for social networks.

The web user interface allows the client to configure the botnets that will interact with their *Twitter* profiles both easily and quickly. This option to configure your own network will also make it easier for the team to create their own networks in case the client does not want to configure them himself. The large number of configurable options on the web allows the creation of bots with complex personalities and very close to the characterization of any possible real user.

The messages and answers that are generated are of great quality and originality, being practically indistinguishable from those published by a real user. This is the result of a complex process of collecting data sets, developing appropriate algorithms, and a thorough botnet training. Although training has been seen to be costly in terms of time and computation, this does not affect the user experience since it is done only for the creation of specialized botnets. Once the botnet with various features is created, the actual interaction with *Twitter* offers a great user experience. The interaction with the published messages occurs practically in real time, with the delay that is expected from the interaction of a real user.

7.2.1. Future work

Development has focused on the use of *Twitter*, but the modular architecture allows the service to be integrated with other social networks, an improvement that would expand the market in which the service can be offered.

The publication is currently carried out exclusively in English, so one of the next steps to take is to develop the service in the Spanish language, which would increase the potential number of clients to whom the service can be offered.

The actual *Twitter* profiling generation, along with the authorization for the posting to *Twitter* could not be fully automated due to strict *Twitter* usage rules. Although a script has been developed to make this task as easy as possible, it would be interesting to develop a graphical web interface to make it even easier in the future.

The lengthy training time for the algorithms is due to their computational and memory cost. This time is currently too high due to the lack of technical means because of the high cost of a large computer equipment. Although a GPU has already been used as a first improvement, it becomes necessary to get more powerful equipment in the following steps.

Capítulo 8

Contribuciones

A continuación exponemos las contribuciones de cada integrante del proyecto, así como los principales obstáculos a los que han tenido que enfrentarse.

8.1. Enrique Ballesteros Horcajo

Mi contribución se ha centrado principalmente en tres áreas: el desarrollo del estado del arte, la creación del business plan y la programación y diseño de la interacción con la red social *Twitter*. Las dos primeras partes en una fase inicial se realizaron conjuntamente entre todos los integrantes del equipo y fueron desarrolladas en profundidad por mí una vez se tuvo una aproximación consensuada. Para la interacción con la red social Twitter primero he realizado un análisis de las distintas opciones ya desarrolladas y finalmente he decidido crear un desarrollo propio buscando la mejor integración con la API de *Twitter* del resto módulos del proyecto.

En la creación del análisis del estado del arte se han desarrollado dos líneas principales de investigación: una centrada en herramientas y empresas de habla hispana y otra para todo el mercado internacional. Los prinicpales puntos de análisis en ambos casos han sido los siguientes:

- La red social *Twitter*, foco principal del estudio por su interés al ser una red social centrada en la interacción mediante mensajes de texto escrito.
- La red social *Facebook*, por su opción de comentarios tanto para perfiles de usuario como páginas de empresa. Estos comentarios centran la interacción y también están compuestos en su mayor parte por texto.
- Los precios que ofrecen las distintas empresas online para la compra de interacciones en ambas redes sociales, con el objetivo de tener un análisis completo de los costes medios.

El principal obstáculo en el análisis de las distintas empresas analizadas ha sido parametrizar cada uno de los indicadores que querían representarse y estandarizarlos para poder

obtener datos medibles. Cada una de las empresas ofertaba paquetes para interacciones de muy distintos tipos, por lo que se ha tenido que limitar el tipo que se analizaban con el objetivo de sacar conclusiones comunes a todas.

Una vez finalizado el estado del arte, he desarrollado el business plan centrándome en la creación de una empresa sostenible en base al análisis realizado previamente. Utilizando los costes estimados necesarios para el desarrollo de la actividad, se ha buscado crear un plan de negocio competitivo con las oferaas del mercado analizadas en el estado del arte. Los principales puntos del business plan han sido:

- Estimación de los principales gastos fijos relevantes para el funcionamiento del negocio.
- Elaboración de un modelo de negocio sostenible a largo plazo.
- Realización de varios modelos de negocio y un plan de marketing viable.
- Análisis DAFO del business plan: debilidades, amenazas, fortalezas y oportunidades.

El mayor obstáculo para el desarrollo del business plan ha sido cómo estimar el gasto en servidores necesario para ejecutar los distintos módulos que componen todo el entorno. Para solucionarlo ha sido necesario realizar varias ejecuciones de los distintos componentes del entorno en distintos equipos, y con los resultados obtenidos se ha realizado una estimación. Otro gran obstáculo ha sido encontrar un modelo de financiación viable y que pudiese llevarse a la práctica en este mismo momento con los recursos de los que disponemos. Analizadas todas las opciones se ha decidido desarrollar dos soluciones alternativas e incluso complementarias: primero buscando financiación a través de inversión externa centrándonos en aceleradoras de *Startups*, que consideramos la opción con mayor posibilidad de éxito, y segundo aportando nosotros mismos el capital inicial necesario.

El último punto más destacable que he realizado ha sido el módulo de interacción con la red social *Twitter*, con cuatro puntos principales:

- La obtención de los distintos tuits publicados en la red social y su envío al servidor central para su procesamiento.
- La publicación en la red social de los distintos mensajes y respuestas que se han generado para cada uno de los bots, incluyendo las interacciones entre ellos.
- La gestión de las distintas autorizaciones que requiere *Twitter* para poder tanto obtenter como publicar tuits.
- El almacenamiento de toda la información sobre cuentas y sus autorizaciones.

El principal obstáculo ha sido cumplir en todo momento con los distintos términos de uso de *Twitter* para la utilización de su API, lo que principalmente frena el desarrollo de herramientas automáticas para la publicación de todo tipo de interacciones en la red social. Una vez se ha analizado todo el conjunto de normas se ha podido desarrollar un módulo para

la publicacion tanto de mensajes como de respuestas respetando las mismas y cumpliento los términos de uso de *Twitter*.

Además de estos puntos principales, se han desarrollado tareas propias de un equipo de trabajo: eleaboración de la introducción, traducción de varias secciones, proposición de soluciones para distintos ámbitos, participación en

8.2. Sergio Calero Robles

Inicialmente mi misión ha consistido en diseñar y desarrollar toda la aplicación frontend, vistas, componentes, así como la arquitectura e implementación de todos los módulos (stores, servicios, etc.) requeridos para alcanzar los objetivos del proyecto. Esto incluye algunos puntos a destacar:

1. La creación de la vista de 'Nueva trollnet', el diseño e implementación de cada uno de los distintos componentes que sirven para configurar cada parámetro de la trollnet y sus bots, así como las opciones para darle nombre a la trollnet, para crear una red de ejemplo por defecto, y para deshacer el borrador de trollnet elaborado.

2. La creación de la vista de 'Galería' (así como el diseño e implementación de la lista mostrada, que contiene las vistas previas de las trollnets), donde se visualizan la lista de trollnets creadas, con sus acciones asociadas, así como el progreso de cada una en su creación y entrenamiento.

3. Los flujos existentes que se pueden hacer con cada trollnet, como crearlas, eliminarlas, activarlas y desactivarlas. Esto último precisamente contiene a su vez el flujo para introducir el nombre de la cuenta de Twitter en la que se escuchará, para poder iniciar el proceso de generar una conversación en la siguiente publicación de dicho perfil.

4. La implementación de los servicios que interactúan con la API REST del backend, así como el control de los flujos asociados para mostrar feedback al usuario cuando una acción del mismo finaliza en éxito o en fallo.

En paralelo al desarrollo del frontend, me comprometí también al diseño y desarrollo del servidor NodeJS que sirve de backend para la aplicación Frontend y de punto de entrada para la ejecución de todos los scripts de Python desarrollados por mis compañeros. Esto incluye una serie de puntos clave:

1. La implementación de la API REST que atiende al frontend, con el encaminamiento de cada petición hacia la ejecución de un flujo, el control de corrección de los parámetros de la misma y del flujo asociado, así como la elaboración de la respuesta enviada de vuelta al frontend dependiendo de todo lo anterior.

2. Idear e implementar varios flujos de procesos recursivos asociados al lanzamiento ordenado y en serie de los scripts de Python. Esto fue necesario puesto que a la hora

de lanzar varios procesos de Python, ya sea crear y entrenar una ristra de bots uno detrás de otro, llamar a la generación de todos los mensajes de una conversación, etc. Podía surgir un problema de insuficiencia de recursos ya que algunos de estos scripts son altamente exigentes en cuanto a uso de memoria RAM (así como de CPU); de lo contrario, podrían concluir con fallos de los procesos o resultados inesperados o incompletos, y por lo tanto a no alcanzar el objetivo planteado por el proyecto.

3. Aprovechando la existencia de una dependencia npm disponible, idear e implementar el servicio para ejecutar los scripts de Python con sus definiciones de inputs y de outputs, para ejecutar correctamente dichos scripts y extraer los resultados deseados a partir de la información retornada por los mismos.

4. La creación, mantenimiento e interacción con la base de datos de MongoDB donde se almacena toda la información relacionada con las trollnets, los bots, los corpuses procesados, etc.

5. Aprovechando la existencia de una dependencia npm disponible, idear e implementar el sistema de detección de nuevos archivos en el sistema de archivos, utilizado para detectar los archivos de corpus que en última instancia ejecutan el flujo de entrenamiento de modelos a partir dichos archivos de corpus, y de marcarlos como usados una vez el entrenamiento ha sido completado exitosamente.

6. Para aumentar el grado de realismo de la conversación generada por los bots, dentro de una complejidad acotada, ideé e implementé un sistema para decidir la cantidad de intervenciones de los bots, el orden en que intervienen, y cuándo deciden responderse entre ellos en lugar de al hilo principal.

Finalmente, dado que he sido el único integrante del proyecto que dispone de un equipo con los requisitos de RAM necesarios para ejecutar los altamente exigentes scripts de Python, tuve que asumir la misión de aglutinar los diferentes repositorios desarrollados por todos los integrantes, sincronizarlos y preparar mi equipo personal para el entorno virtual requerido para los scripts de Python; y, por la misma razón, encargarme de ejecutar todas las pruebas End to End del proyecto. Esto incluye:

1. Ejecuciones individuales de todos los scripts de Python lanzados desde la infraestructura de backend y ayuda a mis compañeros a debugar los errores de dichos scripts durante su desarrollo y con casos de uso reales, proveyéndoles de los logs generados por sus scripts;

2. El lanzamiento de entrenamientos de modelos a partir de los corpuses .txt que me proveían mis compañeros;

3. Crear trollnets con todos sus parámetros, así como proveer a la MongoDB de las cuentas de Twitter y sus api y secret key para que las vinculaciones entre bots y cuentas reales tuviese efecto;

4. Lanzar las generaciones de conversaciones para su publicación en Twitter, y vigilar el correcto progreso del proceso.

8.3. Jaime Fernández Díaz

Mis tareas han sido principalmente el desarrollo de la lógica de aprendizaje y generación de mensajes de los bots junto a Ventura, la obtención de corpus para enseñar a los bots, y la unificación de la memoria en un mismo estilo (así como algunas pequeñas correcciones y revisión general de la misma).

En cuanto a la programación del código de los bots, si bien la estructura principal e implementación de librerías ha sido casi exclusivamente realizada por Ventura, yo me he encargado de las siguientes tareas:

1. El diseño e implementación del filtro de 'formalidad', el cual se encarga de transformar las respuestas candidatas del bot en otras similares pero con un aspecto más coloquial, de la jerga de internet. Si bien la implementación no fue excesivamente complicada, la elección de los parámetros y medidas empleadas para transformar el texto supuso hacer cierta búsqueda en distintas redes sociales y páginas (especialmente Twitter por motivos obvios) para analizar qué clases de expresiones carácterísticas de internet son más usadas, y así hacer una lista de las mismas. No sólo eso, sino que también se les asignó distintas probabilidades según cuan frecuentemente la gente las emplea en internet. Hubo que ajustar muchos valores y probabilidades haciendo multitud de pruebas hasta lograr resultados que parecieran lo más humanos posible.

2. La creación del filtro de palabras clave. Existen dos variantes, que creé por si una resultaba mejor que la otra. Ambas reciben una lista de 'keywords' (las palabras clave) que se espera tener en las respuestas y otorgan puntos según aparezcan. El primero simplemente comprueba cuántas de las palabras aparecen mientras que el segundo cuenta espeíficamente cuantas veces aparece cada palabra, favoreciendo más a los textos que repitan mucho una palabra clave. Como ambos podrían ser utiles en distintas ocasiones, creé los dos, aunque debido a la naturaleza de los tweets, que tienen un límite de 280 caracteres, al final se decidió usar solo el primer filtro.

3. El sistema de generación de hechos. Básicamente, hacemos uso de Prolog dentro de nuestro código para establecer relaciones entre dos términos, con valores entre -1 y 1. De esta forma podemos controlar lo positivas o negativas que serán las respuestas de los bots. Mi labor ha consistido principalmente en programar toda la lógica de aprendizaje automático de estos hechos, a partir del corpus inicial, comprobando todas las frases del texto y analizando su positividad, para así saber si las parejas de palabras que aparecen en la misma tienen una buena o mala relación. Posteriormente se realizará la media de esas puntuaciones, ya que varias parejas aparecen varias veces. El principal problema que surgió trabajando en esta parte del proyecto fue la constante aparición de caracteres extraños en los corpus recibidos, que posteriormente fue solucionado

en ambas partes (es decir, mejoré la detección de dichos caracteres pero también me aseguré de que los corpus generados produzcan menos caracteres de ese tipo).

4. La implementación de dos tipos de inferencer, el de slang y el de keywords. Es decir, la parte del código encargada de obtener automáticamente los parámetros de los filtros que desarrollé anteriormente, ya que no en todos los casos queremos otorgar esos parámetros manualmente, si no que queremos una red de mensajes más autónoma que los aprenda entre sí. Este desarrollo no fue muy complicado ya que generalmente consistía en realizar el método inverso al ya realizado en los filtros.

Para ampliar la oferta de temas sobre los que pueden hablar los bots, es necesaria una gran cantidad de corpus de mensajes humanos sobre los que puedan aprender, y que sean de unos temas específicos. Solo disponíamos de un buen corpus para aprender, pero era un tema muy específico, política americana, por lo que necesitábamos más. Así que me encargué de desarrollar un script que obtuviera grandes cantidades de mensajes de twitter sobre un tema específico automáticamente, y así poder obtener nuestros propios corpus sobre lo que queramos. La primera dificultad fue obtener una cuenta de desarrollador de Twitter, para poder tener acceso a la API mediante las claves que nos proporciona. Una vez hecho pude probar el script desarrollado, sin embargo el proceso fue muy lento ya que cada cierta cantidad de tweets (1500-2000) el script se quedaba parado unos minutos debido a que llegó al límite de llamadas. Este límite saltaba cada 15 minutos debido a la gran cantidad de tweets que estábamos obteniendo, pero eran necesarios. Además, hubo que ejecutar el script varias veces para un mismo corpus, ya que lo iba mejorando a medida que veía que se podían añadir ciertos filtros, de forma que evitemos tweets de spam o con caracteres extraños que perjudiquen el futuro aprendizaje de los bots. Finalmente creo que logré una buena versión del script, que genera corpus muy útiles para el aprendizaje sobre un tema o temas específicos.

Por último, cabe destacar mi contribución en la memoria, principalmente la de unificador. Básicamente, revisé la memoria entera con cuidado asegurándome que se usara el mismo estilo de escritura durante la misma, ya que, al fin y al cabo, somos varios miembros en el equipo, por lo que era inevitable que se emplearan distintas formas de redactar, y era necesario que alguien lo corrigiera para que se asemeje más a lo que habría escrito la misma persona. Además, aprovechando esta revisión, además de las correcciones necesarias de personas y tiempos verbales, así como cambiar algunas expresiones, también corregí pequeños errores que habían quedado sueltos.

Y como había tenido la ocasión de revisar la memoria entera, fue buena idea que también realizara la introducción de la misma, para poder así resumir lo que contiene y ofrecer una pequeña guía al lector de lo que viene a continuación.

8.4. Ventura Mateos Pérez

Mi papel principal ha sido el diseño e implementación de los bots en Python, la selección de librerías y creación los "wrappers". He diseñado la arquitectura y piezas que se han

utilizado en esta parte, así como su correspondiente documentación en esta memoria, que listo a continuación:

- Generación de texto: Elegí tras varias pruebas con otras tecnologías y librerías, la que creo que más se ajusta al objetivo del proyecto, GPT-2. El uso de esta librería tuvo una dificultad de tipo técnico, ya que se requería de una resolución de dependencias bastante específica, por lo que preparar el entorno en el que funcionase no fue fácil. Más tarde desarrollé la interfaz con esta librería, implementando el *Pipeline de filtrado*, en el que añadí los filtros de formato y positividad.

- Modelos: Para hacer uso de GPT-2 se necesitó desarrollar un módulo que permitiese entrenar facilmente modelos, además de conseguir guardarlos de una manera que permitiese que se seleccionen más tarde de manera adecuada al contexto. Para ello, a parte de implementar la interfaz con la librería, tuve la idea de utilizar keywords como sistema de "clasificación" de los modelos, haciendo así que se pudieran más tarde comparar entre ellos y asociarlos a los bots a los que mejor se adaptasen. El algoritmo de extracción de keywords fue desarrollado con este objetivo, junto con la implementación de la *model descriptor list*, que asocia los datos a cada modelo y permite que se guarden facilmente.

- Análisis de sentimiento: La preparación del uso de la librería *Spacy* tuvo dificultades similares a las de GPT-2, además de la necesidad de adaptar esta potente herramienta a nuestras necesidades. Para ello diseñé el *classifier* y *properties*, a modo de wrapper con algo de funcionalidad que nos facilitase el uso durante el resto del desarrollo.

- Bots: Para conseguir lograr simular personalidades entre los bots, cree la clase *Bot* que introduce la lógica necesaria para asociar los bots a los modelos, y que forma una manera de guardar los datos de los bots de manera que más tarde se puedan instanciar de nuevo.

- Parámetros de filtrado, base de datos de relaciones e inferencia: Siguiendo con el objetivo de conseguir personalidades en los bots, era necesario un sistema que fuera capaz de ajustar los parametros de filtrado para cada bot y contexto. Por ello elegí Prolog para implementar una base de datos de relaciones con algunas reglas que permitieran de manera sencilla guardar y consultar las relaciones en el mundo. Esto vino sumado a la implementación del *Inferencer*, que abstrae esta generación de parametros de filtrado, además de ocuparse de hacer la conexiónentre Python y Prolog. Además añadí la inferencia del factor de positividad y del parámetro keywords.

- Unificación de entidades: Para que la base de datos de relaciones fuera consistente y tratando de buscar unicidad de conceptos introduje la idea de utilizar Wikipedia como unificador, implementando esta función.

- Scripts y utils: Por último encapsulé todo lo anterior descrito en un fichero de código al que se llama con distintos argumentos para cada una de las tareas, abstrayendo por

completo este sistema y convirtiendolo en una ”caja negra”. Esto junto a otro módulo de utils permite que el sistema tenga un sistema de loggeo, contención de errores y otras funcionalidades que facilitasen tanto el desarrollo como la solución de problemas que surgen durante el desarrollo.

Bibliografía

[1] E. Arcoya. 5 herramientas para comprar seguidores en twitter, (2020). https://www.actualidadecommerce.com/comprar-seguidores-twitter.

[2] M. Camacho. El coste de un trabajador para la empresa, (2021). https://factorialhr.es/blog/coste-empresa-trabajador/.

[3] A. Cossío. Realidad y perspectivas de la inteligencia artificial en españa, (2018). https://www.pwc.es/es/publicaciones/tecnologia/assets/pwc-ia-en-espana-2018.pdf.

[4] J. Mir de xplora. Google ads, ¿qué precio, tarifas y cuánto cuesta la gestión de publicidad?, (2020). https://www.xplora.eu/precio-google-ads/.

[5] Statista Research Department. Redes sociales a nivel mundial con más usuarios activos, (2021). https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.

[6] A. Durán. 11 formas de financiar un proyecto, (2021). https://www.bbva.com/es/11-formas-de-financiar-un-proyecto/.

[7] Jobted. Sueldo del ingeniero informático en españa, (2021). https://www.jobted.es/salario/ingeniero-informático.

[8] Microsoft. Calculadora de precios, (2021). https://azure.microsoft.com/es-es/pricing/calculator.

[9] J. Quaglia. Celebridades y políticos compran seguidores para obtener falsa popularidad, (2013). https://www.huffingtonpost.es/juan-quaglia/celebridades-y-politicos-_b_3069721.html.

[10] A. Reyna. ¿qué es una aceleradora de 'startups'?, (2019). https://www.bbva.com/es/que-es-una-aceleradora-de-startups/.

[11] Startupxplore. Principales aceleradoras de españa, (2021). https://startupxplore.com/es/blog/aceleradoras-espana-principales-como-funcionan-demasiadas/.

[12] E. Tornos. Tasa de interacción en twitter, (2021). https://eduardotornos.com/tasa-de-interaccion-engagement-twitter/.

[13] Inc. Twitter. Política de desarrolladores de twitter, (2020). https://developer.twitter.com/es/developer-terms/agreement-and-policy.

[14] Inc. Twitter. Cuánto cuesta un servicio de marketing digital en zaask, (2021). https://www.zaask.es/cuanto-cuesta/marketing-digital.

[15] Inc. Twitter. Políticas y reglas de la api de twitter, (2021). https://help.twitter.com/es/rules-and-policies/twitter-api.

Apéndice A

Imágenes del Estado del Arte y del Business Plan

Se incluyen en este apéndice las capturas, imágenes y tablas referenciadas en los capítulos del Estado del Arte (capítulo 2) y del Business Plan (capítulo 3).

Figura A.1: *Web compra-seguidores.com*

Figura A.2: *Compra de Me Gusta para Facebook*

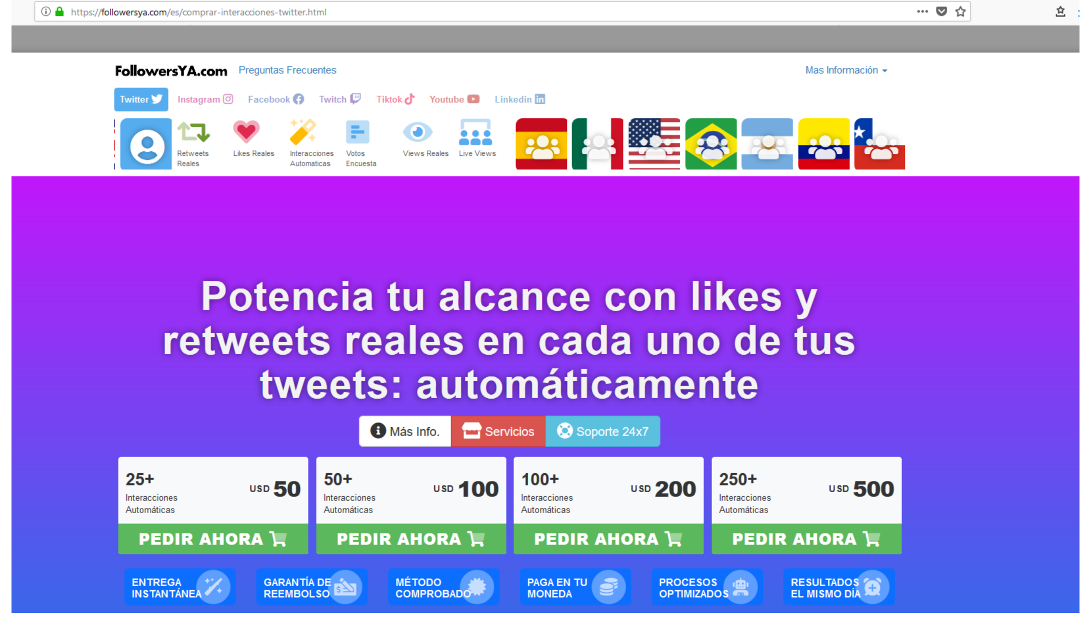


Figura A.3: *Interacciones ofrecidas por FollowersYA*

Figura A.4: *Interacciones ofrecidas por FollowersYA*

Figura A.5: *Compra de seguidores ofrecida por compraseguidores.com*

Figura A.6: *Compra de seguidores ofrecida por compraseguidores.com*

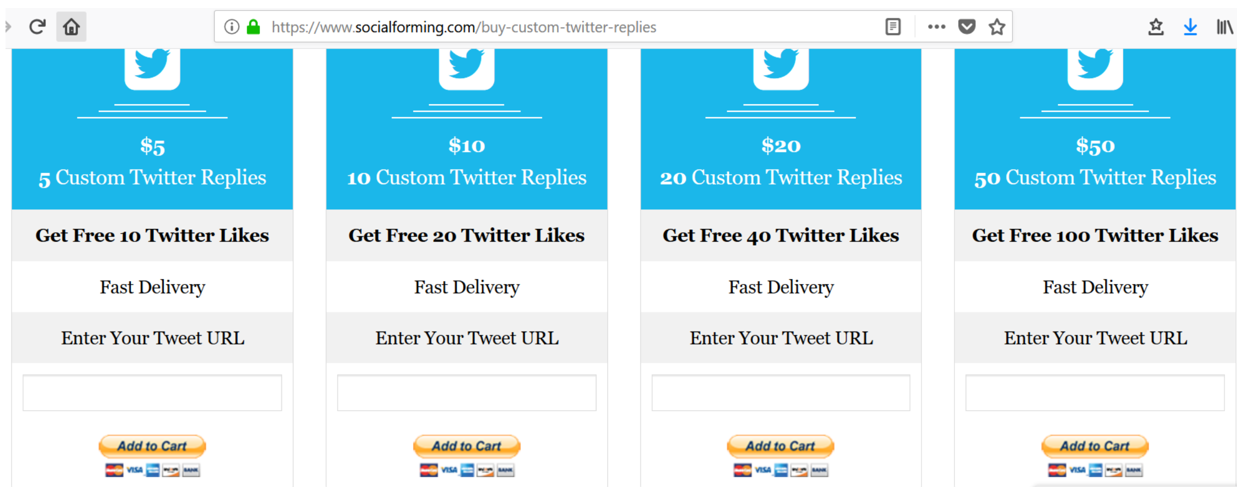


Figura A.7: *Compra de respuestas personalizadas de Socialforming*

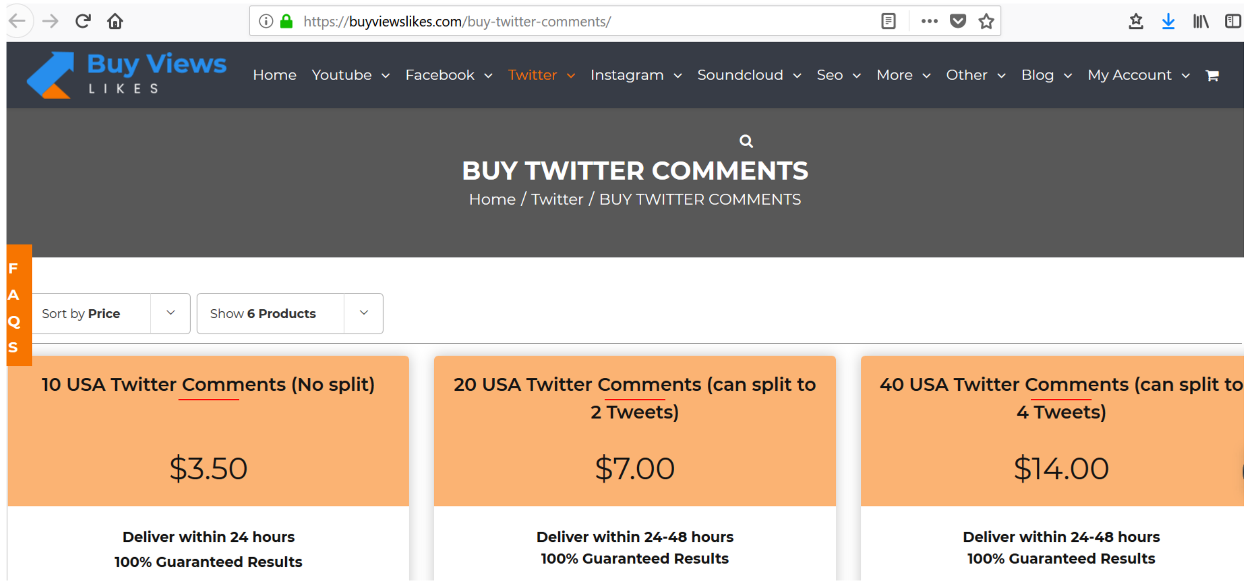


Figura A.8: *Compra de respuestas de Buyviewslikes*

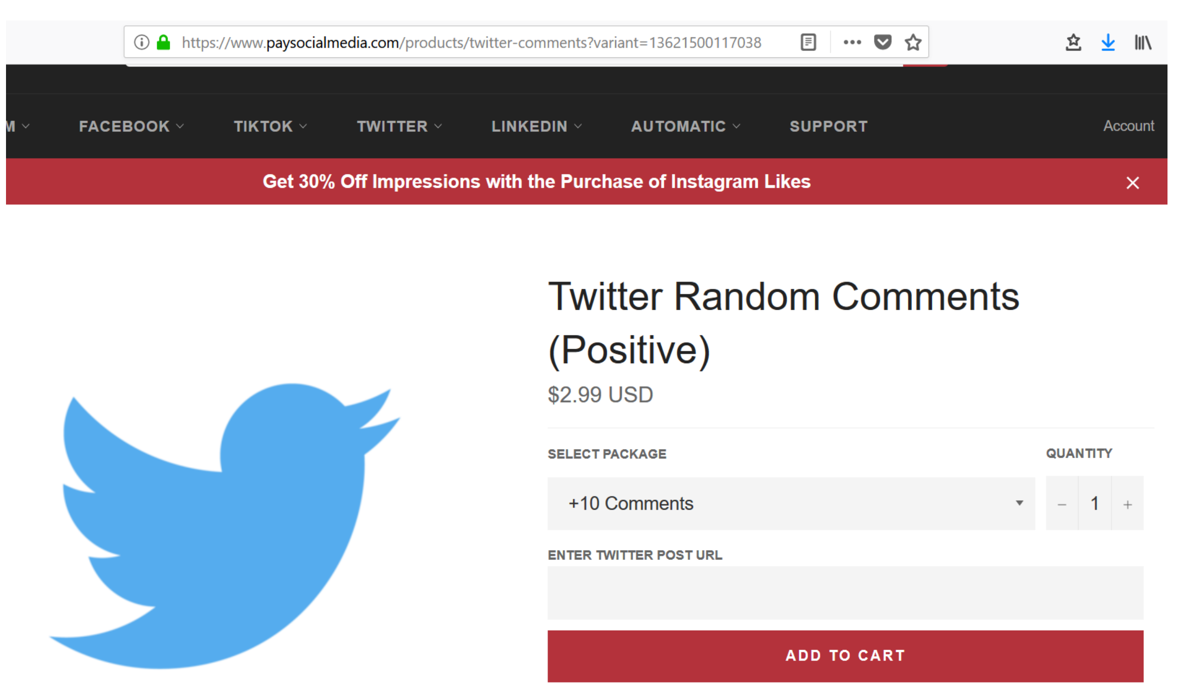


Figura A.9: *Compra de respuestas de Paysocialmedia*

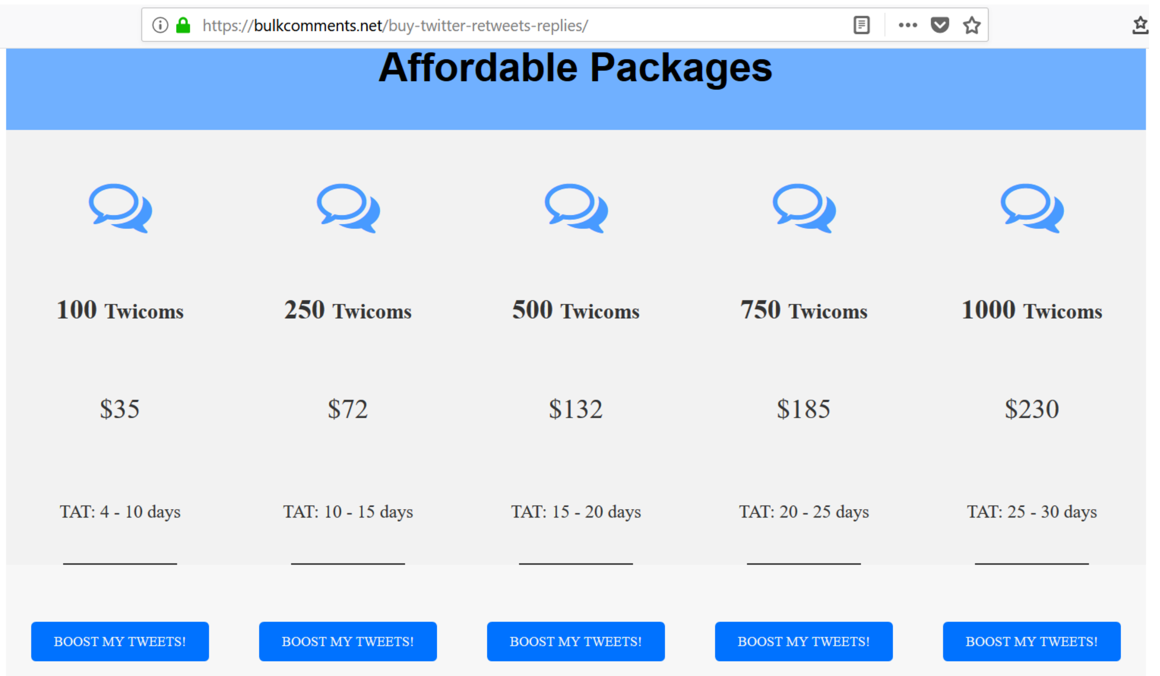
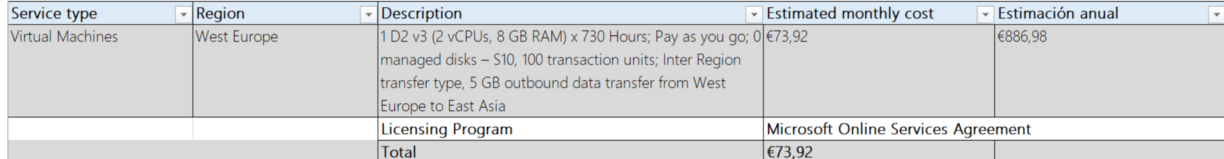


Figura A.10: *Compra de respuestas de Bulkcomments*

Ingenieros informáticos	Sueldo bruto mensual	Sueldo más impuestos	Sueldo bruto anual	Sueldo anual
Ventura	1.400,00 €	1.819,60 €	19.600,00 €	25.474,40 €
Sergio	1.400,00 €	1.819,60 €	19.600,00 €	25.474,40 €
Jaime	1.400,00 €	1.819,60 €	19.600,00 €	25.474,40 €
Enrique	1.400,00 €	1.819,60 €	19.600,00 €	25.474,40 €

Figura A.11: *Salarios por ingeniero informático*



Service type	Region	Description	Estimated monthly cost	Estimación anual
Virtual Machines	West Europe	1 D2 v3 (2 vCPUs, 8 GB RAM) x 730 Hours; Pay as you go; 0 managed disks – S10, 100 transaction units; Inter Region transfer type, 5 GB outbound data transfer from West Europe to East Asia	€73,92	€886,98
		Licensing Program	Microsoft Online Services Agreement	
		Total	€73,92	

Figura A.12: *Coste estimado de un servidor de Microsoft Azure*

Total ingresos	Total gastos	Número de servidores	Balance
10.000,00 €	9.378,45 €	12	621,55 €

Cliente	Pack contratado	Unidades contratadas	Coste para cliente (con IVA)	Interacciones
Agencia Alpha	PRO	1	1.000,00 €	5000
Agencia Beta	PRO	1	1.000,00 €	5000
Agencia Gamma	PRO	1	1.000,00 €	5000
Agencia Delta	PRO	1	1.000,00 €	5000
Agencia Epsilon	PRO	1	1.000,00 €	5000
Agencia Lambda	PRO	1	1.000,00 €	5000
Agencia Pi	PRO	1	1.000,00 €	5000
Agencia Ro	PRO	1	1.000,00 €	5000
Agencia Sigma	PRO	1	1.000,00 €	5000
Agencia Zeta	PRO	1	1.000,00 €	5000

Figura A.13: *Balance mínimo sostenible por clientes PRO*

Figura A.14: *Coste de un servicio de marketing, media nacional*

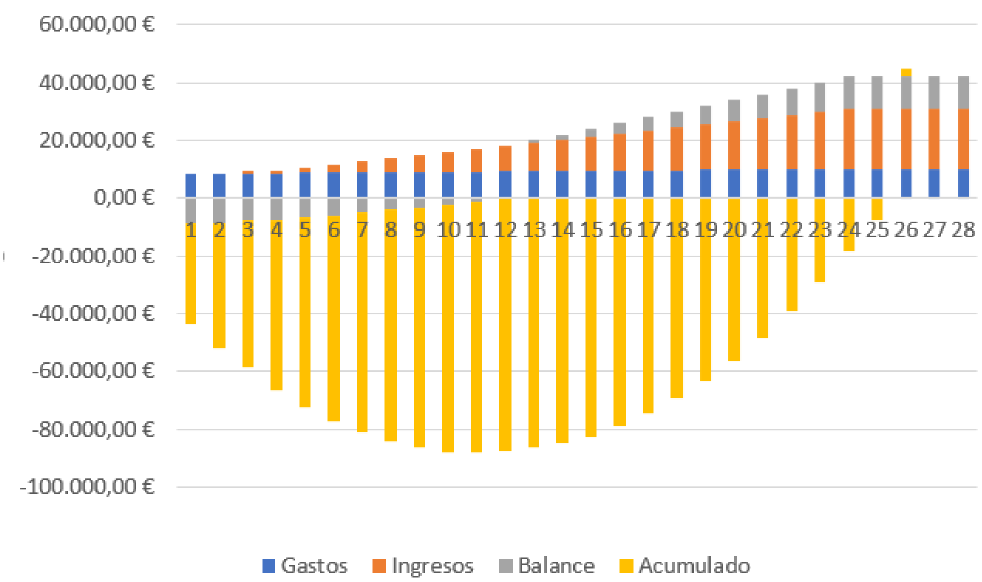


Figura A.15: *Evolución del balance con inversión inicial de 26.000 euros para marketing*

Mes	PRO	Servidores	Clientes P	Gastos	Ingresos	Balance
1	0	2	0	8.639,30 €	0,00 €	-8.639,30 €
2	0	2	0	8.639,30 €	0,00 €	-8.639,30 €
3	1	3	1	8.713,21 €	1.000,00 €	-7.713,21 €
4	1	3	1	8.713,21 €	1.000,00 €	-7.713,21 €
5	2	4	2	8.787,13 €	2.000,00 €	-6.787,13 €
6	3	5	3	8.861,04 €	3.000,00 €	-5.861,04 €
7	4	6	4	8.934,96 €	4.000,00 €	-4.934,96 €
8	5	7	5	9.008,87 €	5.000,00 €	-4.008,87 €
9	6	8	6	9.082,79 €	6.000,00 €	-3.082,79 €
10	7	9	7	9.156,70 €	7.000,00 €	-2.156,70 €
11	8	10	8	9.230,62 €	8.000,00 €	-1.230,62 €
12	9	11	9	9.304,53 €	9.000,00 €	-304,53 €
13	10	12	10	9.378,45 €	10.000,00 €	621,55 €
14	11	13	11	9.452,36 €	11.000,00 €	1.547,64 €
15	12	14	12	9526,28	12.000,00 €	2473,7199
16	13	15	13	9600,195	13.000,00 €	3399,8047
17	14	16	14	9674,111	14.000,00 €	4325,8894
18	15	17	15	9748,026	15.000,00 €	5251,9742
19	16	18	16	9821,941	16.000,00 €	6178,0589
20	17	19	17	9895,856	17.000,00 €	7104,1437
21	18	20	18	9969,772	18.000,00 €	8030,2284
22	19	21	19	10043,69	19.000,00 €	8956,3132
23	20	22	20	10117,6	20.000,00 €	9882,3979
24	21	23	21	10191,52	21.000,00 €	10808,483

Balance Total
7.508,53 €

Figura A.16: *Balance objetivo en los dos primeros años*

Figura A.17: *Aceleradorea de Startups Wayra*

Total ingresos	Total gastos	Número de servidores	Balance
2.400,00 €	8.713,21 €	3	-6.313,21 €

Cliente	Pack contratado	Unidades contratadas	Coste para cliente (con IVA)	Interacciones
Agencia Alpha	PRO	1	1.000,00 €	5000
Agencia Alpha	Premium	1000	150,00 €	1000
Agencia Beta	Semanal	50	1.250,00 €	3500
Agencia Gamma	Prueba	1	0,00 €	10

Figura A.18: *Balance mensual inicial*

Apéndice B

Imágenes de la Arquitectura e Implementación y de los Casos de Uso

Se incluyen en este apéndice las capturas, imágenes y tablas referenciadas en los capítulos de Arquitectura e Implementación 5 y Casos de Uso 6.

Figura B.1: *View-Logic-Store-Service Pattern*

Store-Api-DB Pattern

Backend

Specific Endpoint

Api Service

Store

DB Service

Local IndexedDB

Specific Table

Figura B.2: *Store-API-DB Pattern*

Api Service Classes Diagram

ApiTrollnetProvider

- Promise<TrollnetModel[]>: getAllTrollnets()
- Promise<TrollnetModel>: createTrollnet()
- Promise<any>: deleteTrollnet()
- Promise<any[]>: getUntrainedStatus()
- Promise<any>: renameTrollnet()
- Promise<any>: activateTrollnet()
- Promise<any>: deactivateTrollnet()

Inheritance

ApiGenericProvider

- Promise<any>: get()
- Promise<any>: create()
- Promise<any>: update()
- Promise<any>: delete()

private data

- url: string
- headers: any
- Observable<any>: handleError()

Inheritance

Inheritance

ApiPaymentProvider

- Promise<any>: getAccount()
- Promise<any>: verifyCard()
- Promise<any>: authorizePayment()

ApiUserProvider

- Promise<UserModel>: register()
- Promise<UserModel>: login()

Figura B.3: *API Providers Class Diagram*

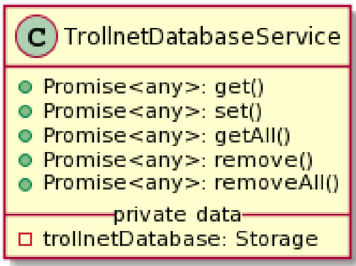


Figura B.4: *Database Service Class Diagram*

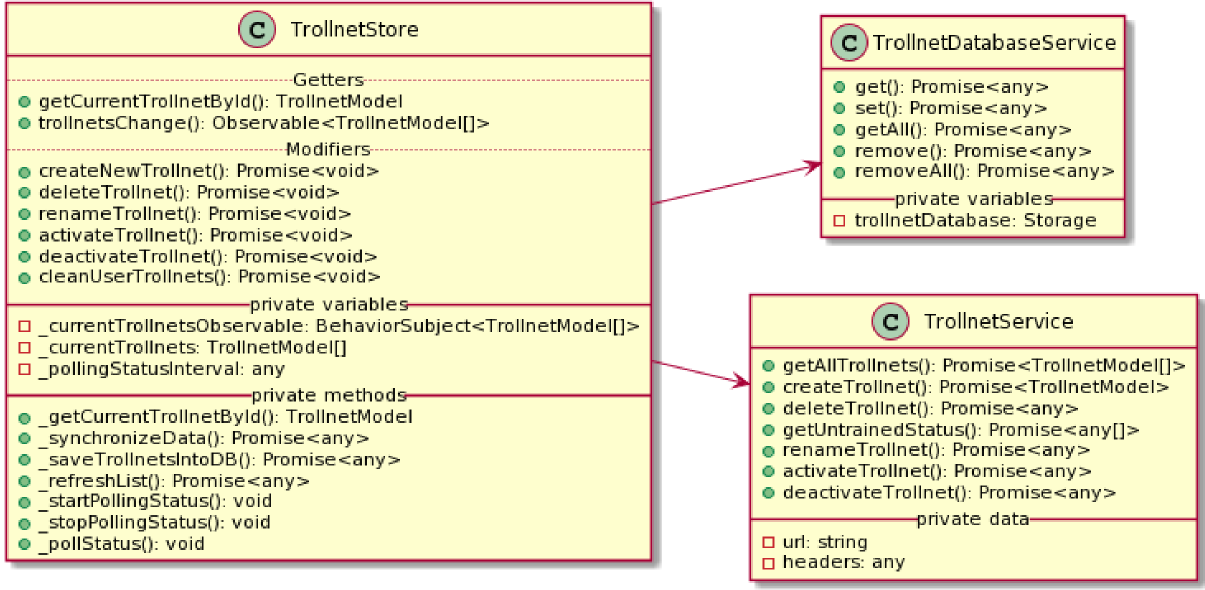


Figura B.5: *Trollnet Data Class Diagram*

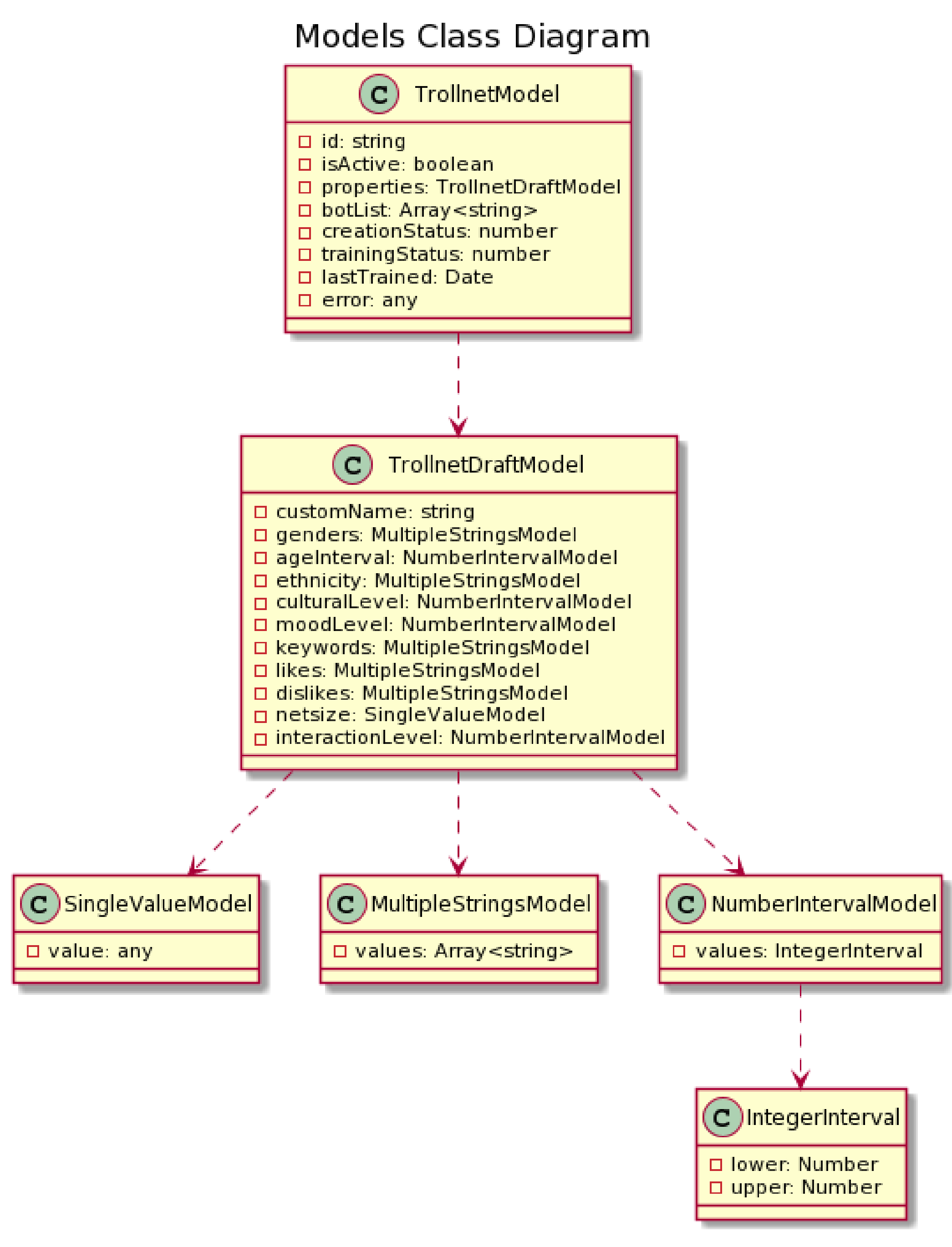


Figura B.6: *Models Classes Diagram*

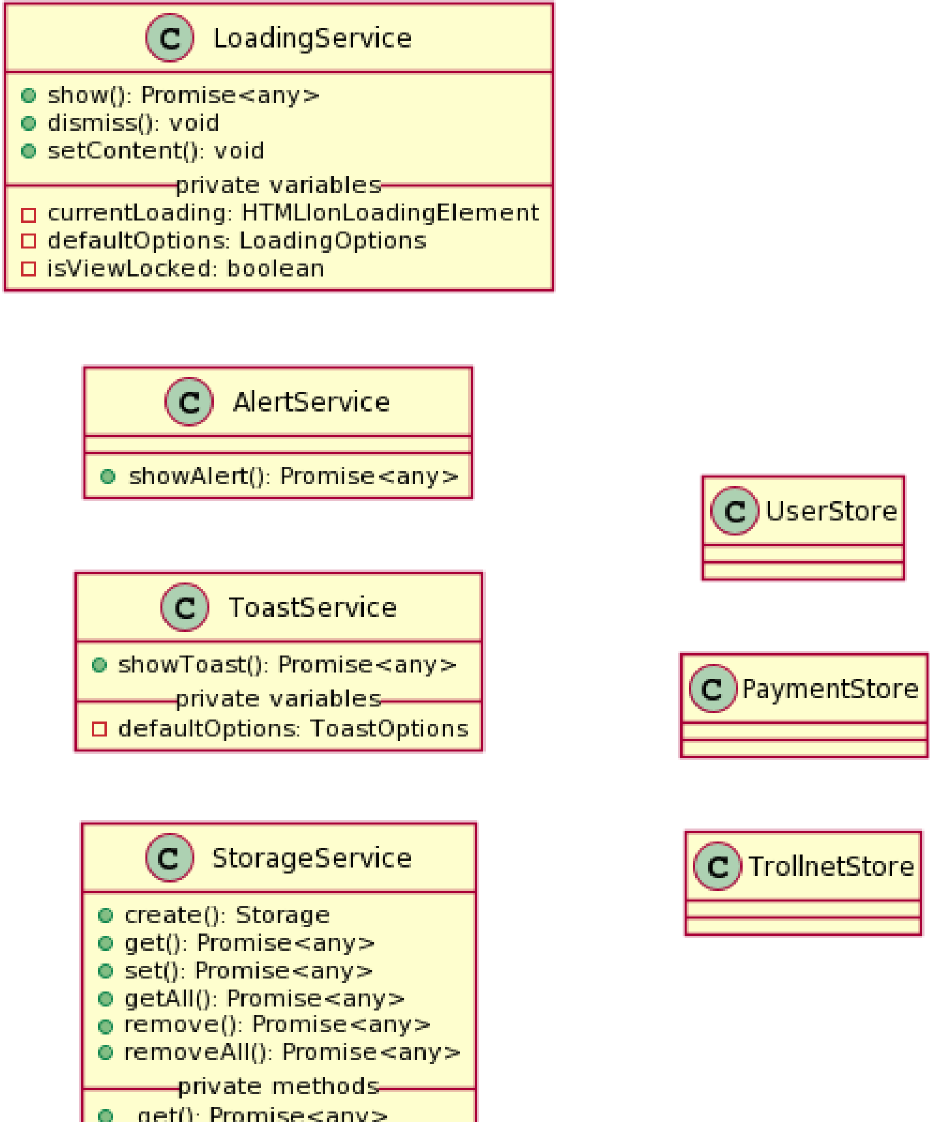


Figura B.7: *Services Class Diagram*

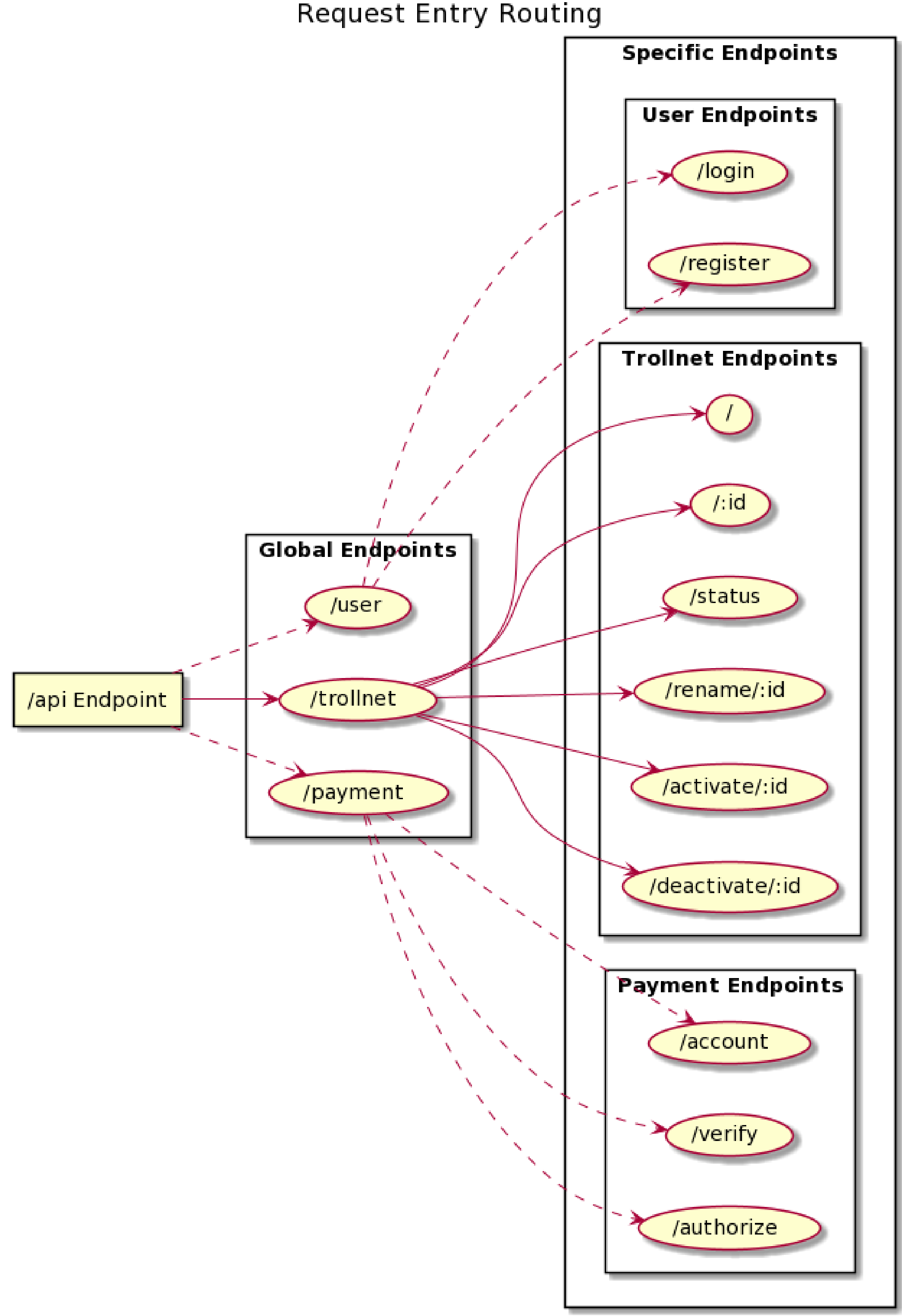


Figura B.8: *Base API Entry Schema*

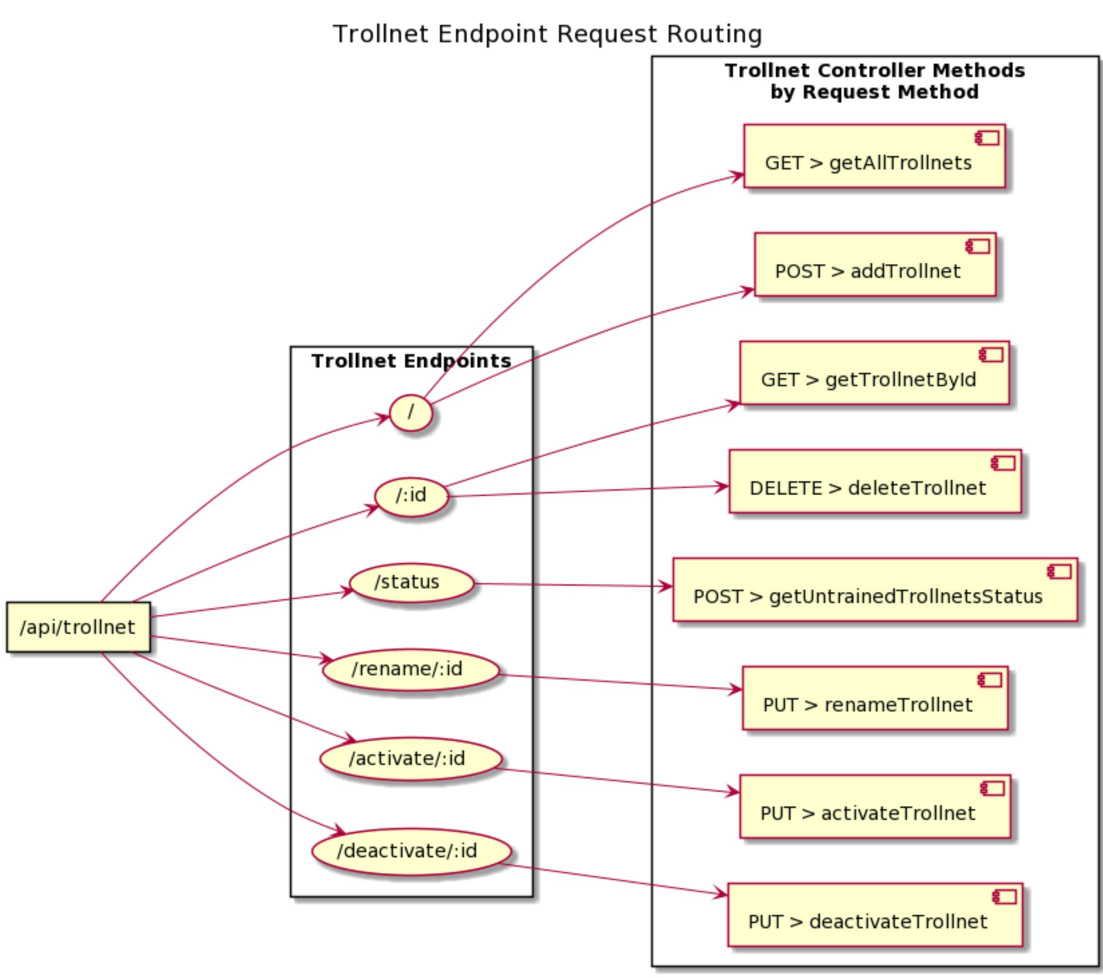


Figura B.9: *Trollnet API Schema*

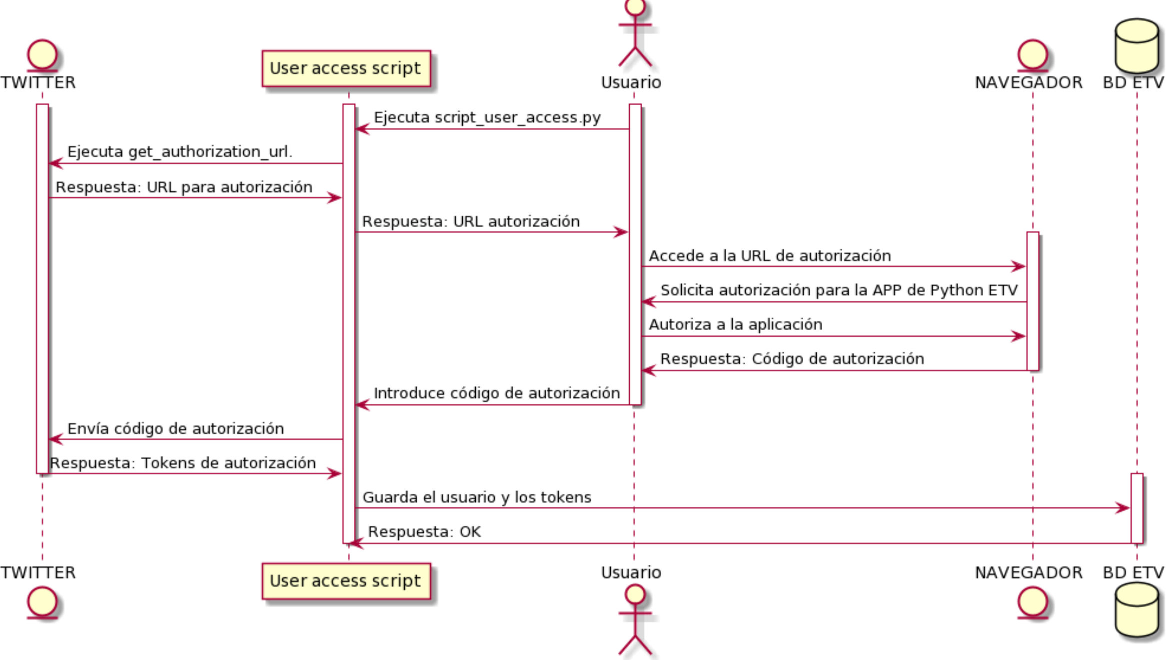


Figura B.10: *Diagrama de autorización a la APP de un usuario de Twitter*

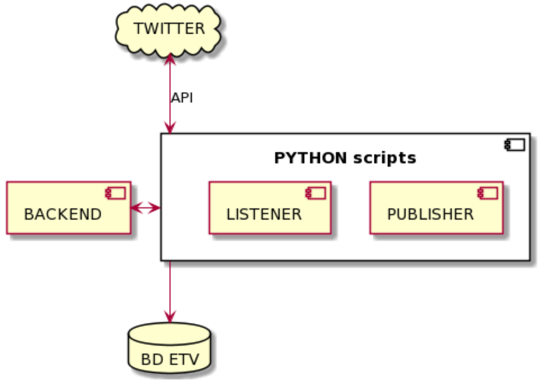


Figura B.11: *Arquitectura básica de interacción con Twitter*

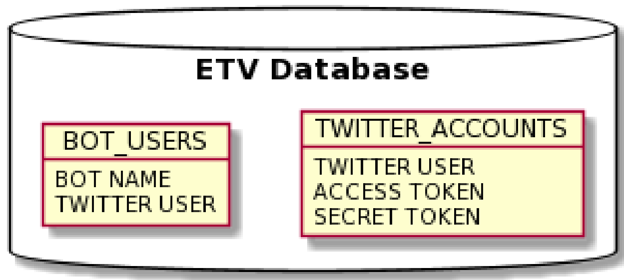


Figura B.12: *Arquitectura de la base de datos de interacción con Twitter*

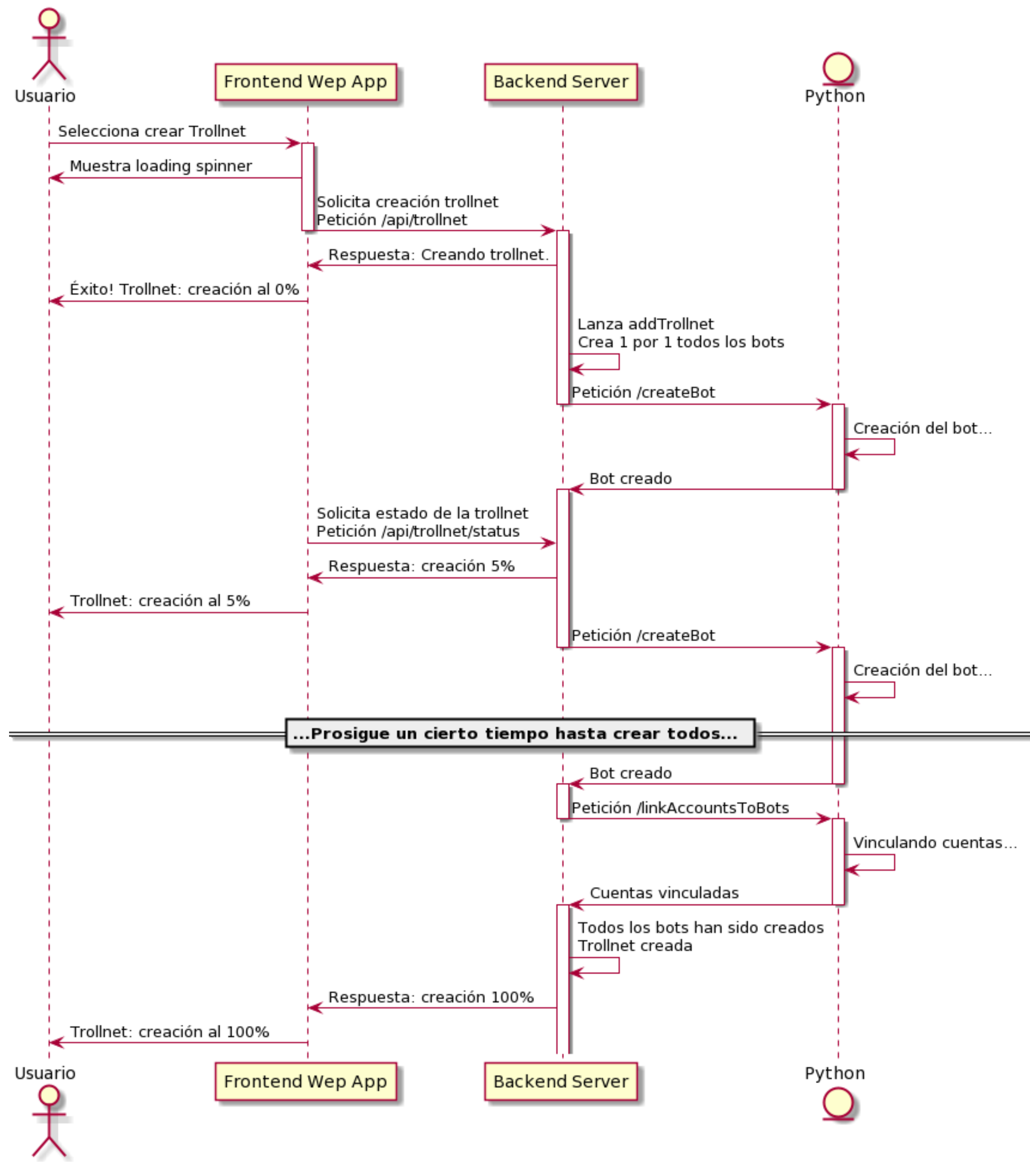


Figura B.13: *Diagrama de comunicación entre frontend, backend y lado Python para crear y entrenar una nueva trollnet*

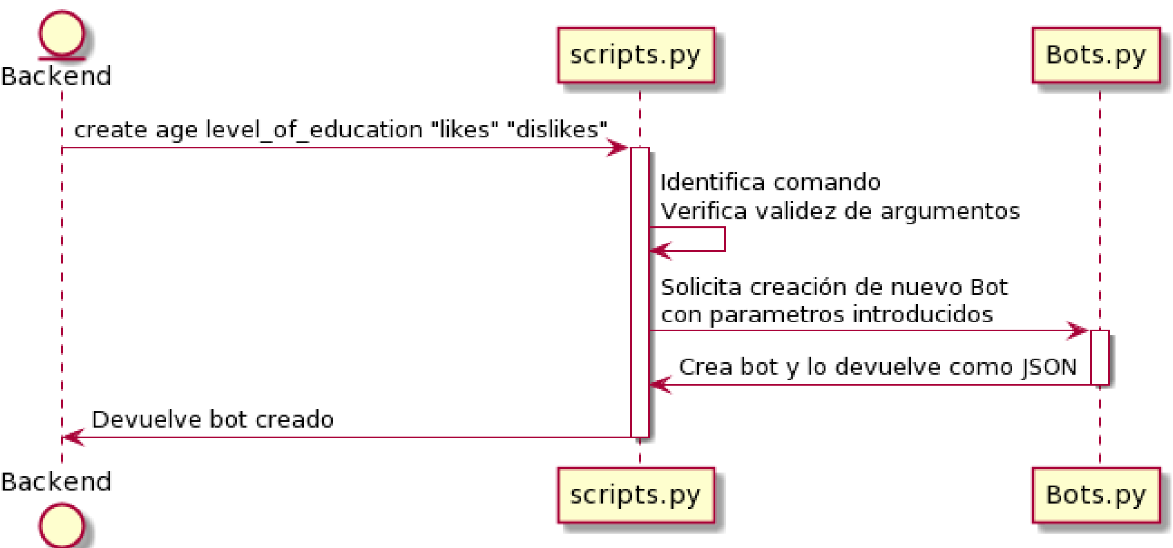


Figura B.14: *Diagrama de creación de un bot en entorno de Python*

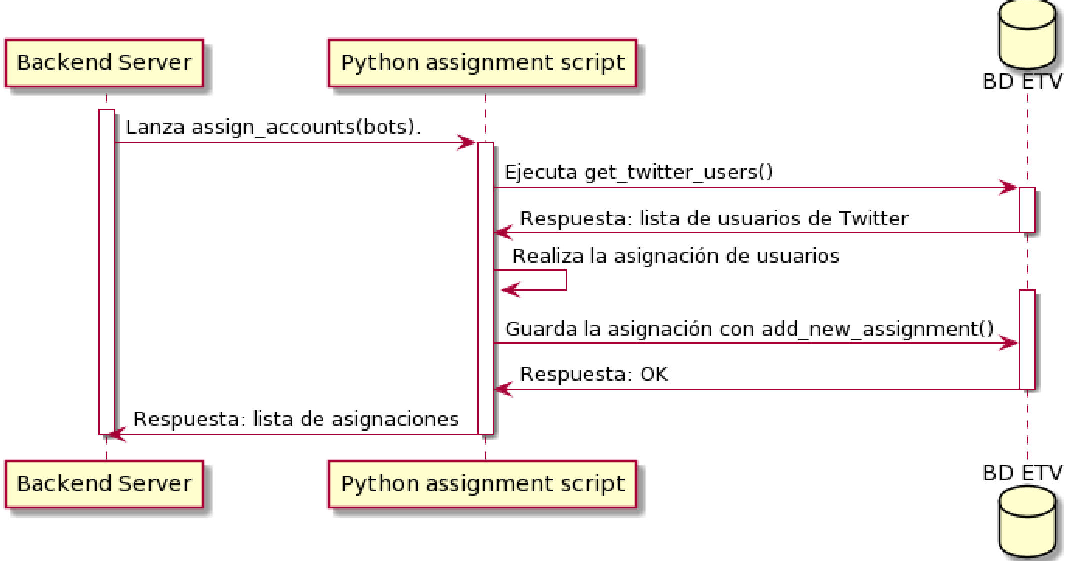


Figura B.15: *Diagrama de vinculación de un usuario de Twitter a un bot*

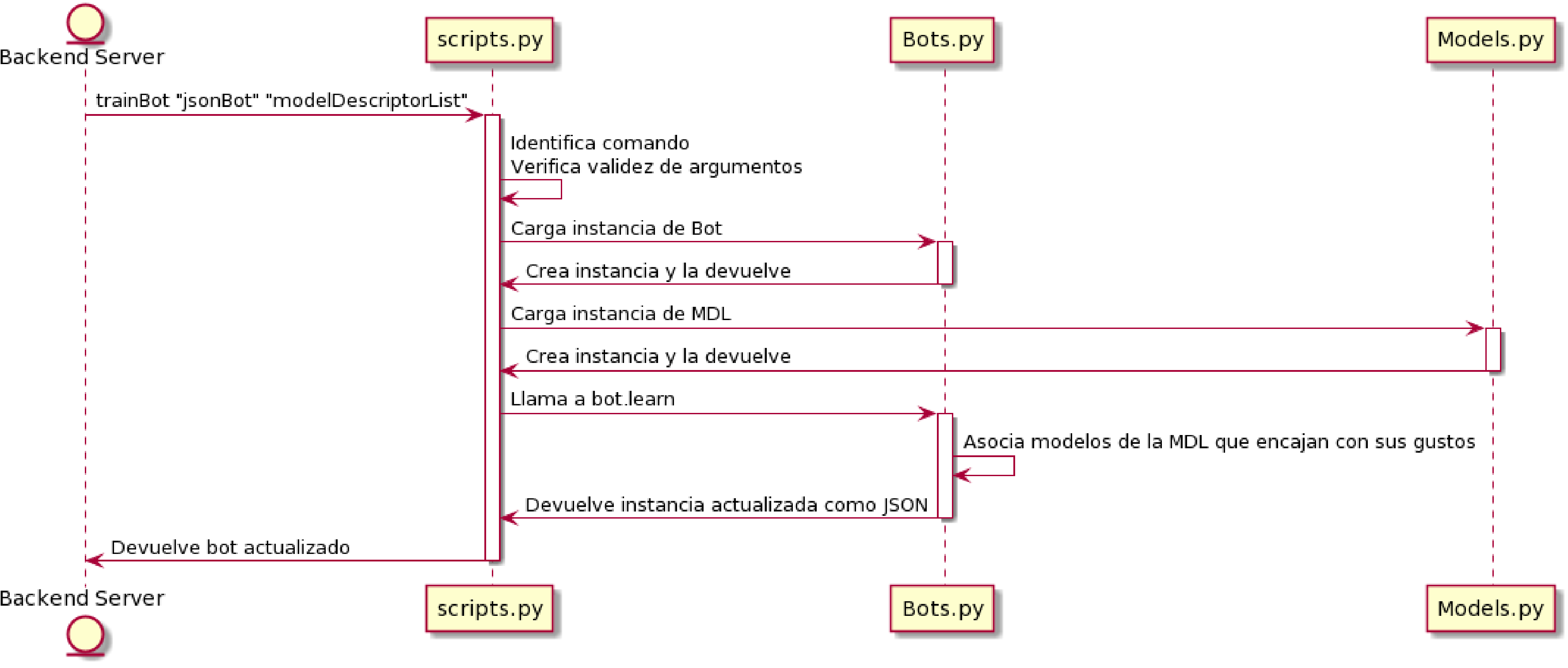


Figura B.16: *Diagrama de entrenamiento de un bot en entorno de Python*

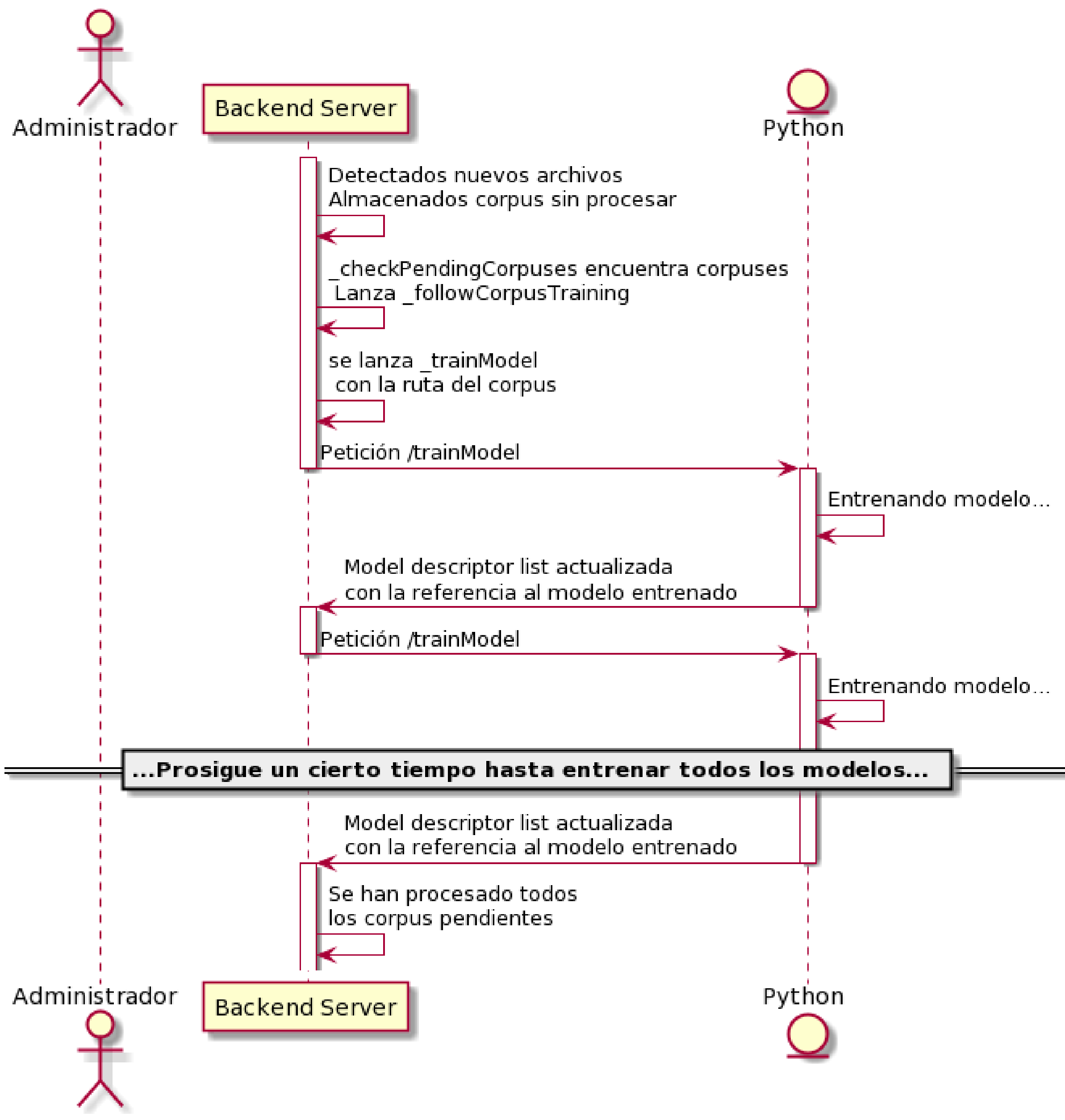


Figura B.17: *Diagrama de comunicación entre backend y el lado Python para entrenar un modelo a partir de un corpus*

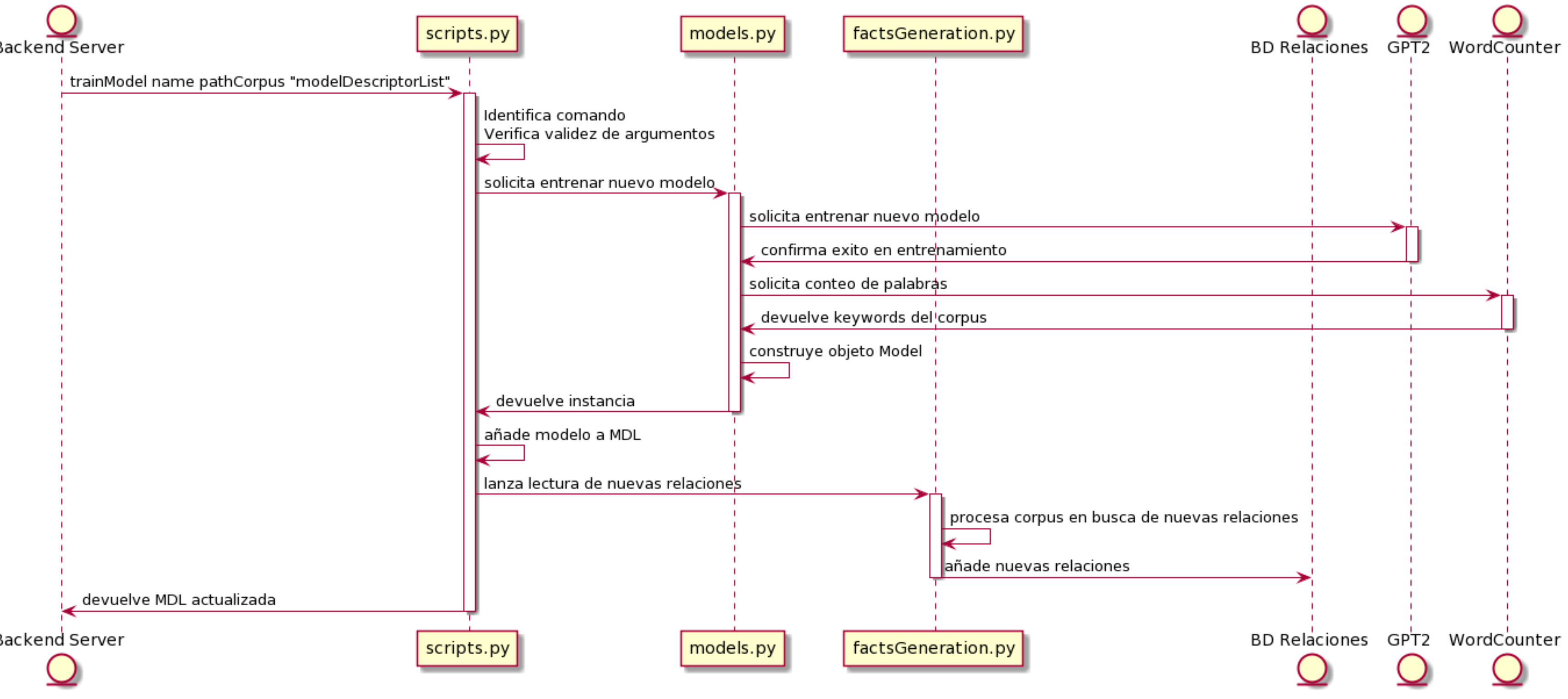


Figura B.18: *Diagrama de entrenamiento de un modelo en entorno de Python*

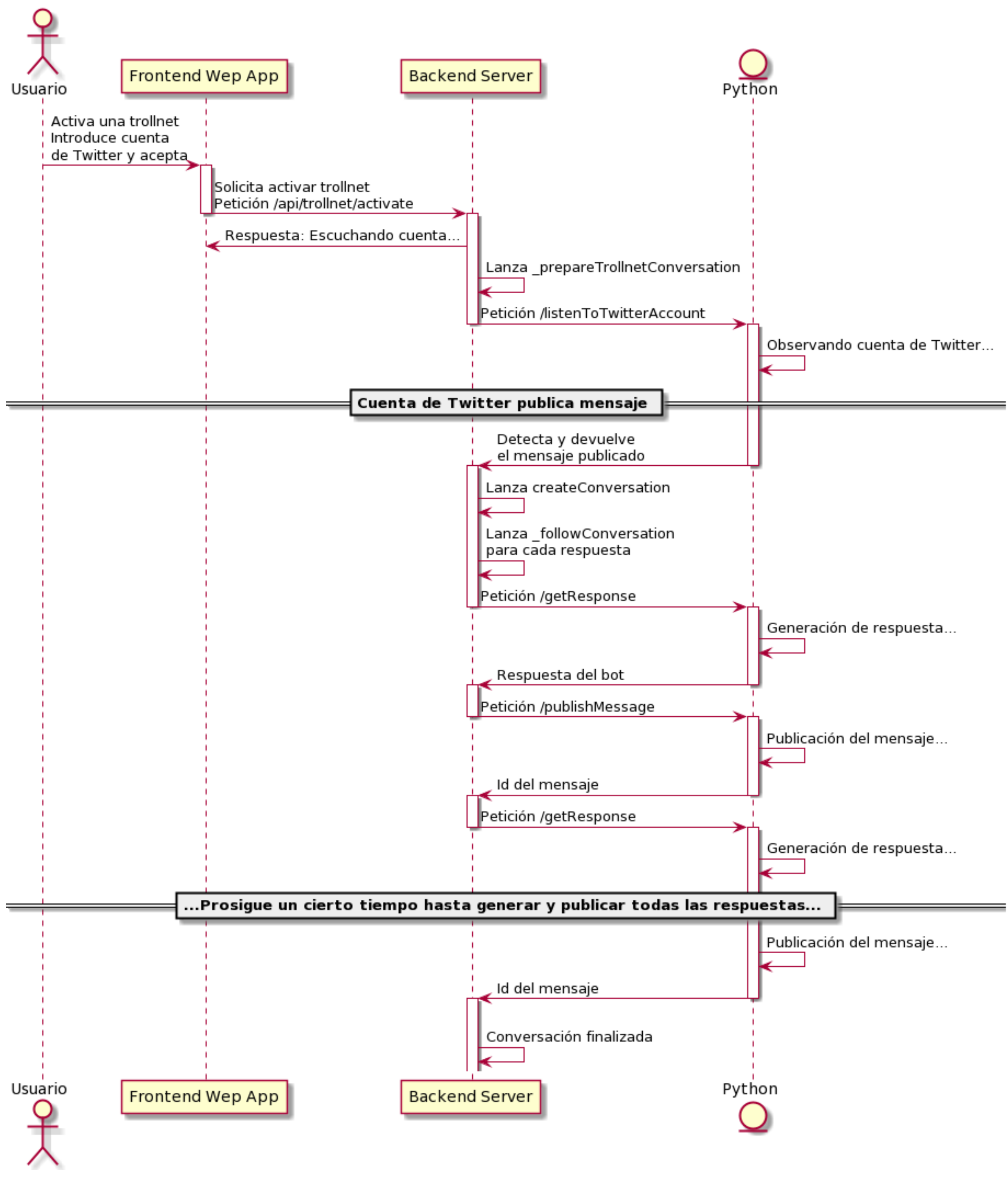


Figura B.19: *Diagrama de comunicación entre frontend, backend y el lado Python para generar una conversación*

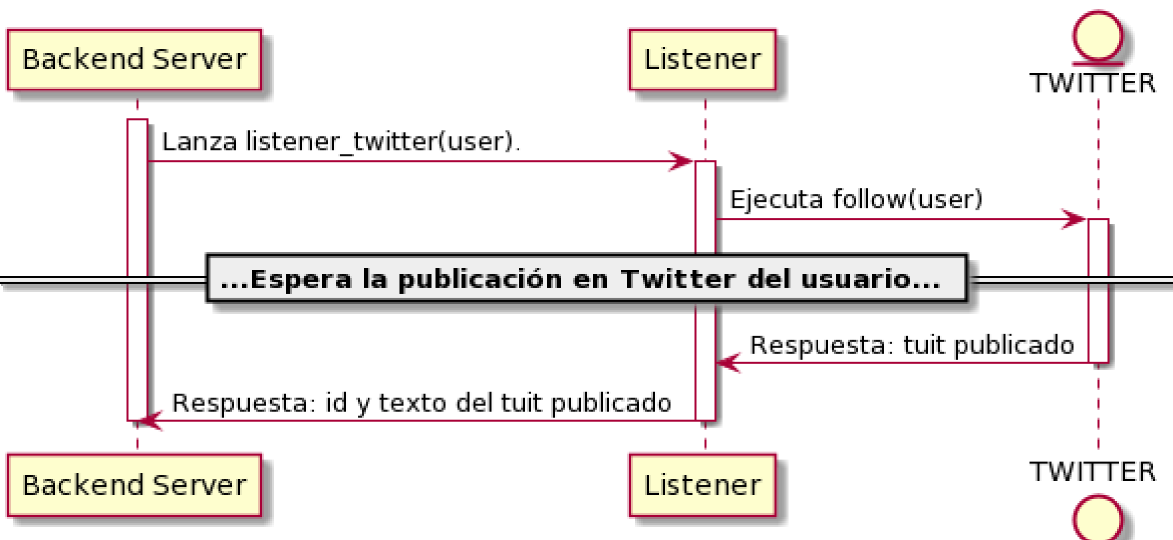


Figura B.20: *Diagrama del listener de Twitter en Python*

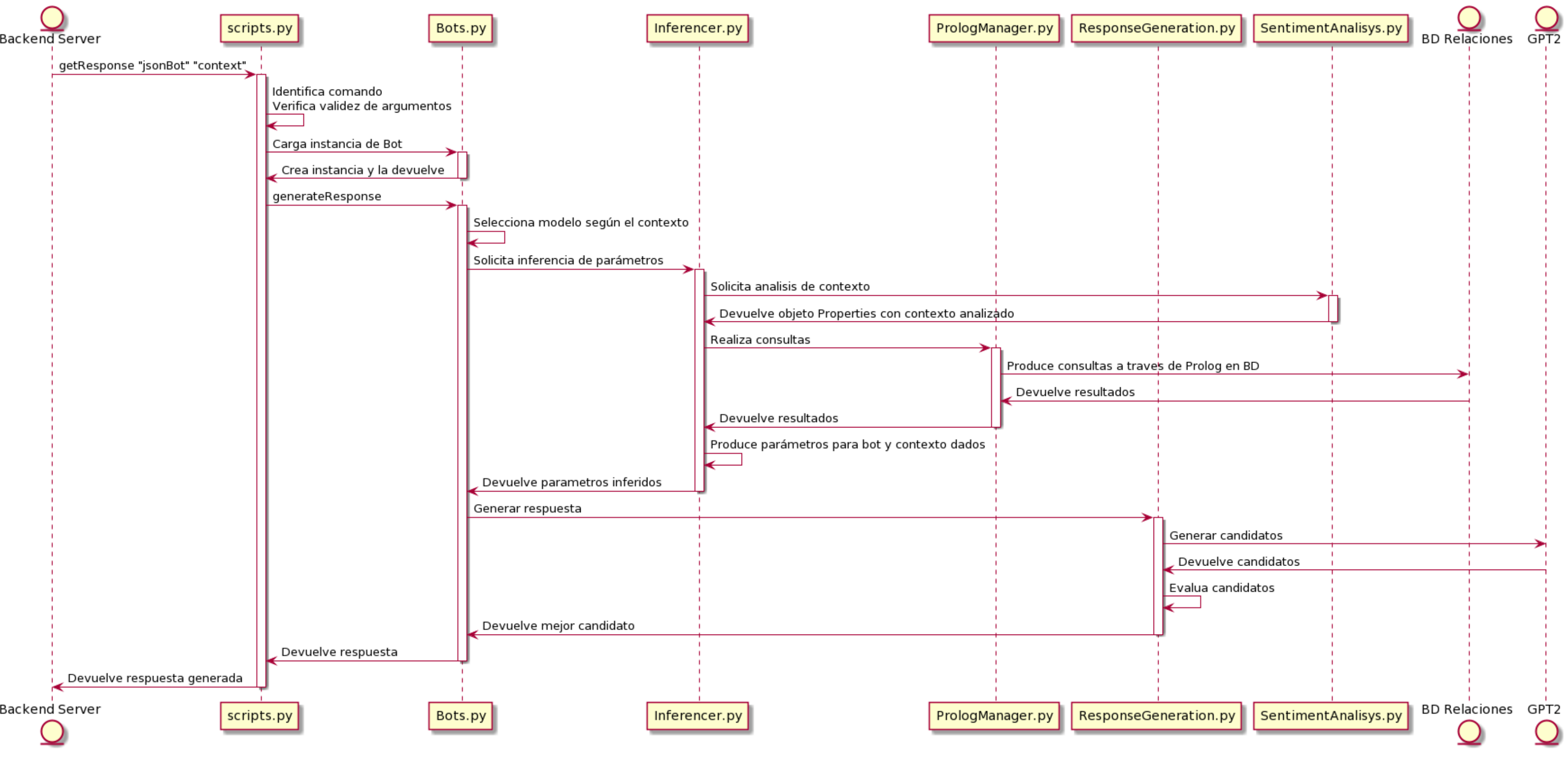


Figura B.21: *Diagrama de generación de texto en entorno de Python*

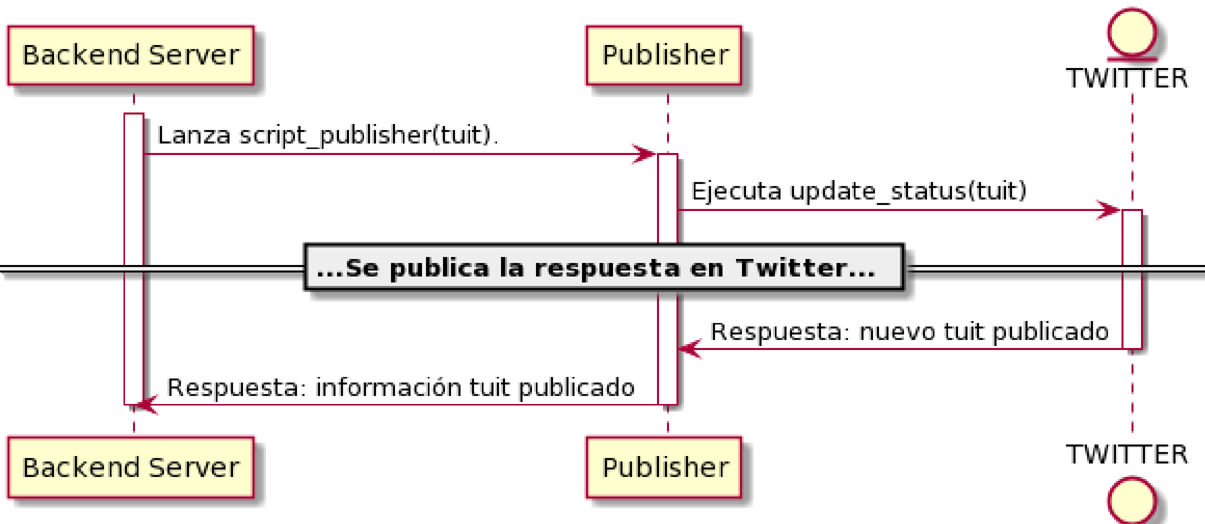


Figura B.22: *Diagrama de publicación de una respuesta en Twitter*