Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Facultad de Informática

Grado de Ingeniería Informática

· Trabajo Fin de Grado ·

Computación

Smart: detección de SPAM en comentarios realizados en la red social EkitApp

Elsa Scola Martín

Junio 2019

Director: Dr. Alfredo Goñi Sarriguren

Resumen

En este Trabajo de Fin de Grado, se desarrolla el sistema Smart, el cual proporciona un filtro de comentarios Spam en tiempo real. Smart se integra exitosamente con EkitApp, una red social de eventos universitarios fruto de la idea de un emprendizaje en colaboración con Julen Miner. Esta solución combina algunas de las técnicas más efectivas halladas hasta el momento, y proporciona un enfoque práctico y completo de lo que significa realizar un producto en el sector del Data Science, desde la recogida de datos hasta su despliegue; posibilitando su uso desde cualquier parte del mundo.

Para ello, en primer lugar, se han construido datasets con distintas características sobre los que se han aplicado múltiples algoritmos de Machine Learning. En segundo lugar, se ha seleccionado el clasificador a desplegar como función API y se ha implementado adicionalmente una versión multi-idioma de esta.

Posteriormente, se han integrado las funciones con la red social EkitApp y, por último, tras la descripción de las aportaciones más relevantes, se han propuesto vías de mejora y líneas futuras que el proyecto adoptaría una vez el emprendizaje se encontrase en una fase más avanzada.

Agradecimientos

Antes de comenzar, me gustaría agradecer la ayuda recibida por todas las personas que han formado parte de este proyecto.

En primer lugar, agradecer al director del proyecto, Alfredo Goñi, y a José Miguel Blanco, su compromiso con el proyecto al apostar por una idea propia y emprendedora como esta, y por haberme transmitido tanto conocimiento a través de sus consejos.

Por otro lado, quisiera mostrar mi gratitud hacia Julen con quien he tenido la oportunidad de trabajar desde el inicio de la carrera, tanto en Magna SIS, como en la realización de numerosos proyectos en colaboración, y finalmente, en este Trabajo de Fin de Grado. Es un gran compañero, un fiel amigo, y estoy segura de que será un gran profesional. No podría imaginar mejor forma de finalizar mis estudios en el grado que colaborando con Julen en este proyecto.

Por último, pero no por ello menos importante, me gustaría dar las gracias a mi madre por su apoyo, por enseñarme a creer en mí misma y por darme la oportunidad de estudiar, a ella le debo todos mis logros y le dedico este trabajo.

Índice

Índice de figuras

Índice de tablas

1. Introducción

En este capítulo se expone el contexto actual y se describe brevemente la idea general del proyecto, así como el desarrollo que ha sido llevado a cabo. Además, se listan e introducen brevemente los capítulos de esta memoria.

1.1 Introducción

La creciente popularidad de las redes sociales ha propiciado que éstas implementen progresivamente técnicas relacionadas con la ciencia de datos, como la de Machine Learning, con el objetivo de proporcionar una mejor experiencia de usuario. En la actualidad, las funcionalidades inteligentes se han convertido en una parte esencial dentro de las redes sociales, y esta tendencia no hará más que aumentar en los próximos años.

Las técnicas de Machine Learning tienen numerosas aplicaciones dentro de las redes sociales (marketing, experiencia de usuario...), siendo la seguridad una de las áreas con mayor potencial de mejora. Una de las principales problemáticas de seguridad dentro de las redes sociales es la detección de Spam en tiempo real. Durante los últimos años los ataques de Spam han cambiado su entorno más habitual de los correos y SMS por uno cada vez más utilizado por la gente de a pie, las redes sociales. A pesar de que numerosos estudios han tratado de solventar esta problemática, todavía no se ha alcanzado un consenso que determine la técnica más efectiva para detectar y filtrar Spam en tiempo real.

En este Trabajo de Fin de Grado (TFG), se desarrolla el sistema Smart, el cual proporciona un filtro de comentarios Spam en tiempo real. Smart se integra exitosamente con EkitApp, una red social de eventos universitarios fruto de la idea de un emprendizaje en colaboración con Julen Miner. Esta solución combina algunas de las técnicas más efectivas halladas hasta el momento, y proporciona un enfoque práctico y completo de lo que significa realizar un producto en el sector del Data Science, desde la recogida de datos hasta su despliegue; posibilitando su uso desde cualquier parte del mundo.

1.2 Contenido

Esta memoria se estructura en 13 capítulos, organizados de la siguiente manera: En el Capítulo 2, se presentan los antecedentes del proyecto. En el Capítulo 3, se exponen los objetivos y alcance del proyecto. En el Capítulo 4, se realiza un análisis de requisitos del sistema a construir y su correspondiente base de datos, así como un análisis de trabajos previos y datasets empleados en estudios de relevancia. En el Capítulo 5, se presentan una serie de experimentos a partir de los cuales se selecciona el mejor modelo y se propone su versión en castellano. En el Capítulo 6, se profundiza en el diseño de la solución y del modelo de datos. Adicionalmente se realiza una propuesta de dos funciones de las cuales una es multi-idioma. En el Capítulo 7, se explica la implementación de las funciones, así como el uso de varios servicios empleados a lo largo del proyecto. En el Capítulo 8, se muestran las pruebas unitarias de las funciones además de aquellas en las que se realiza la integración con EkitApp. En el Capítulo 9, se explica cómo desplegar Machine Learning en la nube, así como el ciclo de vida de la solución una vez EkitApp se encuentre en explotación. En el Capítulo 10, se detalla la gestión del proyecto y la colaboración, además de mostrar las dedicaciones por cada paquete de trabajo. En el Capítulo 11, se exponen las conclusiones finales del proyecto y se proponen posibles mejoras y líneas de continuación futuras. Finalmente, en los capítulos 12 y 13 se encuentran las referencias bibliográficas y anexos.

2. Antecedentes

Este capítulo de la memoria trata sobre todos aquellos aspectos que precedieron al comienzo de Smart. Se describen los factores que suscitaron el interés en el área de trabajo de este proyecto, la vocación de desarrollar una iniciativa empresarial y de colaborar, así como los antecedentes tecnológicos.

2.1 Interés del área

Mi interés en la ciencia de datos y, en especial, en el aprendizaje automático nació mientras cursaba el grado en segundo curso al descubrir que ésta es una de las formas más efectivas actualmente de dotar a una máquina de inteligencia. Durante los años sucesivos tuve la oportunidad de cursar "Minería de Datos" y "Machine Learning and Neural Networks", dos asignaturas que tuvieron un gran impacto en mi percepción sobre la ciencia de datos y las cuales sentaron las bases de mi conocimiento en esta área. Adicionalmente, me presenté por cuenta propia a la beca "Bertelsmann Data Science Scholarship"[1] de Udacity en colaboración con Bertelsmann para aprender más sobre Data Science, la cual me fue concedida.

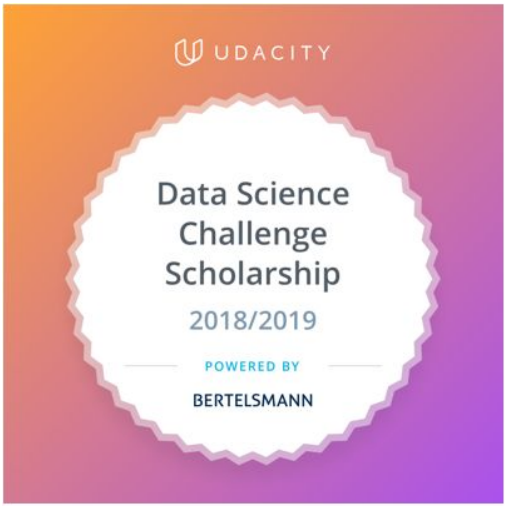

Figura 1: Sello de recepción de la beca.

Además, realicé varios proyectos de forma independiente al grado en los cuales ponía en práctica algunos conceptos aprendidos en clase, como un predictor de resultados deportivos. Todas estas actividades no hicieron más que acrecentar mi interés en la materia hasta el punto de tener la intención de orientar mis estudios de máster a esta área.

[1] Enlace a las becas: https://www.udacity.com/bertelsmann-data-scholarships

2.2 Vocación de desarrollar una iniciativa empresarial

Mi vocación por el emprendimiento y el desarrollo de una iniciativa empresarial en torno a la informática nace al finalizar el primer curso en el grado. Mi compañero Julen Miner (en lo sucesivo, Julen) y yo desarrollamos una aplicación llamada "The Word Game"[2] que finalmente publicamos en Google Play Store.

Desde entonces comprendí el enorme potencial de estudiar informática y comencé a acercarme al mundo de la empresa para así aprender a compartir los proyectos que estaba realizando. Meses más tarde, en 2017, comenzó mi labor como presidenta y socia en Magna SIS[3], la cual me dotó de una perspectiva realista acerca del mundo empresarial. Durante mi desempeño tuve la oportunidad de aprender a través de la experiencia y sentar las bases de mi carrera en este sector.

Finalmente, a falta de unos meses para finalizar mi etapa en la carrera, me dispongo a dar un paso más en colaboración con Julen, para así ver realizada nuestra idea empresarial, una red social de eventos universitarios que, esperamos, tenga un impacto positivo en la sociedad y facilite la experiencia universitaria a los estudiantes que nos sucedan.

2.3 Vocación de colaborar

La colaboración es un aspecto clave en el sector de la Informática. Todo proyecto de éxito nace de la colaboración de diversos puntos de vista. También es una virtud que conviene cultivar para construir una sociedad mejor como colectivo y no como individuos aislados.

Durante la carrera he tenido la oportunidad de trabajar con Julen desde el principio. Comenzando por nuestra primera aplicación, continuando por numerosos proyectos en colaboración, desempeñando nuestra labor en Magna SIS y, finalmente, en este proyecto de fin de grado.

El hecho de habernos especializado en distintas áreas, Ingeniería del Software y Computación, ha contribuido a nuestra ambición de colaborar, debido a que trabajando en equipo podemos aportar perspectivas y conocimiento diverso a nuestro proyecto común, para así llevar a cabo un producto completo y con valor añadido.

Nuestra iniciativa por ir más allá de lo aprendido en clase y nuestro objetivo común dentro del emprendizaje nos ha unido haciendo posible la realización de esta idea.

[2] Enlace al blog de la aplicación: https://ellenco.tumblr.com/
[3] Enlace al sitio web de Magna SIS: https://magnasis.com/

2.4 Antecedentes tecnológicos

Al finalizar el primer año en el grado, completé el curso "Desarrollo Android para Principiantes"[4], disponible gratuitamente en Udacity y, más adelante, se me concedió la beca "Google Developer Scholarship"[5] de Udacity en colaboración con Google. Los conocimientos adquiridos en ese tiempo han contribuido a mi comprensión del desarrollo de Ekitaldi, para así mejorar nuestra colaboración.

Durante el grado, he tenido la oportunidad de aprender a utilizar herramientas como "Weka" y el lenguaje de programación Python en combinación con numerosas librerías, como "scikit-learn"[6] y "Pandas", para la realización de análisis de datos y predicciones. En este proyecto me he decantado por utilizar Python, debido a la disponibilidad de librerías, su soporte y numerosos métodos de aprendizaje automático. Además, Python proporciona marcos de aprendizaje profundos, como "TensorFlow"[7], que son los enfoques actuales más avanzados.

Desde el comienzo de este proyecto, optamos por trabajar con las tecnologías de Google en nuestro proyecto, debido a que son vanguardistas y disponen de un gran soporte. Además, Google ofrece herramientas para las dos áreas de trabajo de este proyecto, la del desarrollo de software y la ciencia de datos, por lo que utilizar herramientas de un mismo ecosistema facilita el flujo de trabajo y hace que las distintas partes del proyecto se integren eficazmente.

Una de las herramientas de las que hago uso es Google Datalab, la cual he aprendido a utilizar con facilidad debido a que he cursado la asignatura de "Machine Learning and Neural Networks", donde se enseña a trabajar con Jupyter Notebook[8].

En cuanto a la base de datos, sopesamos si utilizar Realtime Database, una base de datos en la nube con algo más de recorrido, o Cloud Firestore, que presenta prestaciones mejoradas como la velocidad de respuesta y la escalabilidad. Finalmente escogimos esta última ya que nos permitiría utilizar la tecnología más vanguardista del momento.

[4] Enlace al curso: https://eu.udacity.com/course/android-development-for-beginners--ud837
[5] Enlace a las becas: https://www.udacity.com/google-scholarships
[6] Enlace al sitio web de scikit-learn: https://scikit-learn.org/stable/
[7] Enlace al sitio web de TensorFlow: https://www.tensorflow.org/
[8] Jupyter Notebook es un entorno de trabajo interactivo que permite desarrollar código en Python.

3. Objetivos

Definir los objetivos es parte crucial de todo proyecto, debido a que permite definir y acotar la dirección que éste toma, al igual que sus limitaciones. En este capítulo se comentan los objetivos generales de Smart-Ekitaldi, así como los específicos de Smart y el alcance que tendrá. Además se mencionan las exclusiones y riesgos de este proyecto.

3.1 Objetivos generales

En la actualidad, la comunicación de eventos en el entorno universitario se realiza a través de correos electrónicos habitualmente ignorados por el alumnado. El objetivo de este proyecto es crear una red social completa y segura, que sirva como comunidad en la que estudiantes y profesores puedan hacer públicas estas actividades. De este modo, se crea un núcleo que propicia la iniciativa del alumnado y facilita a las instituciones universitarias la comunicación de eventos oficiales.

En el mercado actual toda red social ofrece de base una aplicación y un sistema inteligente que lo respalda, por lo que la colaboración de miembros de distintas áreas en este tipo de proyectos es fundamental. Por esta razón, es imprescindible sentar unas bases sólidas de colaboración entre las dos partes de los Trabajos de Fin de Grado para que la cohesión de éstas tenga sentido.

Por consiguiente, para la realización exitosa de este proyecto, es de gran importancia la alta integración de todos los servicios que ofrece Google; desde el desarrollo del software para la implementación de una aplicación, hasta los servicios de Machine Learning para dotar al producto de Artificial Intelligence.

Asimismo, se busca dar comienzo a un producto que en el futuro pueda servir como base para la realización de un emprendizaje, buscando así un proyecto que sea de utilidad y formativo al mismo tiempo.

3.2 Objetivos específicos Smart

Durante los últimos años, numerosas redes sociales han comenzado a implementar técnicas relacionadas con la analítica de datos, como la de Machine Learning, para resolver problemas dentro de ésta o proporcionar una mejor experiencia de usuario. En la actualidad, se ha convertido en una parte esencial dentro de las redes sociales, y esta tendencia no hará más que aumentar en los próximos años.

Smart pretende complementar a Ekitaldi proporcionándole funciones adicionales relacionadas con la analítica de datos. Estas funciones tienen como objetivo dotar de inteligencia a la red social EkitApp.

Se trata de desarrollar una *caja de herramientas* que proporcione servicios relacionados con la experiencia de usuario (ej. sistema de recomendaciones) y seguridad (ej. detección de contenido inapropiado) entre otros, con ayuda de la analítica de datos, de forma que puedan integrarse fácilmente con Ekitaldi.

3.3 Descripción del alcance

Debido a las limitaciones de tiempo y recursos de un Trabajo de Fin de Grado, el alcance del proyecto es el siguiente:

Desarrollar un filtro de comentarios Spam funcional que se integre con la aplicación Ekitaldi. A través de dicho filtro, se pretende aumentar la seguridad en la red social y solventar uno de los principales riesgos para los usuarios dentro de ésta.

Experimentar con todo el proceso de creación de un producto de analítica de datos. En este proceso se incluyen: estudio de trabajos anteriores, la recogida de datos, realización de pruebas, selección de modelo y despliegue de la solución. El trabajar con un ciclo completo de creación de producto permitirá tener una perspectiva realista de las aplicaciones de la analítica de datos que va más allá del entorno académico.

A continuación se describen los paquetes de trabajo soportados por la Estructura de Descomposición de Trabajo (EDT) de la Figura 2:

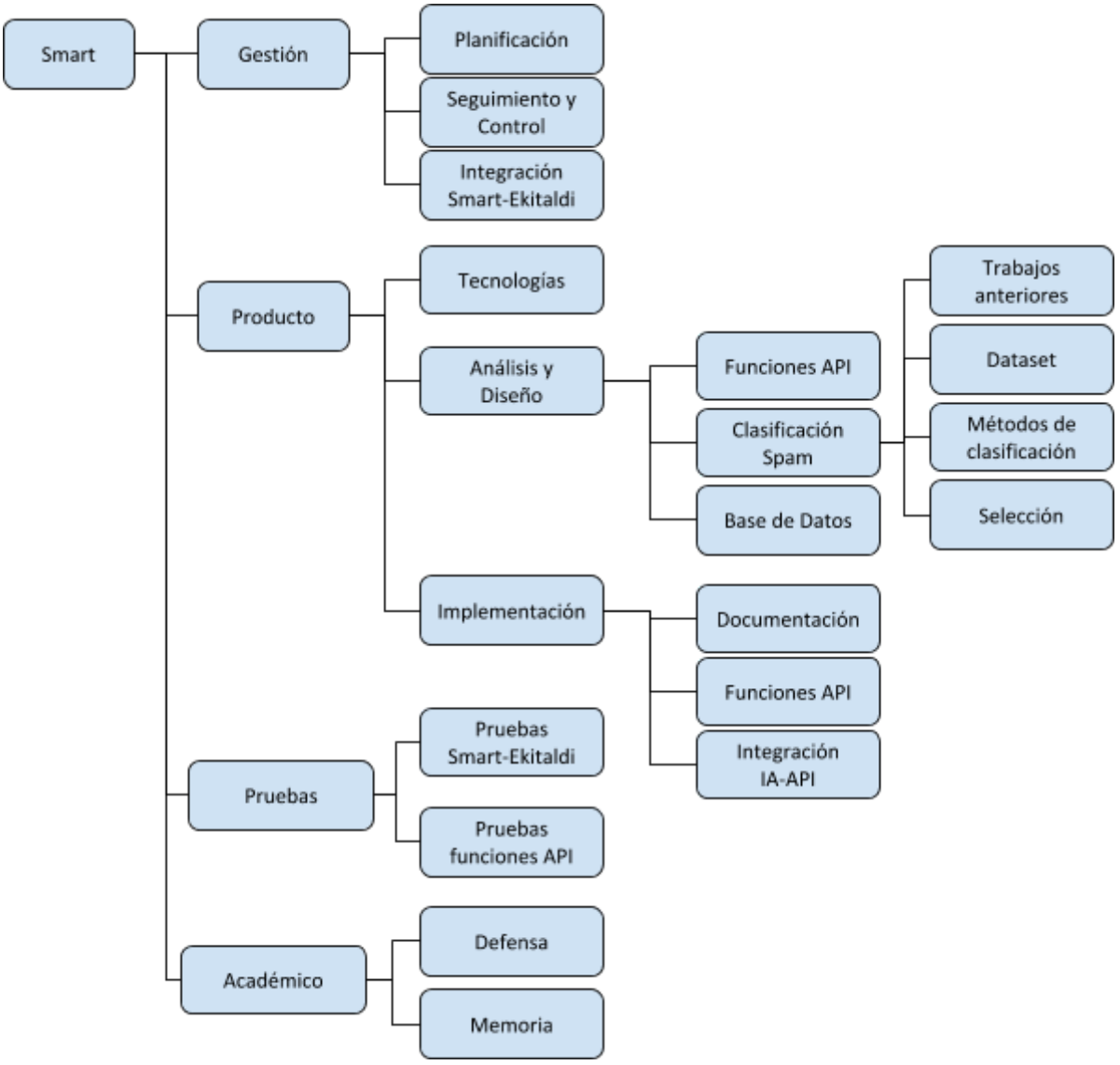

Figura 2: Estructura de Descomposición de Trabajo.

Smart: es la raíz del EDT y engloba el TFG presentado en esta memoria.

- Gestión: contiene todo lo relacionado con la gestión del proyecto Smart.
    - Planificación: en este bloque se agrupan todas aquellas tareas orientadas a establecer el plan conforme al que se ha de desarrollar Smart.
    - Seguimiento y control: engloba el conjunto de acciones para la comprobación de la correcta ejecución de las actividades de Smart.
    - Integración Smart-Ekitaldi: conforma todos aquellos acuerdos alcanzados entre la parte Smart del proyecto y la parte Ekitaldi.
- Producto: contiene todas las actividades referentes al producto de Smart.
    - Tecnologías: en este bloque se encuentran todas las tareas relacionadas con el aprendizaje de nuevas tecnologías, incluyendo documentación sobre éstas.

    - Análisis y Diseño: en este bloque se encuentran todas las tareas relacionadas con el análisis y diseño del producto.
        - Funciones API: en este paquete se encuentran las actividades relacionadas con la selección de la función que debería implementarse (filtrado de Spam, sistema de recomendaciones,...), además de los requisitos de ésta. En este paquete también se incluyen los diseños de las funciones a implementar.
        - Clasificación Spam: en este bloque se encuentran todas las tareas relacionadas con el estudio de trabajos anteriores, la recogida de datos,y selección de modelo.
            - Trabajos anteriores: tareas relacionadas con el estudio de trabajos de relevancia e interés sobre el filtrado de Spam en redes sociales.
            - Dataset: se incluyen todas las tareas relacionadas con la construcción de los datasets a utilizar en cada una de las implementaciones de las distintas soluciones a experimentar.
            - Métodos de clasificación: incluye las tareas relacionadas con la implementación de los modelos en base a la labor de investigación realizada en el paquete *Trabajos anteriores*.
            - Selección: análisis y extracción de conclusiones a partir de los resultados obtenidos por los distintos modelos para la selección de aquel que se empleará en las funciones API.
        - Base de Datos: agrupa las tareas de análisis y diseño de la base de datos. Estas tareas las realizan Smart y Ekitaldi de forma conjunta.
    - Implementación: agrupa las tareas orientadas a la implementación de las funciones.
        - Documentación: engloba las tareas de documentación, en la cual se describe la lógica de las funciones API.
        - Funciones API: agrupa las tareas orientadas a la implementación del código de las funciones de la API.
        - Integración IA-API: engloba las actividades orientadas a la integración del modelo seleccionado y la función API.
- Pruebas: incluye las tareas relacionadas con las pruebas unitarias y de integración.
    - Pruebas funciones API: agrupa las tareas relacionadas con probar que las funciones API se comportan de forma correcta. Nota: en estas pruebas no se mide el rendimiento de los algoritmos IA, eso se realiza en el paquete Producto>Análisis y Diseño>Métodos de clasificación.
    - Pruebas Smart-Ekitaldi: en este paquete se encuentran las tareas destinadas a la realización de pruebas conjuntas de Smart y Ekitaldi.
- Académico: se encuentran las actividades destinadas a la elaboración de los entregables adicionales al producto.
    - Defensa: tareas relacionadas con la defensa de Smart ante el tribunal.
    - Memoria: tareas orientadas a la redacción de la memoria.

3.4 Exclusiones

Debido a las limitaciones de tiempo y recursos de un Trabajo de Fin de Grado, se listan a continuación los puntos que se excluyen del alcance en el proyecto actual:

- No se estudiarán ni aplicarán medidas de seguridad respecto a las funciones.
- No se realizará ningún análisis ni se aplicará medida de protección alguna de acuerdo con el Reglamento General de Protección de Datos.

4. Análisis

El análisis es de gran importancia durante un proyecto, ya que permite reconocer requisitos y orientar el proyecto partiendo de una base de estudio. En este capítulo se tratan todos aquellos aspectos relacionados con el análisis de los requisitos de Smart, la base de datos de Smart-Ekitaldi y trabajos previos relacionados con la detección de Spam en redes sociales.

4.1 Requisitos no funcionales

- Desarrollar una funcionalidad que se integre de forma sencilla con la app Ekitaldi.
- El desarrollo de nuestras soluciones está basado en tecnologías del ecosistema Google, por lo que la funcionalidad debe emplear los siguientes servicios proporcionados por dicha entidad:
    - Datalab para el entrenamiento de modelos.
    - Firebase para almacenar el modelo y desplegar la función como API.
    - ML Engine para hacer accesible el modelo.
- El servicio de la base de datos seleccionado es Firestore.
- El diseño de la estructura de la base de datos de la red social debe ser realizado de forma conjunta y servirá como base para ambos proyectos.
- Realizar un análisis de estudios que propongan soluciones en ámbito de la clasificación de Spam.
- Debido a que EkitApp no está operativo inicialmente, habrá que construir un dataset que simule los datos que se generarán en un futuro por la red social cuando esté en explotación. Este dataset se utilizará para entrenar y probar el modelo inicial.

4.2 Requisitos funcionales

- Desarrollar una función accesible como API.
- La funcionalidad de la función es detectar Spam.
- La función debe poder integrarse con la aplicación Ekitaldi.
- La función debe detectar los comentarios Spam que publiquen los usuarios en tiempo real.
- Realizar un modelo para la clasificación de Spam.
- El modelo será accesible a través de Cloud Functions.

- La función devolverá *True* si se trata de Spam y devolverá *False* en caso de no ser Spam.
- La función estará orientada a detectar principalmente Spam en inglés.

4.3 Sistema a construir

A continuación se muestra un diagrama simplificado para simular el comportamiento del sistema a construir:

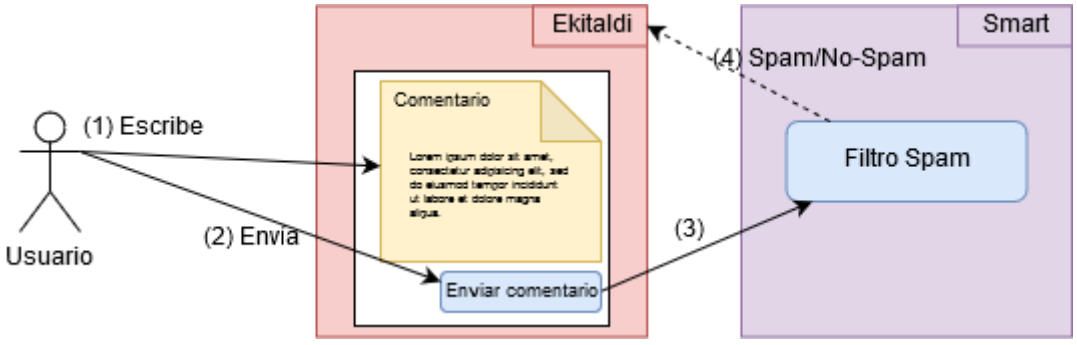

Figura 3: Diagrama simulación comportamiento del sistema.

El usuario escribe un comentario dentro de la aplicación desarrollada en Ekitaldi y pulsa el botón de *Enviar*, en ese momento se llama a la función Smart *Filtro Spam*. La función devuelve a Ekitaldi True o False dependiendo de si se ha detectado que el comentario es Spam o no.

4.4 Base de datos

Las bases de datos no relacionales, NoSQL, son frecuentemente utilizadas en proyectos donde se manejan grandes volúmenes de datos, ya que habitualmente son fácilmente escalables y además soportan procesamiento paralelo de datos de forma simultánea (Lee[8]). Existen numerosos tipos de bases de datos NoSQL que difieren en el sistema de almacenamiento de los datos. En nuestro caso empleamos un sistema de base de datos documental, debido a que Firestore se basa en éste. Este sistema se caracteriza por la estructuración de los datos, "a partir del modelo de datos NoSQL de Cloud Firestore, almacenas los datos en documentos que contienen campos que se asignan a valores. Estos documentos se almacenan en colecciones, que son contenedores para tus documentos y que puedes usar para organizar tus datos y compilar consultas"[9].

Al realizar consultas en una base de datos NoSQL documental, cuando se necesita un dato, el sistema carga el documento completo, es decir, se realiza la transferencia completa aunque no se necesite el resto de datos. Por esta razón, y para mejorar la eficiencia de las consultas, datos de diferentes documentos suelen estar duplicados, de este modo se evita tener que transferir varios

[9] Documentación oficial de Firestore: https://firebase.google.com/docs/firestore#how_does_it_work

documentos. Es posible que las características ACID[10] no se mantengan debido a la duplicación de datos, por lo que el programador debe asegurarlo al modificar la base de datos.

4.5 Trabajos previos y datasets

Respecto al filtrado de Spam en redes sociales, hay numerosos trabajos e investigaciones que se han llevado a cabo para tratar de solventar este problema que persiste actualmente. Algunas de las soluciones estudiadas son aplicables a nuestra red social, debido a las similitudes existentes entre EkitApp y redes sociales que se encuentran actualmente en el mercado, como Twitter y YouTube.

Hay varios artículos que proponen utilizar como *features* información del usuario combinada con características del comentario Chen[1], Gupta[2] y hay otros que optan por emplear únicamente el texto del comentario Ezpeleta[3]. También existen propuestas basadas en los vecinos del usuario en la red social Yang[4], sin embargo, consumen mucho tiempo y recursos a medida que la red social va creciendo por lo que no son adecuados para la detección en tiempo real.

Mi propuesta intenta abordar el problema mediante la combinación de técnicas empleadas en distintos estudios con el objetivo de obtener unos resultados mejorados. Dicha propuesta se describe en profundidad en el capítulo 5.

A continuación se mencionan 3 trabajos previos relacionados con la detección de Spam en redes sociales. Se trata de 3 artículos, de los cuales se comentan algunas ideas relevantes y los datasets empleados.

4.5.1 6 Million Spam Tweets: A Large Ground Truth for Timely Twitter Spam Detection

Chen[1] pone a disposición 4 datasets:

Dataset	Sampling Method	NO. of Spam Tweets	NO. of Non-spam Tweets
I	Continuous	5000	5000
II	Continuous	5000	95000
III	Non-continuous	5000	5000
IV	Non-continuous	5000	95000

Tabla 1: Datasets empleados en Chen[1].

2 datasets de 100.000 tuplas (cada tupla representa un tweet) y 2 de 10.000. Por cada par de datasets hay uno en el que los tweets están ordenados de forma continua en el tiempo, y otro con los tweets en un orden aleatorio (sin orden).

[10] Acrónimo de Atomicity, Consistency, Isolation and Durability: Atomicidad, Consistencia, Aislamiento y Durabilidad en español.

Los 4 datasets siguen la siguiente estructura:

Feature Name	Description
account_age	The age (days) of an account since its creation until the time of sending the most recent tweet
no_follower	The number of followers of this twitter user
no_following	The number of followings/friends of this twitter user
no_userfavourites	The number of favourites this twitter user received
no_lists	The number of lists this twitter user added
no_tweets	The number of tweets this twitter user sent
no_retweets	The number of retweets this tweet
no_hashtag	The number of hashtags included in this tweet
no_usermention	The number of user mentions included in this tweet
no_urls	The number of URLs included in this tweet
no_char	The number of characters in this tweet
no_digits	The number of digits in this tweet

Tabla 2: Features de los datasets empleados en Chen[1].

Para la clasificación de Spam, se emplean los siguientes clasificadores: Random Forest, Decision Tree, Bayes Network, Naive Bayes, K-NN y SVM.

A continuación, se muestran los resultados obtenidos empleando el dataset II (a rayas) y IV (color oscuro):

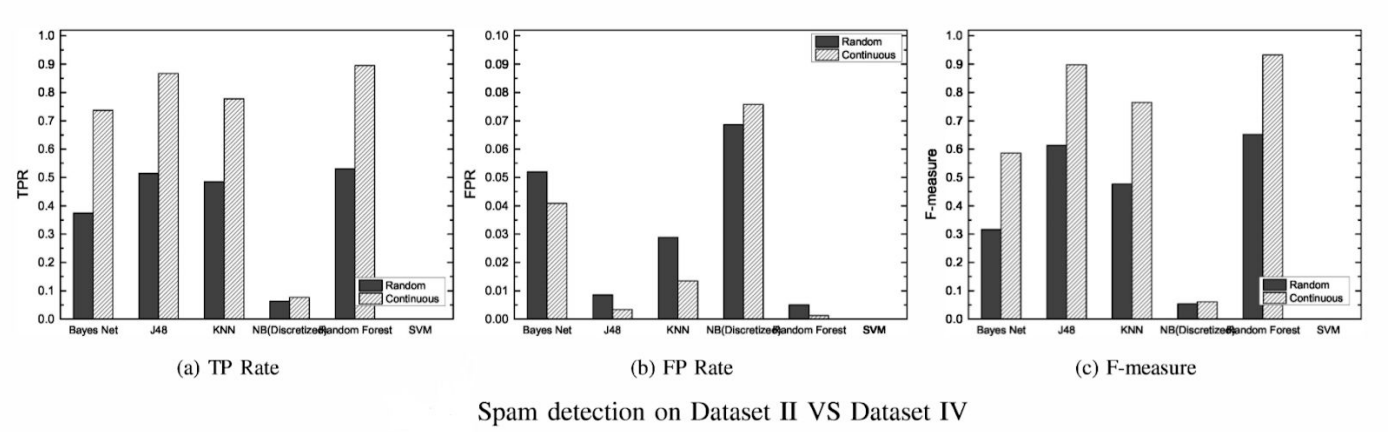

Figura 4: Detección de Spam en Chen[1] empleando Dataset II y Dataset IV.

Podemos apreciar que en el caso del dataset IV, se obtienen los mejores resultados con Random Forest y Decision Tree.

Entre las ideas mencionadas en este estudio, una de las más destacadas con relación a mi proyecto es la proporción de tweets Spam dentro de la red social Twitter, que se posiciona en aproximadamente un 5% Grier[6]. Por esta razón, no es representativo de un entorno real dentro de las redes sociales, el emplear un dataset con el mismo número de publicaciones Spam que no Spam.

Los *features* basados en grafos sociales discriminan mejor entre publicaciones Spam y no Spam, sin embargo, se incide en la inviabilidad de extraer *features* basados en grafos sociales, debido al alto coste de recoger esos *features* y que no pueden ser calculados hasta que el grafo social esté formado.

Es significativo detectar el Spam lo antes posible, por lo que se concentran en extraer *features* de "poco peso" que pueden ser empleados en tiempo real.

4.5.2 A framework for real-time spam detection in Twitter

Gupta[2] no pone a disposición su dataset. Han empleado el dataset *HSpam14 Dataset*[11], sin embargo este no dispone de todas las propiedades que han extraído durante el estudio (como los textos de los tweets).

Por otro lado, realizan algunas aportaciones respecto a Chen[1]:

Se añade el número de caracteres no-ASCII como *feature* a añadir a la propuesta del Chen[1]. Por lo que quedaría un total de 13 *features*.

Feature Name	Description
account age	The age (days) of an accounl since its creation until the time of sending the most recent tweet
no_follower	The number of followers of this Twitter user
no_following	The number of followings/friends of this Twitter user
no_userfavourites	The number of favourites this Twitter user received
no_lists	The number of lists this Twitter user added
no_tweets	The number of tweets this Twitter user sent
no_retweets	The number of retweets this tweet
no_hashtag	The number of hashtags included in this tweet
no_usermention	The number of user mentions included in this tweet
no_urls	The number of URLs included in this tweet
no_char	The number of characters in this tweet
no_digits	The number of digits in this tweet
no_non-ASCII_characters	The number of non-ASCII characters in this tweet

Tabla 3: Features empleados en Gupta[2].

Además, durante este estudio, se han analizado las palabras que aparecen con más frecuencia únicamente en tweets Spam y únicamente en tweets no-Spam. Tras extraer la ganancia de información (*Information Gain*) del modelo *Bag-of-Word*, se hallan las 15 palabras que aparecen con más frecuencia en tweets Spam y lo mismo para tweets no-Spam. Esto es un total de 30 palabras que pueden emplearse como *features*. Con lo cual, el conjunto de *features* ascendería a 43 *features* (13 + 30). Sin embargo, los spammers pueden cambiar su comportamiento con el tiempo, por lo que en el mundo real, los features siguen cambiando de forma inesperada y habría que re-calcularlos periódicamente, a este fenómeno, se le llama ”Spam Drift”.

Se emplean los siguientes clasificadores: SVM with Kernel, Neural Network, Gradient Boosting y Random Forest.

A continuación, se muestran los resultados obtenidos:

Unit %	**Feature-set- 1**	
Classifier	F-Measure	Accuracy
SVM with Kernel	86.18	85.95
Neural Network	90.56	91.65
Gradient Boosting	75.81	85.84
Random Forest	75.39	86.25

Tabla 4: Resultados obtenidos en Gupta[2].

[11] Enlace a HSpam14 Dataset: http://www.ntu.edu.sg/home/axsun/datasets.html

4.5.3 A Mood Analysis on Youtube Comments and a Method for Improved Social Spam Detection

Ezpeleta[3] emplea dos datasets disponibles públicamente:

- *Youtube Comments Dataset*[12] que dispone de 6.431.471 comentarios de Youtube de los cuales 481.334 son Spam y han sido publicados por un total de 2.860.264 usuarios.

  La estructura de los datos es la siguiente:

  - *published* - milliseconds since Unix epoch
  - *videoId* - anonymized
  - *userId* - anonymized
  - *commentText* - line break characters have been replaced with '\n' and '\r' as required. User ids have been anonymized.
  - *spam* - *True*|*False*, if the comment was marked as spam or not.

  Figura 5: Estructura Youtube Comments Dataset.

- *YouTube Spam Collection Data Set*[13] que dispone de 1.956 mensajes, de los cuales 1.005 son Spam. Este dataset únicamente se empleó para validar los resultados obtenidos con el dataset previo.

Para los experimentos se toma una porción de los datos del primero de los dos datasets recién mostrados, obteniendo así una un subconjunto de datos con 1.000 comentarios Spam y 3.000 no Spam.

Lo que se propone en este estudio, es demostrar que la clasificación de Spam mejora cuando se tiene en cuenta el humor (*mood* en inglés) asociado al texto. Para ello, se utilizan algunos clasificadores con el dataset *Youtube Comments Dataset* y se obtienen unos resultados. Tras ello, se extrae el "humor" de los comentarios y se añade como *feature* adicional y vuelven a aplicarse los mismos clasificadores, obteniéndose así mejores resultados que la primera vez. Para la extracción del "humor" del texto, se emplea una herramienta disponible de forma pública de uClassify[14] que tiene una precisión del 96%.

[12] Enlace a Youtube Comments Dataset: http://mlg.ucd.ie/yt/
[13] Enlace a YouTube Spam Collection Data Set: https://archive.ics.uci.edu/ml/datasets/YouTube+Spam+Collection
[14] Sitio web uClassify: https://www.uclassify.com/

A continuación se muestran los clasificadores empleados y los resultados obtenidos con y sin "humor":

	Normal		*Mood*	
Name	**FP**	**Acc**	**FP**	**Acc**
NBM.c.stwv.go.ngtok	89	82.50	**77**	**82.53**
NBMU.c.stwv.go.ngtok	89	82.50	**70**	**82.58**
NBM.stwv.go.ngtok	71	**82.48**	**63**	82.43
NBMU.stwv.go.ngtok	71	**82.48**	**59**	82.43
NBM.c.stwv.go.ngtok.stemmer	81	82.45	**73**	**82.45**
NBMU.c.stwv.go.ngtok.stemmer	81	82.45	**68**	**82.48**
NBM.stwv.go.ngtok.stemmer	64	82.35	**58**	**82.38**
NBMU.stwv.go.ngtok.stemmer	64	82.35	**53**	**82.35**
CNB.stwv.go.ngtok	125	82.30	**110**	**82.43**
CNB.stwv.go.ngtok.stemmer	109	**82.28**	**98**	82.20

Tabla 5: Resultados obtenidos en Ezpeleta[3].

Nomenclatures

	Meaning
CNB	Complement Naive Bayes
NBM	Naive Bayes Multinomial
NBMU	Naive Bayes Multinomial Updatable
.c	idft F, tft F, outwc T [a]
.i.c	idft T, tft F, outwc T [a]
.i.t.c	idft T, tft T, outwc T [a]

	Meaning
.stwv	String to Word Vector
.go	General options
.wtok	Word Tokenizer
.ngtok	NGram Tokenizer 1-3
.stemmer	Stemmer
.igain	Attribute selection using InfoGainAttributeEval

Tabla 6: Nomenclaturas empleadas en Ezpeleta[3].

En el artículo actual, Ezpeleta et al.[3] se basan en una de las técnicas utilizadas en Ezpeleta[5], donde los autores prueban las técnicas basadas en el contenido pueden mejorar los resultados en el filtrado de Spam. Ezpeleta[5] realizan análisis de sentimientos para enriquecer el dataset original y obtienen mejoría en el *accuracy* y *False Positives*.

En Ezpeleta[3] el humor de cada mensaje se añade al dataset original para crear un segundo dataset con un *feature* adicional.

Para la validación de los resultados obtenidos se emplea la técnica *10-fold cross-validation* y se analizan en términos de falsos positivos y accuracy.

5. Clasificación de comentarios en redes sociales

En el capítulo actual se profundiza en la clasificación de Spam en redes sociales, exponiendo el planteamiento seguido, continuando por la construcción de los datasets, pasando por las técnicas empleadas para la creación de los clasificadores y seleccionando uno. Adicionalmente se expone cómo puede extrapolarse el clasificador a distintos idiomas. Este capítulo constituye la aportación que realizo en el área de la clasificación de Spam en redes sociales. En base al mismo, posteriormente diseñaré, implementaré y desplegaré el modelo escogido.

5.1 Introducción

Partiendo de los trabajos de investigación analizados durante este proyecto, se han realizado una serie experimentos cuyos resultados se comparan con los obtenidos en dichos estudios para la selección de un clasificador a emplear como filtro de Spam.

En ocasiones estos experimentos implican la obtención de distintos datasets, a menudo los cuales necesitan construirse. Una vez se dispone del dataset se realizan pruebas con distintos algoritmos para clasificación y finalmente se selecciona aquel modelo que tenga un mejor rendimiento para ser integrado en la Cloud Function.

Adicionalmente, debido a que las pruebas se han realizado únicamente para la obtención de un filtro de Spam en inglés, una vez se seleccione el modelo a implementar en la Cloud Function se realizará un experimento adicional en el que se emplea el mismo algoritmo del modelo seleccionado pero con el objetivo de clasificar Spam en castellano.

5.2 Construcción del dataset

Tal y como se ha mencionado en la introducción, para la realización de algunos de los experimentos se han tenido que emplear distintos datasets, ya que los *features* han ido variando. A continuación se listan los datasets empleados durante los experimentos y se describe su obtención:

100K

Este dataset corresponde al dataset IV empleado en Chen[1] y el cual puede obtenerse a través del sitio oficial[15]. Se habla de su estructura en el apartado 4.5.1 de Análisis.

100K + Comment

Para la detección efectiva de Spam dentro de EkitApp el objetivo es crear datos que sean similares a los que va a generar la aplicación una vez esté en explotación. Para lograr esto se opta por generar un nuevo dataset a partir de los datos contenidos en dos datasets:

Por un lado, disponemos del dataset *100K* mencionado anteriormente, en el cual cada tupla de este dataset representa un tweet, sin embargo, para la creación de una base de datos más completa, se supondrá que cada una de éstas representa un usuario dentro de EkitApp.

En EkitApp se prevé que se dispondrá de todos los datos del dataset *100K*, excepto *no_userfavourites*, *no_lists*, *no_hashtag*, *no_usermention* y *no_retweets*. *no_urls* tampoco va a utilizarse, ya que hace referencia a un tweet concreto y se va a suponer que cada tupla de esta base de datos no es un tweet, sino un usuario.

Además, para adaptar los datos a lo que generará en un futuro EkitApp, *no_tweets* pasa de representar el número de tweet de un usuario a representar el número de eventos publicados por un usuario.

Por otro lado, se dispone del dataset *Youtube Comments Dataset*[16] empleado en Ezpeleta[3], del cual se habla de su estructura en el apartado 4.5.3 de Análisis. Estos datos simulan los comentarios de los usuarios de EkitApp y se combinan con el dataset *100K*. *Youtube Comments Dataset* dispone de comentarios en numerosos idiomas, por lo que primero se ha realizado un preprocesado empleando Cloud Translation API[17] de Google (puede aprenderse más sobre su uso en el apartado 7.2.3 de Implementación) para detectar el idioma de los comentarios y se han extraído 280.779 comentarios en inglés y 9.433 en castellano. Llamaremos a estos datasets resultantes *YCD_EN* y *YCD_ES* respectivamente.

El siguiente paso es combinar los dataset *YCD_EN* y *100k*, para obtener un dataset más completo que represente datos similares a los que se generarán en EkitApp una vez la aplicación esté en explotación.

Tal y como se ha comentado anteriormente, *YCD_EN* dispone de 280.779 comentarios (23.986 Spam y 256.793 no-Spam), mientras que el dataset *100K* contiene 100.000 usuarios (5.000 Spammer y 95.000 no-Spammer). Debido a que hay un mayor número de comentarios que de usuarios debemos asignar varios comentarios a cada usuario, sin embargo, para que esta unión de datasets sea lo más realista posible, si más del 50% de comentarios de un usuario de *YCD_EN* son Spam, a éste se le asignará un perfil de Spammer del dataset *100K*. Es decir, habrá usuarios que publiquen Spam aunque no tengan un perfil de Spammer.

Para obtener los porcentajes que comentábamos anteriormente, es necesario realizar una BD de usuarios de *YCD_EN* en la que posteriormente puedan verse esas proporciones de los comentarios

[15] Enlace al dataset IV de Chen[1]: http://nsclab.org/nsclab/resources/
[16] Enlace a *Youtube Comments Dataset*: http://mlg.ucd.ie/yt/
[17] Cloud Translation API: https://cloud.google.com/translate/?hl=es

realizados. Por ello, se crea una BD en Firestore, en la que se insertan los usuarios de *YCD_EN* y el número de publicaciones Spam y no-Spam realizadas por cada uno de éstos. Una vez se ha construido dicha BD de usuarios, se le añade el campo `'spammer'` que tiene el valor `True` si en dicho usuario de *YCD_EN* más del 50% de comentarios son Spam y viceversa. Finalmente, a aquellos usuarios que tienen `'spammer'= True` se les asigna un perfil Spammer del dataset *100K* y viceversa.

100K + Comment + Sent

Este dataset es el dataset *100K + Comment*, con un *feature* adicional, el sentimiento de cada comentario. Para hallar el sentimiento se emplea un modelo para el análisis de sentimiento llamado VADER (Hutto[7]), se explica en profundidad el funcionamiento de la versión para Python[18] en el apartado 7.2.2 de Implementación.

YouTube Spam Collection Data Set

Youtube Spam Collection Data Set dispone de 1.956 mensajes, de los cuales 1.005 son Spam. el dataset actual se empleó para la validación de resultados en Ezpeleta[3]. En el apartado 5.3 se expone su uso en este proyecto.

100K + Comment + Sent + Custom

Este dataset es el *100K + Comment + Sent*, con un *feature* adicional. Este nuevo *feature* representa la predicción (True/False) de un modelo para la clasificación de Spam customizado, del cual se profundizará más en el apartado 5.3.

100K + Comment + Sent + Custom + FastText

Este dataset es el dataset *100K + Comment + Sent + Custom*, con un *feature* adicional que representa la predicción (True/False) de un modelo para la clasificación de Spam creado con la librería FastText[19]. Se habla en más profundidad de este planteamiento en el apartado 5.3.

100K + Comment + Sent + Plino

Este dataset es el dataset *100K + Comment + Sent*, con un *feature* adicional. Este *feature* adicional representa la predicción (True/False) de Plino[20], una API para la clasificación de Spam. Se habla en más profundidad de esta API en el apartado 5.3.

Comment + Sent + Custom

Este es el dataset *100K + Comment + Sent + Custom*, quitando los *features* de *100K*. Se profundiza más en la intención detrás de este experimento en el apartado 5.3.

[18] Enlace a GitHub de VADER: https://github.com/cjhutto/vaderSentiment
[19] Enlace sitio web FastText: https://fasttext.cc/
[20] Enlace sitio web Plino: https://plino.herokuapp.com/

5.3 Técnicas de clasificación de Spam

En este apartado se profundiza en los experimentos realizados y la lógica detrás de estos. Para ello, al igual que en la sección anterior, se agrupan los experimentos por datasets para facilitar la comprensión de éstos al lector.

100K

El experimento realizado con el dataset *100K* tiene como objetivo evaluar el rendimiento de modelos implementados desde cero en Python. Para evaluar su rendimiento, es necesario realizar estas pruebas en un contexto equiparable, por lo que se emplea el dataset IV de aquellos propuestos en Chen[1] y se implementan varios modelos empleando los mismos algoritmos propuestos en dicho estudio. De esta forma, pueden compararse los resultados obtenidos para concluir si las implementaciones de este proyecto tienen un rendimiento lo suficientemente aceptable y partir de esa misma base para la realización de otros experimentos.

Los algoritmos evaluados son los siguientes: Decision Tree, KNN, Naive Bayes, Random Forest.

A continuación se listan los *features* empleados (en este caso, los mismos que en Chen[1]): `"account_age","no_follower","no_following","no_userfavourites","no_lists","no_tweets","no_retweets","no_hashtag","no_usermention","no_urls","no_char","no_digits"`.

100K + Comment

La realización de un clasificador de Spam orientado a integrarse con EkitApp demandaba de un dataset que simulara los datos que podría generar dicha aplicación una vez estuviera en explotación. Por esta razón se ha creado este dataset que sirve como base para ser completado con más *features*.

Tal y como se explica en la sección anterior, algunos de los *features* se descartan debido a que no coinciden con el contenido que produce la aplicación EkitApp.

Además, aunque este dataset dispone de los siguientes *features*: `"commentText","account_age","no_follower","no_following","no_tweets","no_retweets"`, los comentarios (`"commentText"`) no se utilizan para este experimento, es decir, en estos experimentos el objetivo es ver cómo influye en los resultados el hecho de que los *features* de *100k* se repitan varias veces en en el mismo dataset respecto a las pruebas realizadas con el dataset original.

Los algoritmos evaluados son los siguientes: Decision Tree, KNN, Naive Bayes, Random Forest.

100K + Comment + Sent

El objetivo que se persigue con este experimento es crear un dataset más realista introduciendo un *feature* adicional relacionado con los comentarios respecto al dataset *100K + Comment*. Este nuevo *feature* es el sentimiento asociado a los comentarios. Para este experimento, se ha tomado

como referencia el trabajo realizado en Ezpeleta[3], en el cual se añade el humor asociado a los comentarios para obtener mejores resultados.

Para la extracción del sentimiento a partir de los comentarios, se ha empleado VADER (Valence Aware Dictionary and sEntiment Reasoner), un modelo presentado en Hutto[7] especialmente adaptado para la extracción de sentimiento a partir de textos de redes sociales. VADER está basado en un diccionario léxico, esto significa que por cada palabra de una frase se observa su puntuación o categoría a la que pertenece (positiva, negativa, neutra) y se determina la puntuación de toda la frase a partir de éstas. La ventaja de este tipo de enfoque es que no requiere de ningún entrenamiento previo con datos etiquetados, ya que la información necesaria se encuentra en el diccionario.

Principalmente, el análisis de sentimientos de VADER se basa en un diccionario que asigna características léxicas a intensidades emocionales llamadas puntuaciones de sentimientos. La puntuación de sentimiento de un texto se puede obtener al sumar la intensidad de cada palabra en el texto.

Por característica léxica, se refiere a cualquier término que se utilice para la comunicación textual. En las redes sociales pueden encontrarse emoticonos, acrónimos y coloquialismos a los cuales VADER también asigna intensidades emocionales, esto hace que tenga un mejor rendimiento en este tipo de entornos. La intensidad emocional, también llamada puntuación de sentimiento (*sentiment score* en inglés) se mide en una escala desde -4 (extremo negativo) a +4 (extremo positivo), donde el punto medio representa un sentimiento neutral. Para construir este diccionario tan completo en el que se incluyen características léxicas de todo tipo se han empleado votantes humanos que han sido los encargados de asignar las intensidades emocionales a todas ellas; las puntuaciones más populares para cada una de las características léxicas han sido las escogidas.

La implementación de VADER en Python, sin embargo, dada una frase devuelve un número del -1 (extremo negativo) al +1 (extremo positivo). Esta puntuación se halla sumando las puntuaciones de cada una de las palabras del diccionario léxico de VADER de la frase, las cuales tienen una puntuación entre -4 y +4, por lo que tras realizar la suma el total se normaliza a un número entre -1 y +1. La normalización empleada por Hutto es

$$\frac{x}{\sqrt{x^2 + \alpha}}$$

donde x es la suma de las puntuaciones de sentimiento de las palabras de la frase y α es el parámetro de normalización cuyo valor es 15. A continuación puede verse la gráfica de la función de normalización, donde a medida que x crece se acerca más a -1 y 1.

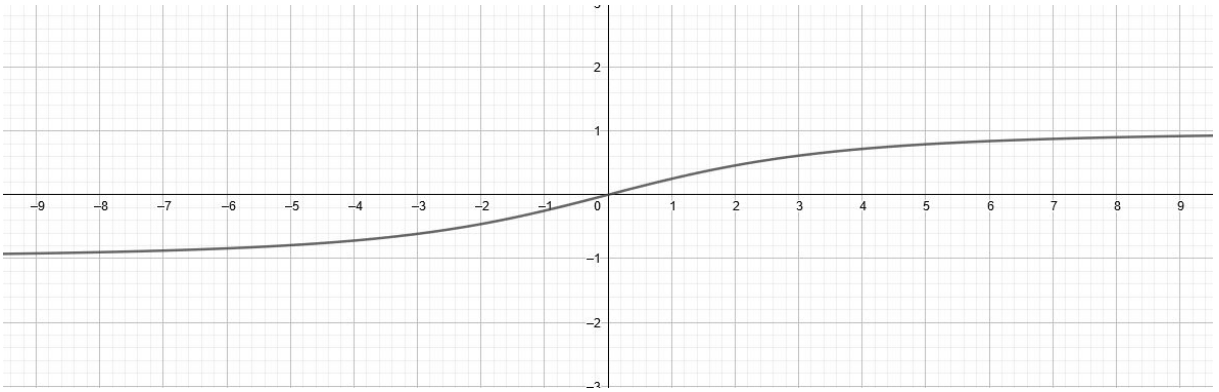

Figura 6: Gráfica generada a partir de la función de normalización.

Las características léxicas no son los únicos factores en la oración que afectan al sentimiento. Existen otros elementos contextuales como la puntuación, el uso de mayúsculas y los modificadores, que también influyen en la emoción[21]. Haciendo uso de todas las características mencionadas, VADER se ha considerado la mejor alternativa para la extracción de sentimiento a partir de comentarios de redes sociales.

Además de emplear VADER para la obtención de sentimiento a partir de texto, se han realizado pruebas con Natural Language API[22] de Google. Se habla del funcionamiento de estos servicios en el apartado 7.2.1 de Implementación.

Los algoritmos evaluados son los siguientes: Decision Tree, KNN, Naive Bayes, Random Forest.

100K + Comment + Sent + Custom

El objetivo de este experimento es tratar de mejorar los resultados obtenidos en anteriores pruebas añadiendo un nuevo *feature*. Para ello, se crea un modelo clasificador de Spam cuyas predicciones sirven como *features* adicionales.

Para la creación del modelo, primero se ha hecho uso de CountVectorizer[23] para el preprocesamiento de texto, tokenización y filtrado de palabras clave. Crea un diccionario de *features* y transforma documentos en vectores de *features*. Tras contar las ocurrencias es posible que un término aparezca muchas más veces en documentos largos que en documentos más cortos. Para evitar estas posibles diferencias, basta con dividir el número de apariciones de cada término en un documento por el número total de términos en el documento (*Term Frequency*).

TF(t) = (Número de apariciones de t en un documento) / (Número total de términos en el documento).

Otro refinamiento además de *Term Frequency*, es reducir el peso de aquellos términos que aparecen en el *corpus* de muchos documentos y, por lo tanto, son menos informativos que los términos que ocurren pocas veces; y aumentar a su vez el peso de aquellos términos menos

[21] Ver http://datameetsmedia.com/vader-sentiment-analysis-explained/ y https://github.com/cjhutto/vaderSentiment para una explicación más detallada de los elementos contextuales empleados por VADER.
[22] Sitio web de Natural Language API: https://cloud.google.com/natural-language/?hl=es
[23] Sitio web de CountVectorizer: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

frecuentes, a esta medida numérica se le llama *Inverse Document Frequency* y se computa de la siguiente manera:

IDF(t) = log_e (Número total de documentos / Número de documentos que contienen el término t).

A partir del producto de estas dos medidas se obtiene el peso TF-IDF[24] (*Term Frequency-Inverse Document Frequency*), el cual es frecuentemente empleado por los motores de búsqueda como herramienta fundamental para medir la relevancia de un documento dada una consulta del usuario.

Para la predicción de la clase se ha empleado Logistic Regression ya que es más apropiado para clasificación binaria.

Una vez se ha construido este modelo customizado, se añade el nuevo *feature* al dataset *100K + Comment + Sent* y se aplica el algoritmo Random Forest. Sin embargo, se realizan dos experimentos distintos, uno entrenando el modelo customizado con el dataset *Youtube Spam Collection Data Set* y otro entrenándolo con los comentarios de *100K + Comment + Sent*. De esta forma puede verse por un lado la posible mejoría en el rendimiento que proporciona este *feature* adicional y por otro, cuánto influye en el resultado el hecho de entrenar el modelo customizado con el mismo dataset que se emplea posteriormente para las predicciones.

Comment + Sent + Custom

Empleando el modelo customizado, del que se habla en el apartado anterior, entrenado con el dataset *Youtube Spam Collection Data Set,* se realiza un experimento que consiste en quitar los *features* del dataset *100K*, es decir, empleando como *features* la predicción del modelo customizado sobre los comentarios y el sentimiento asociado a dichos comentarios. A través de esta prueba se espera observar si empeoran los resultados y al modelo le es realmente útil la información del usuario o si, por el contrario, esta le aporta poca información.

El algoritmo empleado ha sido Random Forest.

100K + Comment + Sent + Custom + FastText

Con el objetivo de intentar mejorar los resultados del experimento *100K + Comment + Sent + Custom*, se opta por añadir un nuevo *feature*. Para ello, se crea un nuevo modelo clasificador de Spam cuyas predicciones sirven como *features* adicionales. En esta ocasión el modelo se realiza con ayuda de FastText[25], una librería de código abierto creada por Facebook AI Research[26], gratuita y ligera que permite a los usuarios aprender representaciones y clasificadores de texto. Se profundiza en el uso de este servicio en el apartado 7.2.4 de Implementación.

El clasificador de FastText obtiene primero un vector a partir del texto realizando un promedio de los *word embeddings*. Para la tarea de clasificación se utiliza Multinomial Logistic Regression, donde el vector corresponde a los *features*.

Al igual que en experimentos anteriores, el modelo que hace uso de FastText se ha entrenado con el dataset *Youtube Spam Collection Data Set* y después se ha entrenado el modelo con los

[24] Para más información consultar http://www.tfidf.com/.
[25] Sitio web de FastText: https://fasttext.cc/
[26] Sitio web de Facebook AI Research: https://research.fb.com/category/facebook-ai-research/

comentarios de *100K + Comment + Sent*. Además, tal y como se ha realizado previamente, el algoritmo aplicado es Random Forest y en esta ocasión, adicionalmente, se ha realizado una prueba con una Neural Network (tratándose en este caso de un Multi-layer Perceptron) tal y como se propone en Gupta[2].

100K + Comment + Sent + Plino

Tal y como se muestra en el apartado 5.4 el mejor resultado hasta el momento se obtuvo con el algoritmo Random Forest empleando el dataset *100K +Comment + Sent*. Partiendo de esta base se ha intentado mejorar los resultados añadiendo un nuevo *feature*, Plino[27], una API para la detección de Spam. El modelo de Plino hace uso de una implementación customizada de Naive Bayes y fue entrenado con un total de cerca de 33.000 correos electrónicos seleccionados del dataset públicamente disponible Enron[28]. Tras clasificar los comentarios del dataset *100K +Comment + Sent*, se ha creado un modelo en el que se aplica el algoritmo Random Forest haciendo uso de este nuevo *feature*.

Además de utilizar Plino, se han realizado pruebas con DatumBox[29] y OOPSpam[30], dos APIs para la detección de Spam. Puede encontrarse documentación sobre el uso de estas tres APIs en los apartados 7.2.5, 7.2.6 y 7.2.7 de Implementación.

5.4 Selección del clasificador

A continuación se muestran los resultados obtenidos en cada uno de los experimentos. Al igual que en anteriores apartados, los experimentos se agrupan por datasets para facilitar la comprensión al lector. Los resultados que se presentan se han medido empleando *10-fold cross validation*. Adicionalmente, se realizan una serie de reflexiones sobre dichos resultados y se selecciona aquel modelo con mejor rendimiento para ser integrado en la función API que será utilizada en Ekitaldi.

Aquellos algoritmos que contengan "-YSCDS" en el nombre significa que dicho modelo ha sido entrenado con *Youtube Spam Collection Data Set*.

[27] Sitio web Plino: https://plino.herokuapp.com/
[28] Dataset Enron: https://www.cs.cmu.edu/~./enron/
[29] Sitio web DatumBox: http://www.datumbox.com/machine-learning-api/
[30] Sitio web OOPSpam: https://www.oopspam.com/

100K

Algoritmo	Accuracy	TPR	FPR	F-measure
Decision Tree	0,95326	0,5548	0,025768421052 63158	0,75906107
KNN	0,95153	0,1538	0,006484210526 315789	0,607921419104 4394
Naive Bayes	0,50981	0,728	0,501673684210 5263	0,394095781665 4014
Random Forest	0,97132	0,4642	0,001989473684 210526	0,801604856226 1247

Tabla 7: Resultados obtenidos empleando dataset 100K.

Aproximadamente, los siguientes resultados son los obtenidos empleando el dataset IV de Chen[1]:

Algoritmo	TPR	FPR	F-measure
Decision Tree	0,52	0,009	0,62
KNN	0,49	0,028	0,46
Naive Bayes	0,06	0,068	0,06
Random Forest	0,54	0,005	0,67

Tabla 8: Resultados obtenidos en Chen[1] empleando dataset 100K (dataset IV).

A continuación se muestra una comparativa de los resultados obtenidos con los del estudio original:

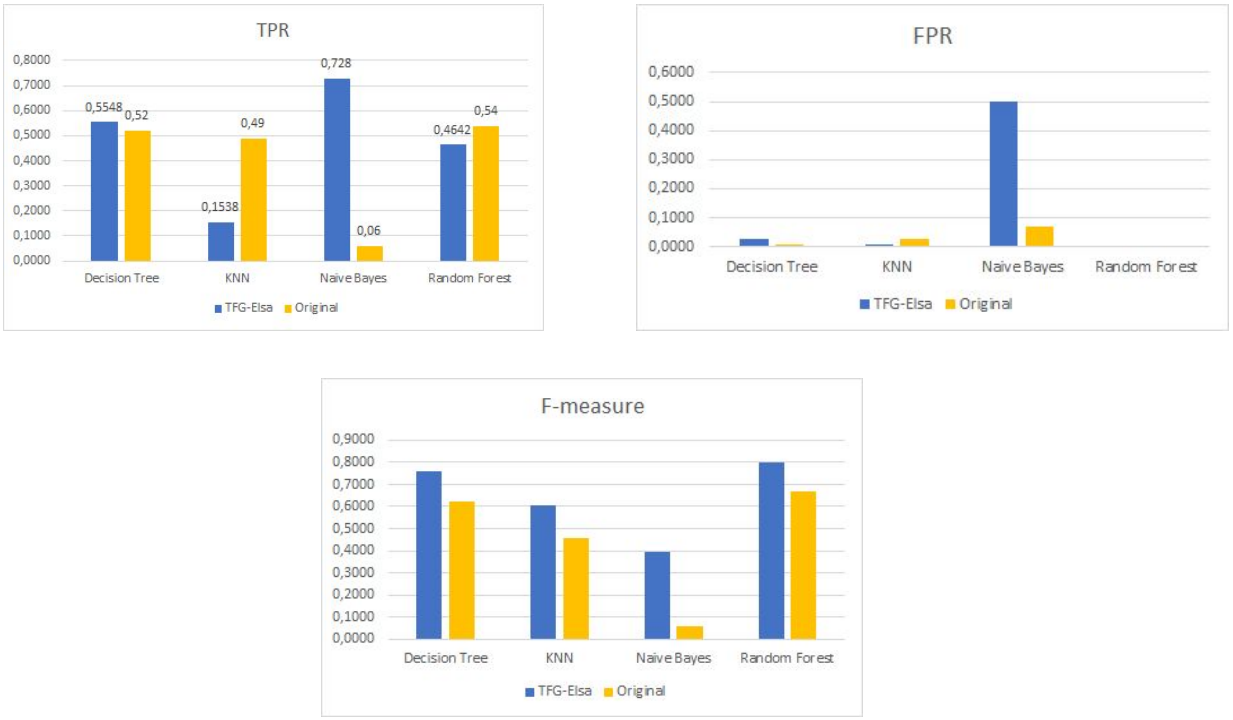

Figura 7: Comparativa resultados obtenidos en este TFG y en Chen[1] empleando dataset 100K.

Tal y como puede apreciarse por los resultados, el rendimiento de los modelos implementados es similar a los que se muestran en Chen[1]. Por lo que se puede afirmar que los modelos implementados a lo largo de este proyecto son comparables a los de otros estudios como garantía adicional.

100K + Comment

Algoritmo	Accuracy	TPR	FPR	F-measure
Decision Tree	0,9655031181106849	0,6872342199616442	0,008504904728711453	0,8771253469503044
KNN	0,9655209257102562	0,6866922371383307	0,008434809360068227	0,8771067963020304
Naive Bayes	0,8825125810690097	0,06253647961310764	0,04089675341617567	0,5102974903768083
Random Forest	0,9652502501965774	0,6753939798215626	0,007675442866433275	0,8748852683657262

Tabla 9: Resultados obtenidos empleando el dataset 100K + Comment.

Se puede apreciar que respecto a los resultados obtenidos con el dataset *100K* se han mejorado los resultados, esto se debe a las repeticiones de la información de usuario.

100K + Comment + Sent

Algoritmo	Accuracy	TPR	FPR	F-measure
Decision Tree	0,9384996741209278	0,5759192862503126	0,02763315199401853	0,7909777622882794
KNN	0,9437457929546013	0,48411573417827064	0,013322014229359835	0,7824849013757962
Naive Bayes	0,8890444085918107	0,05553239389643959	0,033100590748190176	0,5098708799895578
Random Forest	0,9522328949102319	0,5335195530726257	0,008656778027438443	0,8152445590472499

Tabla 10: Resultados obtenidos empleando el dataset 100K + Comment + Sent.

Se puede ver que el mejor resultado lo ofrece Random Forest, el cual sobrepasa los resultados obtenidos por Gupta[2] empleando el mismo algoritmo pero añadiendo el sentimiento de los comentarios, en vez de únicamente empleando la información del usuario.

También se superan los resultados obtenidos en Ezpeleta[3]. En éste se alcanza el mínimo FP en 53 falsos positivos. Teniendo en cuenta que el dataset que utiliza es de 1.000 comentarios Spam y 3.000 comentarios no-Spam, equivaldría a un FPR = 0,017. Sin embargo en este experimento se mejora dicho FPR reduciéndolo a un 0,008, además de obtener un accuracy = 95,22, el cual supera el máximo obtenido en Ezpeleta[3] que ha sido un 82,58.

100K + Comment + Sent + Custom

Algoritmo	Accuracy	TPR	FPR	F-measure
Random Forest	0,9780895294876041	0,8121821062286334	0,006413726230855203	0,9258610589889782
Random Forest-YSCDS	0,9504485734331983	0,5068373217710331	0,008115486014026863	0,8047285594236921

Tabla 11: Resultados obtenidos empleando el dataset 100K + Comment + Sent + Custom.

Al añadir este nuevo *feature* únicamente se ha logrado mejorar los resultados obtenidos anteriormente empleando como *feature* el modelo customizado entrenado con el mismo dataset sobre el que luego se hacen las predicciones. Sin embargo, no es acertado tener en consideración esta "mejoría" ya que el entrenar con el mismo dataset hace que el modelo pierda capacidad de generalización, una cualidad de gran importancia en todo modelo.

Comment + Sent + Custom

Algoritmo	Accuracy	TPR	FPR	F-measure
Random Forest-YSCDS	0,9362274244156437	0,3296089385474 86	0,0071107857301406195	0,7175125888336802

Tabla 12: Resultados obtenidos empleando el dataset Comment + Sent + Custom.

Se puede comprobar en este experimento que los datos de usuario sí aportan información al modelo debido a que los resultados han empeorado considerablemente, por lo que es conveniente tenerlos en cuenta como *features*.

100K + Comment + Sent +Custom + FastText

Algoritmo	Accuracy	TPR	FPR	F-measure
Neural Network	0,9222057205132862	0,1446260318519136	0,005163692156717668	0,6000317100823744
Random Forest	0,9778687152529213	0,8098057200033353	0,006433197166589432	0,92503501711772
Random Forest-YSCDS	0,9492519027420142	0,4879096139414 6587	0,007655971930699046	0,797196487701886

Tabla 13: Resultados obtenidos empleando el dataset 100K + Comment + Sent + Custom + FastText.

Al igual que en *100K + Comment + Sent + Custom*, al añadir el *feature* de las predicciones del modelo que hace uso de FastText, únicamente se ha logrado mejorar los resultados obtenidos anteriormente empleando como *feature* el modelo entrenado con el mismo dataset sobre el que luego se hacen las predicciones. Por lo que, nuevamente, no se considera una mejoría realista.

100K + Comment + Sent + Plino

Algoritmo	Accuracy	TPR	FPR	F-measure
Random Forest-YSCDS	0,9505091192717404	0,5042524806136913	0,007807845229426035	0,8042978446885957

Tabla 14: Resultados obtenidos empleando el dataset 100K + Comment + Sent + Plino.

En esta ocasión tampoco se obtiene una mejoría respecto a *100K + Comment + Sent*, lo cual puede deberse a que Plino es una herramienta cuyo modelo ha sido entrenado con correos y la forma en la que se escribe los correos es muy distinta de la forma en la que se escribe en la redes sociales. Por lo que el no estar optimizada para comentarios de redes sociales, se puede traducir en resultados peores. Además, el modelo es una implementación customizada de Naive Bayes, modelo que durante los experimentos ha mostrado un rendimiento peor al resto de algoritmos.

Clasificador seleccionado

Como conclusión de este apartado, tras ver los resultados obtenidos, se observa que el mejor rendimiento se obtiene con *100K + Comment + Sent* empleando el algoritmo Random Forest, ya que este es el que obtiene mejores resultados dentro de un marco realista, por lo que este será el modelo a integrar en la función API de la que hará uso la red social.

Random Forest, como su nombre lo indica, consiste en un gran número de árboles de decisión individuales que operan como un conjunto. Cada árbol individual en el bosque aleatorio devuelve una predicción de clase y la clase con la mayor cantidad de votos se convierte en la predicción del modelo (véase la Figura 8). La baja correlación entre modelos aumenta las probabilidades de realizar predicciones correctas.

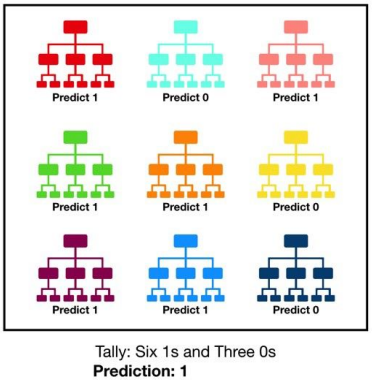

Figura 8: Random Forest realizando una predicción en base al resultado de cada árbol.

En todo los experimentos realizados con Random Forest el bosque ha estado conformado por 100 árboles.

5.4 Multi-idioma

Una vez se ha seleccionado el mejor modelo, se obtiene un clasificador de Spam en inglés. Sin embargo, teniendo en cuenta que se trata de un proyecto con la ambición de ser escalable, y el castellano es una de las lenguas más habladas en el mundo actualmente, se ha considerado de interés realizar una prueba de concepto de cómo sería trasladar todo el proceso realizado hasta el momento a otro idioma (en este caso al castellano), y ver su viabilidad de cara al futuro.

Por esta razón, se va a emplear el dataset *YCD_ES* construido durante la recolección de comentarios (ver apartado 5.2), que contiene 9.433 comentarios en castellano. En este caso al igual que anteriormente, se combina el dataset *100K* con *YCD_ES*, sin embargo, en esta ocasión el número de comentarios es menor que el número de usuarios, por lo que simplemente se asigna un comentario por usuario y los usuarios restantes no se utilizan. El resultado es un dataset que tiene 9.156 comentarios Spam y 277 comentarios no-Spam.

Ya que queremos realizar el mismo proceso aplicado en *100K + Comment + Sent*, debemos añadirle el sentimiento. Para ello, debido a que actualmente no existen soluciones para la extracción de sentimiento a partir de un texto en castellano, y debido a la falta de tiempo para la construcción de un modelo customizado para ello, se ha optado por traducir primero los comentarios al inglés empleando la librería de Python googletrans[31] (también puede servir para detectar el idioma de un texto) y después emplear VADER, al igual que anteriormente, para la extracción de sentimiento.

También se han probado para la traducción de texto el Cloud Translation API[32] de Google y translate[33], una librería de Python para la traducción de texto. Puede encontrarse documentación sobre el uso de ambos en el apartado 7.2.3 de Implementación.

A continuación se muestran los resultados obtenidos por este modelo:

Algoritmo	Accuracy	TPR	FPR	F-measure
Random Forest	0,9709530372097954	0,04693140794223827	0,00109217999126256	0,5359542532945527

Tabla 15: Resultados obtenidos empleando el dataset 100K + Comment + Sent en castellano.

Tal y como puede apreciarse, los resultados son peores que los obtenidos en inglés. Esto se debe a que la traducción de coloquialismos, empleados frecuentemente en redes sociales, no se realiza correctamente, lo cual provoca que la extracción del sentimiento a partir de la traducción no sea de calidad. Además, el tamaño del dataset en castellano es considerablemente menor. Sin embargo, hay oportunidad de mejora si se crea un modelo customizado para la extracción de sentimiento a partir de textos en castellano.

Aunque el rendimiento de este modelo no sea el óptimo, se considera de gran valor realizar una integración de este modelo en una nueva función API que integre el modelo en inglés y en castellano para experimentar con la viabilidad de trasladar el conocimiento adquirido a nuevos idiomas y que sirva como base para ser mejorado en el futuro.

5.5 Resumen

En este apartado se muestra una tabla a modo de resúmen del capítulo 5. En dicha tabla pueden encontrarse listados los datasets junto con su tamaño[34], *features* y el origen de éstos, además de los algoritmos de clasificación empleados con cada uno de ellos y el F-measure obtenido.

[31] La librería googletrans hace uso de Google Translate API, un sistema de traducción automática neuronal de última generación. Enlace a la librería googletrans: https://pypi.org/project/googletrans/

[32] Enlace a AI & Machine Learning Products: https://cloud.google.com/products/machine-learning/?hl=Es

[33] Enlace a la librería translate: https://pypi.org/project/translate/

[34] Por tamaño se entiende el número de muestras de datos.

Dataset	Tamaño	Features	Origen	Algoritmos	F - measure
100K	100.000	account_age, no_follower, no_following, no_userfavourites, no_lists, no_tweets, no_retweets, no_hashtag, no_usermention, no_urls, no_char, no_digits	Dataset IV de Chen[1].	Decision Tree	0,75906107
				KNN	0,607921419104439 4
				Naive Bayes	0,394095781665401 4
				Random Forest	0,801604856226124 7
100K + Comment	280.779	account_age, no_follower, no_following, no_tweets, no_retweets	Es la unión los datasets 100k y *YCD_EN (comentarios en inglés obtenidos a partir de Youtube Comments Dataset* de Ezpeleta[3]).	Decision Tree	0,877125346950304 4
				KNN	0,877106796302304
				Naive Bayes	0,510297490376808 3
				Random Forest	0,874885268365726 2
100K + Comment + Sent	280.779	sentiment, account_age, no_follower, no_following, no_tweets, no_retweets	Es el dataset 100K + Comment con un *feature* adicional: el sentimiento extraído de cada comentario empleando VADER.	Decision Tree	0,790977762288279 4
				KNN	0,782484901375796 2
				Naive Bayes	0,509870879989557 8
				Random Forest	0,815244559047249 9
100K + Comment + Sent + Custom	280.779	custom, sentiment, account_age, no_follower, no_following, no_tweets,	Es el dataset 100K + Comment + Sent con un *feature* adicional: la	Random Forest	0,925861058988978 2
				Random	0,8047285594

		no_retweets	predicción de un clasificador de Spam customizado.	Forest-YSCDS [35]	236921
Comment + Sent + Custom	280.779	custom, sentiment	Es el dataset 100K + Comment + Sent + Custom sin los *features* relacionados con el usuario.	Random Forest-YSCDS	0,7175125888336802
100K + Comment + Sent +Custom + FastText	280.779	fasttext, custom, sentiment, account_age, no_follower, no_following, no_tweets, no_retweets	Es el dataset 100K + Comment + Sent + Custom con un *feature* adicional: la predicción de un clasificador de Spam creado con FastText.	Neural Network	0,6000317100823744
				Random Forest	0,9250350171 1772
				Random Forest-YSCDS	0,7971964877 01886
100K + Comment + Sent + Plino	280.779	plino, sentiment, account_age, no_follower, no_following, no_tweets, no_retweets	Es el dataset 100K + Comment + Sent + Custom con un *feature* adicional: la predicción de Plino, un servicio para la clasificación de Spam.	Random Forest-YSCDS	0,8042978446885957
100K + Comment + Sent (ES)	9.433	sentiment, account_age, no_follower, no_following, no_tweets, no_retweets	Es la unión los datasets 100k y *YCD_ES (comentarios en castellano obtenidos a partir de Youtube Comments Dataset* de	Random Forest	0,5359542532945527

[35] YSCDS hace referencia a que el modelo empleado para el nuevo feature ha sido entrenado con el dataset *Youtube Spam Collection Data Set.*

			Ezpeleta[3]).		

Tabla 16: Resumen capítulo 5.

El mejor rendimiento se obtiene con *100K + Comment + Sent* empleando el algoritmo Random Forest, ya que este es el que obtiene mejores resultados dentro de un marco realista, por lo que es el modelo seleccionado, junto con su versión en castellano, a integrar en las funciones API de las que hará uso la red social EkitApp.

6. Diseño

En este capítulo se expone la arquitectura general de la solución la cual proporciona una visión general del proyecto, tras ello se profundiza en el diseño del modelo de datos realizado en colaboración con Ekitaldi y finalmente se explica el diseño de las funciones API.

6.1 Arquitectura de la solución

A continuación se expone la arquitectura general de la solución, la cual permite adoptar una visión global del proyecto colaborativo de Ekitaldi y el actual, tras ello, se profundizará en la solución específica desarrollada en el Trabajo de Fin de Grado actual. En las distintas arquitecturas pueden visualizarse las partes en lila implementadas por Smart, y en rojo por Ekitaldi.

6.1.1 Arquitectura general de la solución

La arquitectura, en rasgos generales, de la aplicación conjunta Smart-Ekitaldi queda reflejada en el siguiente esquema:

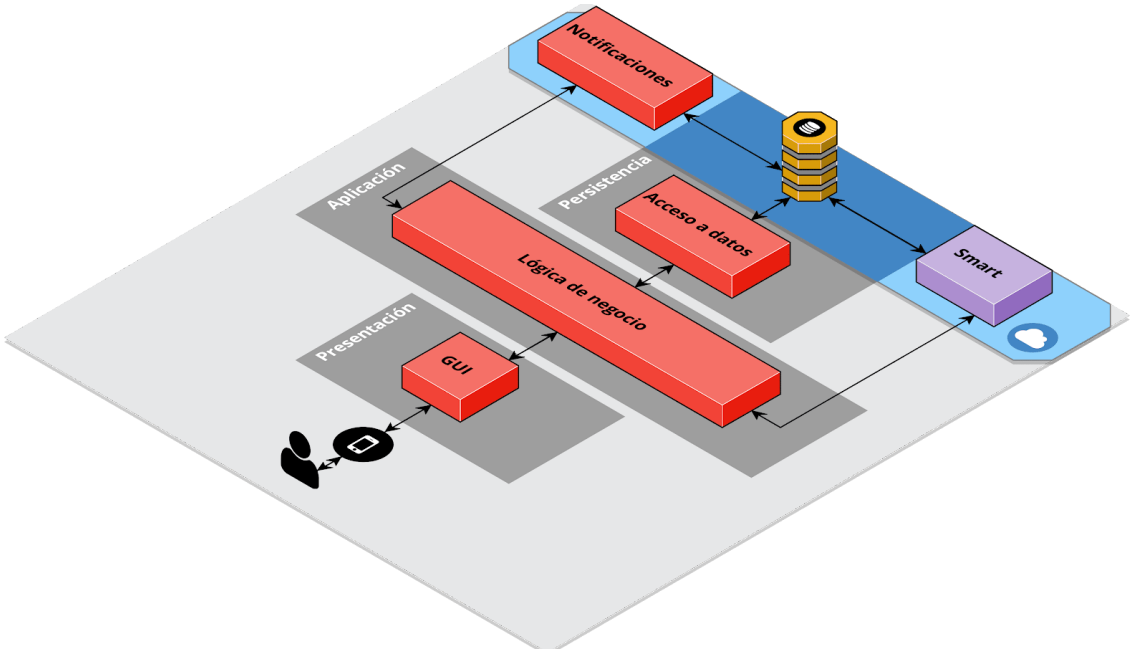

Figura 9: Arquitectura conjunta Smart-Ekitaldi.

La aplicación accede a la capa de Presentación, ésta es la que muestra la interfaz gráfica y a su vez, interactúa con la capa de Aplicación, en la que se incluye la lógica de negocio. Desde la lógica de negocio se invocan los servicios en la nube de las Funciones Cloud, Notificaciones y Smart, las cuales accederán a la base de datos cuando lo necesiten. Además, la lógica de negocio interactúa a su vez con la capa de la persistencia mediante el módulo de Acceso a datos. Finalmente, la base de datos, a la que accede también el módulo de Acceso a datos, está almacenada en la nube y es NoSQL. Su estructura, como puede verse en el apartado 6.2.1, ha sido desarrollada de forma conjunta.

En la nube, el servicio que da soporte al acceso a las funcionalidades de Notificaciones y Smart es FIrebase Functions y, a la base de datos, Firestore.

6.1.2 Arquitectura específica de Smart

En esta sección se ahondará más en la arquitectura de Smart.[36] De nuevo, la caja roja representa Ekitaldi. Se encuentran en color lila aquellas partes que hayan sido desarrolladas enteramente por mi.

En la Figura 10 se muestra la arquitectura con el tipo de servicio empleado, mientras en la Figura 11, más abajo, se encuentra la arquitectura con los nombres de las tecnologías empleadas.

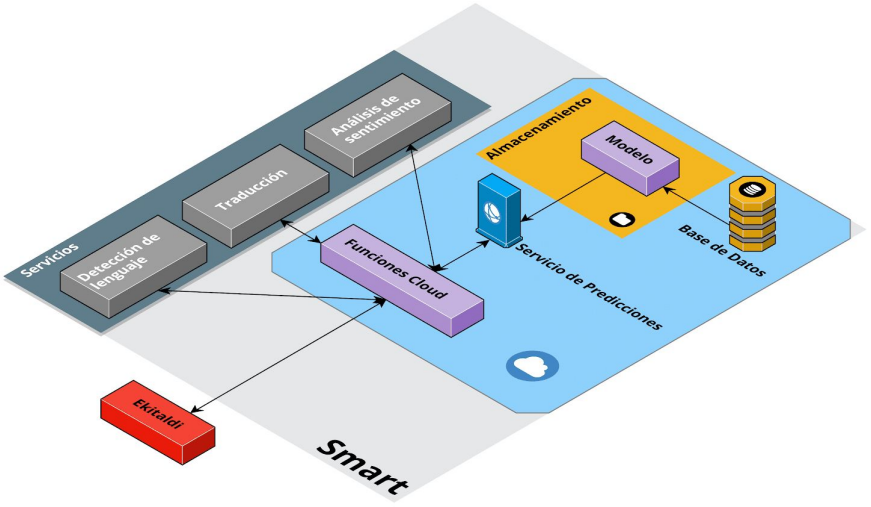

Figura 10: Arquitectura Smart.

[36] Para ver la arquitectura específica de la parte Ekitaldi, dirigirse al apartado 4.1.2 de la memoria de Ekitaldi.

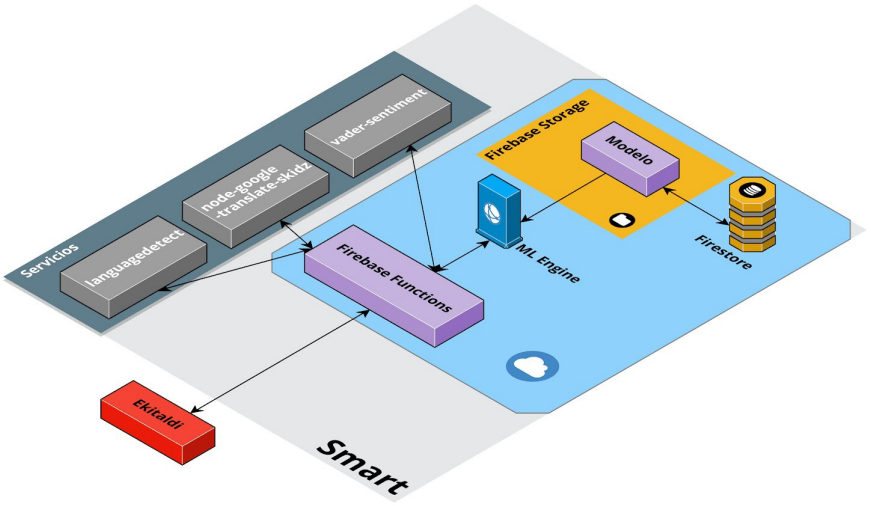

Figura 11: Arquitectura Smart tecnologías.

La aplicación, que se encuentra en la parte Ekitaldi, llama a la función que ha sido desplegada previamente como API con los parámetros correspondientes (comentario y algunos datos de usuario), ésta a su vez llama al paquete de detección de lenguaje, que detectará el idioma en el que está escrito el comentario; si este es en castellano, primero tendrá que traducirlo al inglés empleando el paquete de traducción. Una vez el comentario está escrito en inglés, se efectúa un análisis de sentimiento empleando el paquete de análisis de sentimiento. El sentimiento extraído formará parte de la entrada de la llamada al servicio de predicciones junto con los datos del usuario. El servicio de predicciones introducirá los parámetros al modelo que ha sido entrenado con los datos de la base de datos y le devolverá una clase: True o False, dependiendo de si se trata de un comentario Spam o no. Esta respuesta se devuelve a la función que a su vez devuelve dicha respuesta a la aplicación.

6.2 Diseño del modelo de datos

En este apartado se profundiza en el diseño de la base de datos, el cual se realizó en colaboración con Ekitaldi, y el sistema de ficheros.

6.2.1 Base de datos

A continuación se muestra la estructura de la base de datos, separada en cuatro colecciones de documentos: colección de usuarios, colección de eventos, colección de comentarios y colección de notificaciones. Además, también se muestra el diseño gráficamente en el anexo 1.

Colección de usuarios (users): almacena los datos de cada usuario. El identificador de cada uno de los documentos es el identificador del usuario (proporcionado por el servicio Firebase Auth) y en estos documentos se almacenan sus puntuaciones, los seguidores, los seguidos, los eventos a los que asiste, los eventos creados y los guardados. Algunos datos, como los seguidores y los seguidos, se

duplican para acceder a un solo documento para consultarlos sin tener que recoger los documentos de los otros usuarios.

ID del documento: ID del usuario de Firebase Auth.
name: string - El nombre del usuario.
imageUrl: string - La dirección de la imagen de perfil del usuario.
commentScore: number - La puntuación media que obtengan sus comentarios.
eventScore: number - La puntuación media que obtengan sus publicaciones de eventos.
totalScore: number - La puntuación media total del usuario.
followers: array<string> - Los identificadores de los seguidores del usuario actual.
following: array<string> - Los identificadores de los usuarios a los que sigue el usuario actual.
attending: array<string> - Los identificadores de los eventos a los que asiste el usuario actual.
createdEvents: array<string> - Los identificadores de los eventos que ha creado el usuario actual.
saved: array<string> - Los identificadores de los eventos guardados por el usuario actual.(ID del evento)
creationDate: timestamp - La fecha de creación del usuario actual.

Colección de eventos (events): almacena los datos de los eventos. El identificador de cada uno de los documentos es el identificador del evento, el cual se asigna automáticamente al generar el documento. Se duplican los datos del usuario creador para evitar acceder a su documento cuando se quiera mostrar en el evento. Asimismo, se duplican algunos comentarios, ya que la mayoría de usuarios no suelen ver todos los comentarios, sino unos pocos, y de esta forma evitamos tener que acceder a los documentos de los comentarios para visualizar los primeros.

ID del documento: ID del evento (automático)
title: string - El título de la publicación.
description: string - La descripción de la publicación.
imageName: string - El nombre de la imagen en Firebase Storage.
oneStar: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con una estrella.
twoStars: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con dos estrellas.
threeStars: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con tres estrellas.
score: number - La puntuación media del evento actual.
comments: array<map> - Parte de los comentarios a mostrar del evento.
- **uid**: string - El identificador del usuario creador del comentario actual.
- **name**: string - El nombre del usuario creador del comentario actual.
- **imageUrl**: string - La imagen de perfil del usuario creador del comentario actual.
- **comment**: string - El texto del comentario actual.
- **commentID**: string - El identificador del comentario actual.
- **oneStar**: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con una estrella.

twoStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con dos estrellas.
threeStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con tres estrellas.
score: number - La puntuación media del comentario actual.
creationDate: timestamp - La fecha de creación del comentario actual.
creationDate: timestamp - La fecha de creación del evento actual.
eventDate: timestamp - La fecha de celebración del evento actual.
campus: number - El número del campus (0 - Araba, 1 - Bizkaia, 2 - Gipuzkoa).
eventLocation: string - La ubicación en la que se celebrará el evento actual.
attending: array<string> - Los identificadores de los usuarios que asistirán al evento actual.
tag: number - El número de la etiqueta (0 - Charla, 1 - Taller, 2 - Competición, 3 - Exposición, 4 - Actuación, 5 - Curso, 6 - Fiesta, 7 - Otros).
uid: string - El identificador del usuario creador del evento actual.
imageUrl: string - La imagen de perfil del usuario creador del evento actual.
name: string - El nombre del usuario creador del evento actual.

Colección de comentarios (comments): almacena los datos de los comentarios de los eventos, el identificador de cada uno de los documentos es el identificador del comentario y se asigna de forma automática al generar el documento. Se duplican los datos del usuario creador para evitar acceder a su documento cuando se quiera mostrar en el evento.

ID del documento: ID del comentario (aleatorio)
uid: string - El identificador del usuario creador del comentario actual.
name: string - El nombre del usuario creador del comentario actual.
imageUrl: string - La imagen de perfil del usuario creador del comentario actual.
comment: string - El texto del comentario actual.
oneStar: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con una estrella.
twoStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con dos estrellas.
threeStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con tres estrellas.
score: number - La puntuación media del comentario actual.
creationDate: timestamp - La fecha de creación del comentario actual.
polarity: number - La polaridad del texto del comentario actual.
eventID: string - El identificador del evento al que pertenece el comentario actual.

Colección de notificaciones (notifications): almacena los datos de notificaciones de cada usuario, el identificador de cada uno de los documentos es el identificador del usuario (proporcionado por el servicio Firebase Auth). Dispone de una lista con los tokens de cada usuario para saber a qué

dispositivos enviar las notificaciones y el idioma que emplea para así adaptar los textos. Además, las notificaciones se guardan en una lista para poder mostrarlas en la página de Actividad.

ID del documento: ID del usuario de Firebase Auth.
tokens: array<string> - Los identificadores de los dispositivos del usuario actual a los que mandar notificaciones.
lang: string - El idioma que utiliza el usuario en la aplicación.
countryCode: string - El código del país del idioma que utiliza el usuario en la aplicación.
notifications: array<map> - Las notificaciones del usuario actual.
- **type**: number - El tipo de notificación (0 - seguimiento, 1 - puntuación, 2 - asistencia, 3 - comentario).
- **date**: timestamp - La fecha en la que se produce la notificación.
- **fromUid**: string - El identificador del usuario origen de la notificación.
- **eventID**: string - El identificador del evento al que está asociada la notificación (tendrá valor nulo si el tipo es 0).

6.2.3 Sistema de ficheros

La lógica de los ficheros de la implementación de la función ha seguido la siguiente estructura:

- **Datasets**: contiene todos los datasets obtenidos a partir de estudios realizados con anterioridad, así como aquellos construidos para después ser empleados en la creación de los modelos de Machine Learning.
- **Modelos**: se encuentran los Jupyter Notebook que contienen los distintos clasificadores correspondientes a los experimentos realizados empleando distintos datasets y algoritmos de Machine Learning.
- **Funciones**: incluye todos aquellos archivos correspondientes a la implementación de las funciones y dependencias.

El control de versiones se ha realizado empleando Git[37] y se ha respaldado con Github[38].

6.3 Funciones API

Tras realizar numerosos experimentos, se han seleccionado dos clasificadores de comentarios Spam, uno para textos en inglés y otro para textos en castellano. Por lo tanto, se decide implementar dos funciones (las cuales posteriormente serán desplegadas para ser integradas en EkitApp), una de ellas para la detección de Spam en inglés, y la otra para la detección multi-idioma de Spam (en inglés y castellano). A pesar de que el rendimiento del clasificador en castellano es menor, es interesante implementar una función multi-idioma, ya que permite realizar una prueba de concepto en la que se agrega un idioma adicional en la función, a través de la cual se pone a prueba la viabilidad de la

[37] Sitio web Git: https://git-scm.com/
[38] Sitio web GitHub: https://github.com/

solución y permite reflexionar sobre sus mejoras futuras, así como apreciar algunas diferencias entre clasificadores para un idioma y multi-idioma, como los tiempos de respuesta.

A continuación se muestran de forma detallada los diagramas de flujo de ambas funciones:

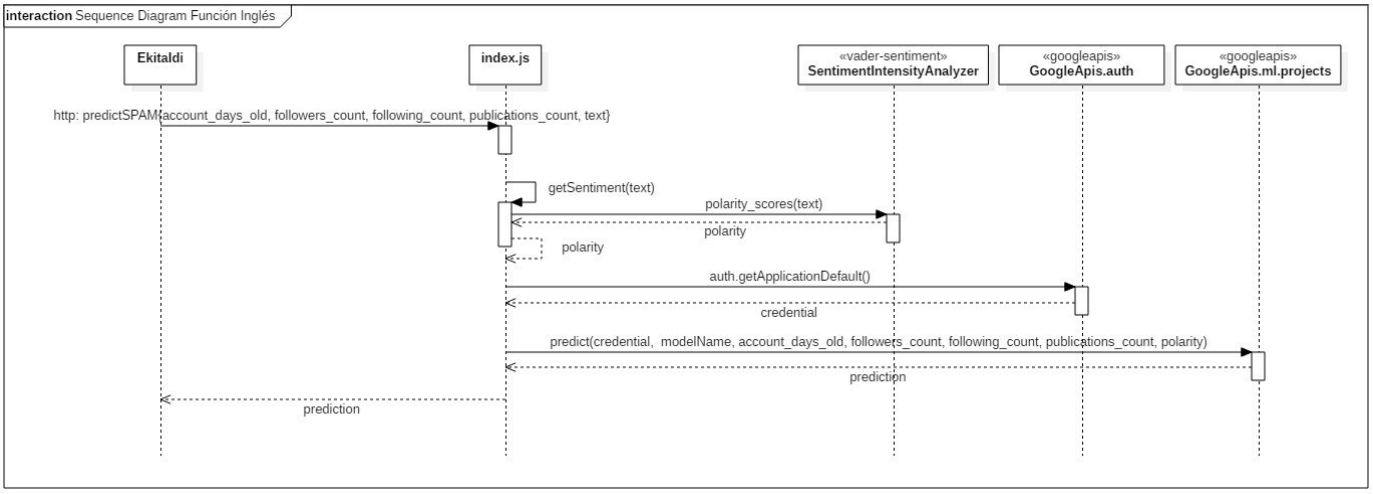

Figura 12: Diagrama de flujo función en inglés.

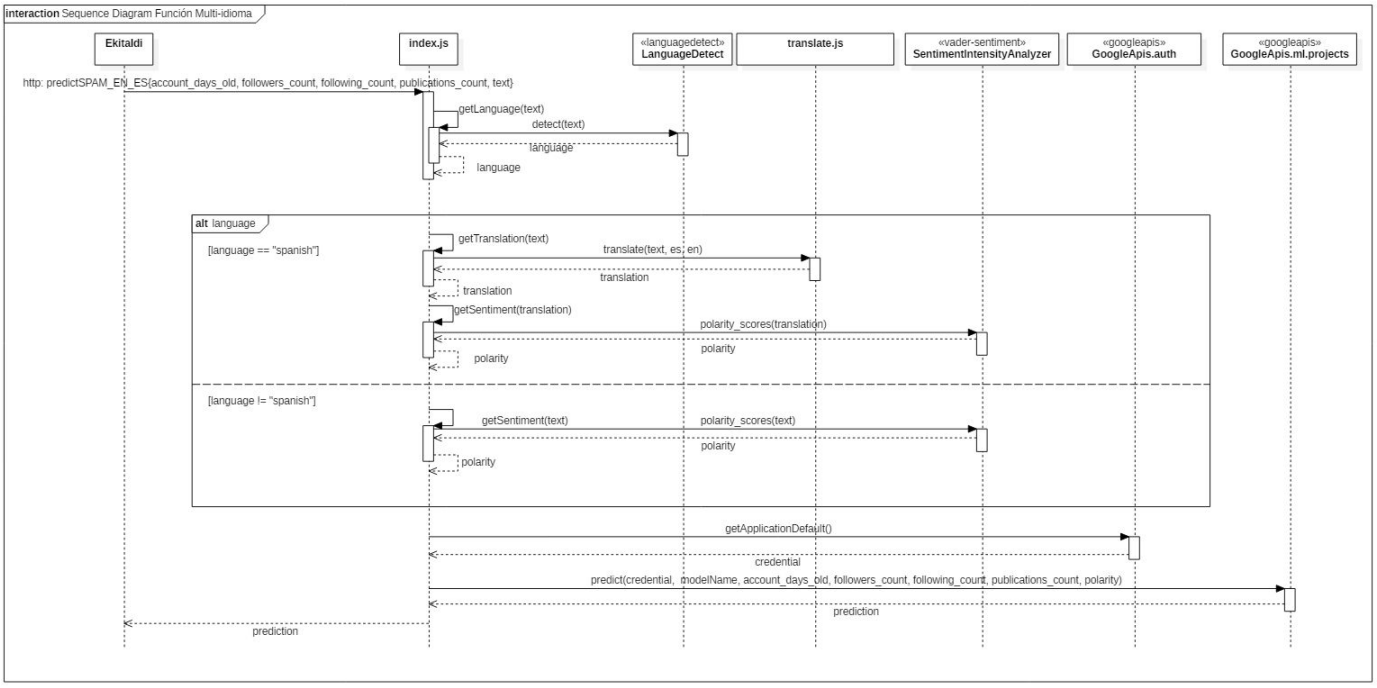

Figura 13: Diagrama de flujo función multi-idioma.

7. Implementación

En el capítulo actual se realiza un recorrido a través de la implementación de las funciones explicando distintos fragmentos de código. Además se explica el uso de distintos servicios empleados a lo largo del proyecto con ejemplos prácticos.

7.1 Funciones

Tal y como se explica en mayor profundidad en el apartado 6.3 de Diseño, se van a implementar dos funciones, una para la detección de Spam en inglés y la otra para la detección de Spam multi-idioma.

Antes de crear las funciones tal y como se muestra en el capítulo actual, primero debe subirse el modelo a ML Engine, se muestra cómo hacerlo en el apartado 9.1 de Despliegue.

Previo al comienzo de la implementación de las funciones, deben realizarse una serie de configuraciones. Para empezar, deben ejecutarse los siguientes comandos en la terminal:

Nota 1: es necesario tener previamente instalado en el ordenador Node.js[39] para que éstos funcionen.

Nota 2: en caso de tratarse de un terminal Unix o Mac OS X debe añadirse `sudo` al principio de cada uno de los comandos.

Primero se instala el cliente de Firebase:

```
$ npm install -g firebase-tools
```

Se accede a la cuenta de Google:

```
$ firebase login
```

Se inicializa un directorio de proyecto nuevo (debe de asegurarse de encontrarse en el directorio en el cual se desea inicializarlo) ejecutando el siguiente comando:

```
$ firebase init
```

Al ejecutar este último comando, se debe responder a una serie de cuestiones.

Al preguntar por el proyecto de Firebase que se quiere en el directorio, debe seleccionarse aquel proyecto de Firebase que contenga el modelo de Machine Learning que previamente se ha exportado. Se selecciona JavaScript como lenguaje de programación. No se utiliza ESLint, por lo que se responde *no*, y finalmente, se responde *sí* a la instalación de dependencias con npm.

[39] Enlace descarga Node.js: https://nodejs.org/es/

Una vez haya terminado de crearse el proyecto, el directorio tendrá la siguiente estructura:

```
myproject
 +- .firebaserc    # Hidden file that helps you quickly switch between
 |                 # projects with `firebase use`
 |
 +- firebase.json  # Describes properties for your project
 |
 +- functions/     # Directory containing all your functions code
      |
      +- .eslintrc.json  # Optional file containing rules for JavaScript
linting.
      |
      +- package.json  # npm package file describing your Cloud Functions code
      |
      +- index.js      # main source file for your Cloud Functions code
      |
      +- node_modules/ # directory where your dependencies (declared in
                       # package.json) are installed
```

Dentro de este directorio únicamente serán modificados los ficheros `index.js` y `package.json`.

Se instalan los paquetes de la API de Google:

```
$ npm i googleapis
```

Se comprueba que los paquetes se han instalado correctamente abriendo `package.json`. En caso de que no estuvieran todos, pueden añadirse a mano, se mostrarán las dependencias en los próximos subapartados 7.1.1 y 7.1.2 en cada una de las funciones.

A continuación se muestra primero la implementación de la función para la detección de comentarios Spam en inglés y seguidamente la versión multi-idioma.

7.1.1 Detección de comentarios Spam en inglés

Antes de comenzar con la implementación de la función, es oportuno asegurarse de la correcta inclusión de todos los paquetes dentro de `package.json` que deberá tener el siguiente aspecto (las versiones varían):

```
"dependencies": {
   "firebase-admin": "~7.0.0",
   "firebase-functions": "^2.3.0",
   "googleapis": "^39.2.0",
   "vader-sentiment": "^1.1.3"
 }
```

En estas dependencias encontramos:

- *firebase-admin*: es el SDK de Admin, el cual permite interactuar con Firebase desde entornos privilegiados.
- *firebase-functions*: es un SDK para la definición de Cloud Functions[40] para Firebase.
- *googleapis*: es la librería cliente Node.js[41] para la utilización de Google APIs.
- *vader-sentiment*: puerto Javascript de la herramienta de análisis de sentimiento VADER[42], a través del cual puede determinarse el sentimiento a partir de texto.

A continuación se explica la implementación de la propia función:

Se comienza por cargar los módulos *firebase-functions* y *firebase-admin.*

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
```

Se declara la función encargada de devolver la polaridad de un texto dado, para ello, hace uso del módulo *vader-sentiment*.

```
function getSentiment(text){
   const vader = require('vader-sentiment');
   const intensity =
vader.SentimentIntensityAnalyzer.polarity_scores(text)['compound'];
   return intensity
}
```

Se carga el módulo *googleapis* y se referencia la versión 1 de ml.

```
admin.initializeApp(functions.config().firebase);
const googleapis_1 = require("googleapis");
const ml = googleapis_1.google.ml('v1');
```

La función a la que se le envía las peticiones es una función https.

```
exports.predictSPAM = functions.https.onRequest(async (request, response) =>
{
```

Se toman los datos de entrada que serán después enviados al módulo, además del texto del comentario, a partir del cual se extrae el sentimiento empleando la función implementada previamente.

```
  const account_days_old = request.body.account_days_old;
  const followers_count = request.body.followers_count;
  const following_count = request.body.following_count;
  const publications_count = request.body.publications_count;
```

[40] Documentación Cloud Functions: https://firebase.google.com/docs/functions
[41] Sitio web Node.js: https://nodejs.org/en/
[42] GitHub de VADER: https://github.com/cjhutto/vaderSentiment

```
  const text = request.body.text;
  const compound = getSentiment(text);
```

Tras ello, se construye el input del modelo, que debe tener la misma estructura (orden de *features*) que con la que se entrenó.

```
  const instance =
[[compound,account_days_old,followers_count,following_count,publications_cou
nt]]
```

Ahora se realiza la petición a la API de Google, esta petición requiere autenticación, la cual conectará nuestros credenciales de Firebase con Google API.

```
  const model = "random_forest_spam_wSentiment";
  const { credential } = await
googleapis_1.google.auth.getApplicationDefault();
```

Después de guardar en una variable el nombre de nuestro modelo, por último se realiza una llamada de predicción a la API de ML engine, esto se hace pasando los credenciales, el nombre del modelo y la instancia que se intenta predecir.

```
  const modelName = `projects/focal-healer-239910/models/${model}`;
  const preds = await ml.projects.predict({
      auth: credential,
      name: modelName,
      requestBody: {
          instance
      }
  });
  response.send(preds.data['predictions'][0]);
});
```

7.1.2 Detección de comentarios Spam multi-idioma

Tal y como se menciona en el anterior apartado, antes de comenzar con la implementación de la función, es oportuno asegurarse de la correcta inclusión de todos los paquetes dentro de `package.json` que deberá tener el siguiente aspecto (las versiones varían):

```
"dependencies": {
   "firebase-admin": "~7.0.0",
   "firebase-functions": "^2.3.0",
   "googleapis": "^40.0.0",
   "vader-sentiment": "^1.1.3",
   "languagedetect": "^1.2.0",
```

```
    "node-google-translate-skidz": "^1.1.2"
}
```

Además de las dependencias mostradas en el apartado anterior encontramos dos adicionales:

- *languagedetect*: puerto de PEAR:Text_LanguageDetect[43] para Node.js, el cual puede identificar 52 idiomas a partir de texto.
- *node-google-translate-skidz*: cliente Node.js para la API de Google Translate.

En esta ocasión, en vez de acceder a un único modelo, la función deberá detectar primero el idioma del comentario, para ello, emplea el módulo *languagedetect*. Si el comentario es en inglés, se extrae directamente el sentimiento (al igual que en la función explicada en el apartado anterior) y se realiza una predicción a través del modelo para la detección de Spam en inglés. Si el comentario es en castellano, se traduce al inglés empleando el módulo *node-google-translate-skidz* (ya que VADER únicamente es capaz de extraer el sentimiento de texto en inglés), se extrae el sentimiento a partir de la traducción y se realiza una predicción a través del modelo para la detección de Spam en castellano.

Para realizar esta implementación, se declaran dos funciones adicionales que se ocupan de la detección del idioma del comentario, y de la traducción de éste al inglés si es en castellano.

```
function getLanguage(text){
    const LanguageDetect = require('languagedetect');
    const lngDetector = new LanguageDetect();
    lang = lngDetector.detect(text)[0][0];
    return lang
}

async function getTranslation(text){
    const translate = require('node-google-translate-skidz');
    const result = await translate({text: text, source: 'es', target: 'en'});
    return result['sentences'][0]['trans'];
}
```

El resto del código es similar al mostrado en el apartado anterior, pero añadiendo una condición en la que se verifica el idioma del texto, y traduciéndolo al inglés en caso de que fuera en castellano, para ello se emplean las funciones recién presentadas.

Una vez se han creado las funciones, solo resta desplegarlas para hacerlas accesibles como APIs. Para saber cómo realizar dicho despliegue, dirigirse al apartado 9.1.4 de Despliegue.

[43] PEAR:Text_LanguageDetect: https://pear.php.net/package/Text_LanguageDetect

7.2 Tecnologías

En este apartado se describe el uso de servicios y tecnologías empleadas durante el proyecto, que han requerido en mayor o menor medida, de un proceso de aprendizaje o formación de forma autónoma. Los siguientes servicios han sido mencionados a lo largo del proyecto y a continuación se muestra cómo utilizarlos con algunos fragmentos de código que sirven a modo de ejemplo.

7.2.1 Google Cloud Natural Language API

La Cloud Natural Language API permite extraer entidades a partir del texto, realizar análisis sintácticos y de opinión, y clasificar el texto en categorías.

Esta API ha sido empleada para realizar pruebas de extracción de sentimiento a partir de comentarios, tal y como se menciona en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

Para utilizarla, se comienza por habilitar la siguiente API dentro de Google Cloud Platform:

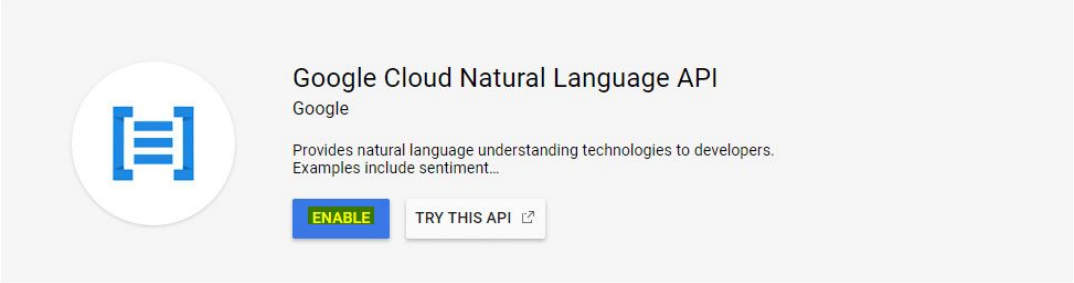

Figura 14: Habilitar Google Cloud Natural Language API.

Tras habilitar la API, se debe instalar la siguiente librería para comenzar a usarla:

```
$ pip install --upgrade google-cloud-language
```

Con el siguiente código se puede extraer el sentimiento a partir de un texto.

```
from google.cloud import language_v1
from google.cloud.language_v1 import enums
import six
from google.oauth2 import service_account

def sample_analyze_sentiment(content):
    cred = service_account.Credentials.from_service_account_file('service_account_dataretrieval.json')
    client = language_v1.LanguageServiceClient(credentials=cred)

    # content = 'Your text to analyze, e.g. Hello, world!'

    if isinstance(content, six.binary_type):
        content = content.decode('utf-8')

    type_ = enums.Document.Type.PLAIN_TEXT
    document = {'type': type_, 'content': content}

    response = client.analyze_sentiment(document)
    sentiment = response.document_sentiment
    print('Score: {}'.format(sentiment.score))
    print('Magnitude: {}'.format(sentiment.magnitude))
```

```
sample_analyze_sentiment("I'm so sad")
```

```
Score: -0.20000000298023224
Magnitude: 0.20000000298023224
```

La interpretación de resultados puede variar dependiendo del servicio, en este caso puede consultarse la documentación[44] disponible para ello.

7.2.2 VADER

VADER es una herramienta de análisis de sentimientos específicamente efectiva para la extracción de sentimientos expresados en las redes sociales.

A continuación se explica el funcionamiento de la implementación de Python de VADER, modelo empleado para la extracción de sentimiento a partir de comentarios, tal y como se explica en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

Para su instalación, debe descargarse la librería a través del siguiente comando:

```
$ pip install vaderSentiment
```

A través del siguiente fragmento puede llamarse al objeto *SentimentIntensityAnalyser*:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyser = SentimentIntensityAnalyzer()
```

Con la siguiente función se obtiene un diccionario con la forma `{"neg" : ..., "neu" : ..., "pos" : ..., "compound" : ...}`:

```
def sentiment_analyzer_scores(sentence):
      score = analyser.polarity_scores(sentence)
      print(sentence, str(score))
```

[44] Documentación para la interpretación de valores de análisis de opiniones: https://cloud.google.com/natural-language/docs/basics#interpreting_sentiment_analysis_values

Las puntuaciones *Negative*, *Positive* y *Neutral* representan la proporción del texto que corresponde a esas categorías, por lo que estas tres puntuaciones suman 1.

La puntuación *Compound* es una métrica que se basa en las tres puntuaciones anteriores y tiene un valor entre -1 (extremo negativo) y 1 (extremo positivo). Es esta métrica la empleada como *feature* en los experimentos del capítulo 5.

7.2.3 Google Cloud Translation API

La API Cloud Translation utiliza modelos ya preparados para realizar traducciones de texto en tiempo real.

La API actual se ha empleado para la detección de idioma de los comentarios, tal y como se comenta en el apartado 5.2 del capítulo Clasificación de comentarios en redes sociales.

Para utilizarla, debe instalarse primero la siguiente librería:

```
$ pip install --upgrade google-cloud-translate
```

Empleando el siguiente código, puede detectarse el idioma a partir de un texto:

```
# Imports the Google Cloud client library
from google.cloud import translate
```

```
"""Detects the text's language."""
translate_client = translate.Client.from_service_account_json(
        'service_account_dataretrieval.json')
```

```
# Text can also be a sequence of strings, in which case this method
# will return a sequence of results for each text.
text = "hey"
result = translate_client.detect_language(text)

print('Text: {}'.format(text))
print('Confidence: {}'.format(result['confidence']))
print('Language: {}'.format(result['language']))
```

```
Text: hey
Confidence: 0.7859922051429749
Language: en
```

7.2.4 FastText

FastText es una biblioteca para el aprendizaje de *word embeddings* y clasificación de textos creada por el laboratorio de investigación de IA (FAIR) de Facebook.

FastText es empleado para el entrenamiento y posterior clasificación de comentarios, tal y como se menciona en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

El primer paso es instalar la librería con el siguiente comando:

```
$ pip install fasttext
```

Tras ello, es importante que los datos del conjunto de entrenamiento estén en un *.txt* empleando la siguiente estructura para los *labels* y así el servicio diferencie cuáles son:

```
__label__0
__label__1
```

Con la siguiente línea de código Python se entrena el modelo:

```
classifier = fasttext.supervised('input_data.txt', 'model')
```

Una vez entrenado el modelo, es capaz de realizar predicciones como la siguiente, en la que clasifica un texto como Spam:

```
texts = ['if you want to make money click this link']
labels = classifier.predict(texts)
print(labels)
```

```
[['1']]
```

7.2.5 Plino

Plino es un sistema de detección inteligente de Spam, del cual se hizo uso para la clasificación de comentarios, tal y como se menciona en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

A continuación se muestra un fragmento de código para realizar predicciones de Spam dado un texto:

```
import requests
import json
import pprint
api_url = "https://plino.herokuapp.com/api/v1/classify/"
payload = \
{
'email_text': 'click this link to earn free money'
}
headers = {'content-type': 'application/json'}
# query Plino API
response = requests.post(api_url, data=json.dumps(payload), headers=headers)
pprint.pprint(response.json())
```

```
{'email_class': 'spam',
 'email_text': 'click this link to earn free money',
 'status': 200}
```

7.2.6 DatumBox

DatumBox es una API para la detección de Spam empleada en la realización de varias pruebas de clasificación de comentarios, tal y como se menciona en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

A continuación se muestra un fragmento de código para realizar predicciones de Spam dado un texto:

```
# importing the requests library
import requests
from credentials import *

# api-endpoint
URL = "http://api.datumbox.com/1.0/SpamDetection.json"

api_key = API_KEY
text = "click this link to earn free money"

# data to be sent to api
data = {'api_key':api_key, 'text':text}

# sending get request and saving the response as response object
r = requests.post(url = URL, data = data)

# extracting response text
response = r.json()
print(response['output']['result'])
```

```
spam
```

7.2.7 OOPSpam

OOPSpam es un filtro de Spam empleado en varias pruebas de clasificación de comentarios, tal y como se menciona en el apartado 5.3 del capítulo Clasificación de comentarios en redes sociales.

A continuación se muestra un fragmento de código para realizar predicciones de Spam dado un texto:

```
# importing the requests library
import requests

# api-endpoint
URL = "https://oopspam.herokuapp.com"

text = "Win money by entering this link"

# data to be sent to api
data = {'text':text}

# sending get request and saving the response as response object
r = requests.post(url = URL, data = data)

# extracting response text
response = r.json()
print(response)
```

7.2.8 Translate

Translate es una librería de Python para la traducción de texto empleada en varias pruebas de traducción de comentarios, tal y como se menciona en el apartado 5.4 del capítulo Clasificación de comentarios en redes sociales.

Primero debe instalarse la librería:

```
$ pip install translate
```

A continuación se muestra un fragmento de código para traducir un texto del castellano al inglés:

```
from translate import Translator
translator= Translator(from_lang="spanish",to_lang="english")
translation = translator.translate("Esta es una frase de prueba")
print(translation)
```

```
This is a test sentence
```

8. Pruebas

En este capítulo se describen las pruebas realizadas durante el desarrollo del proyecto. Por un lado, se realizan pruebas unitarias para comprobar el correcto funcionamiento de las funciones desplegadas como APIs. Por otro lado, se realizan pruebas de la integración Smart-Ekitaldi, donde se muestra una batería de pruebas con el resultado esperado, el resultado obtenido y la conclusión de cada prueba.

8.1 Pruebas unitarias

Para la realización de las pruebas unitarias se ha empleado Insomnia[45]. Insomnia es un cliente REST, esto es, una aplicación que permite hacer consultas a una API REST. Estos clientes REST son realmente útiles para realizar pruebas de funcionamiento de APIs en desarrollo. A continuación, se muestra cómo utilizar Insomnia:

Una vez instalado, lo primero es pulsar sobre el símbolo "+" y *New Request* para realizar una nueva petición.

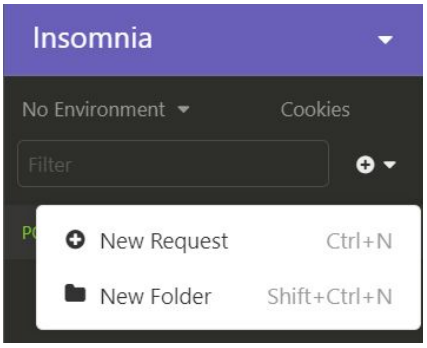

Figura 15: Interfaz de Insomnia.

Se pone nombre a la petición y se escoge *POST* como método, y *JSON* para la estructura.

[45] Sitio web Insomnia: https://insomnia.rest/

Figura 16: New Request en Insomnia.

Una vez creada la nueva petición, se copia la URL obtenida en la sección 9.1.4 del capítulo Despliegue y se pega en el apartado a la izquierda de SEND.

Tras ello, pueden escribirse peticiones siguiendo el formato especificado en la función. En este caso, ambas funciones (la que clasifica texto en inglés y la que clasifica texto en ambos idiomas) admiten la siguiente estructura:

```
{
  "account_days_old": 32,
  "followers_count": 162,
  "following_count": 152,
  "publications_count": 45,
  "text": "click here to win free money"
}
```

Se envía la petición pulsando SEND. Una vez se haya enviado, se recibe una respuesta en cuestión de segundos.

La estructura de la respuesta será similar a la siguiente, *true* si es Spam, *false* si no lo es:

```
true
```

Ahora el modelo puede ser utilizado por cualquier persona alrededor del mundo para realizar predicciones.

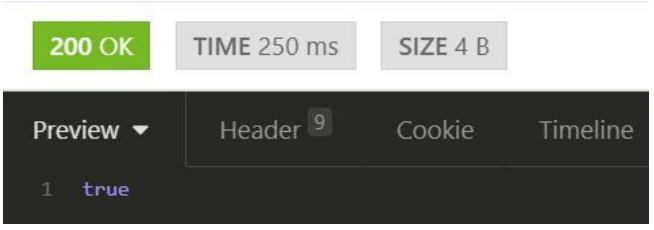

Figura 17: Tiempo de respuesta en Insomnia.

Además de las ventajas comentadas, Insomnia es una herramienta excelente para medir el tiempo de respuesta de las APIs. Una vez enviada una petición, en la parte superior derecha se muestra en el apartado *TIME*.

A continuación se muestran las pruebas unitarias realizadas empleando esta herramienta, al probar las funciones los resultados son los mismos, por lo que únicamente se expone la tabla de resultados una vez. Sin embargo, los tiempos de respuesta de cada función varían, por lo que se muestran en la dos últimas columnas, en las cuales API_en representa la función que detecta únicamente Spam en inglés, mientras que API_en_es representa la función que detecta Spam en inglés y castellano.

Datos de entrada	Datos de respuesta esperados	Datos de respuesta obtenidos	Explicación	TIME API_en	TIME API_en_es
Datos de un ejemplo que no es Spam.	200 False	200 False	Al enviar datos correctos, el formato de respuesta es el esperado.	250 ms	344 ms
Datos de un ejemplo que es Spam.	200 True	200 True			
Enviando un ejemplo eliminando un dato de entrada.	Error genérico	500 Internal Server Error	En el alcance del proyecto no se incluye la captura de excepciones.	3.94 s	157 ms
Enviando un ejemplo en el que un dato numérico es negativo.	- (ver explicación)	200 True/False	La aplicación no puede generar números negativos.	297 ms	281 ms
En vez de enviar valores numéricos, enviar texto.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos en vez de números a enviar.	422 ms	391 ms

El texto es vacío.	Error genérico	[API_en] 200 True/False [API_en_es] 500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos vacíos a enviar.	3.19 s	3.6 s
El texto es un carácter blanco.	Error genérico	[API_en] 200 True/False [API_en_es] 500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos con un solo carácter blanco a enviar.	531 ms	3.77 s
Los campos numéricos de envío son 0.	200 True/False	200 True/False	El resultado es el esperado.	343 ms	281 ms

Tabla 17: Pruebas unitarias Smart.

8.2 Pruebas integradas Smart - Ekitaldi

Las pruebas de integración entre Ekitaldi y este TFG se han realizado llamando a la función desarrollada en este TFG desde la aplicación EkitApp. La función es invocada desde la primera página de crear evento de forma habitual. Sin embargo, para realizar las pruebas se invoca a la función de manera manual al pulsar un botón y con unos parámetros concretos.

Los resultados pueden verse en la siguiente tabla, al probar ambas funciones (al igual que en el apartado anterior) se obtienen los mismos resultados, por lo que únicamente se expone la tabla de resultados una vez.

Datos de entrada	Datos de respuesta esperados	Datos de respuesta obtenidos	Explicación
Datos de un ejemplo que no es Spam.	200 False	200 False	Al enviar datos correctos, el formato de respuesta es el esperado.
Datos de un ejemplo que es Spam.	200 True	200 True	
Enviando un ejemplo eliminando un dato de entrada.	Error genérico	500 Internal Server Error	En el alcance del proyecto no se incluye la captura de excepciones.
Enviando un ejemplo en el que un dato numérico es negativo.	- (ver explicación)	200 True/False	La aplicación no puede generar números negativos.
En vez de enviar valores numéricos, enviar texto.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos en vez de números a enviar.
El texto es vacío.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos vacíos a enviar.
El texto es un carácter blanco.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos con un solo carácter blanco a enviar.
Los campos numéricos de envío son 0.	200 True/False	200 True/False	El resultado es el esperado.

Tabla 18: Pruebas integradas Smart-Ekitaldi.

9. Despliegue

Tras haber escogido el modelo y haber implementado la función, ésta debe desplegarse como API. En este capítulo se ve cómo desplegar Machine Learning en la nube, además del ciclo de vida de la solución de cara al futuro.

9.1 Machine Learning en la nube/ Cloud ML

En el capítulo actual se explica cómo desplegar un modelo de Machine Learning customizado en la nube como API, para ello, se va a emplear un servicio sobre Google Cloud Platform llamado Cloud Datalab. En dicho servicio se puede entrenar un modelo de ML basado en Python con recursos de cómputo ilimitados. Una vez se tiene un modelo definitivo, se despliega en ML Engine para poder versionarlo y analizarlo en producción. Finalmente, se expone al uso público haciéndolo disponible en una Firebase Cloud Function.

9.1.1 Cloud Datalab

En este apartado se explica cómo poner en funcionamiento Cloud Datalab[46]. Para ello, vamos a suponer que se dispone ya de una cuenta en Google Cloud Platform[47].

Dentro de GCP, primero debe habilitarse la siguiente API:

Figura 18: Habilitar Compute Engine API.

[46] Sitio web Datalab: https://cloud.google.com/datalab/?hl=es
[47] Sitio web GCP: https://cloud.google.com/?hl=es

Para habilitarla, en la parte izquierda del panel de inicio hay que dirigirse a *APIs y servicios* > *Biblioteca*.

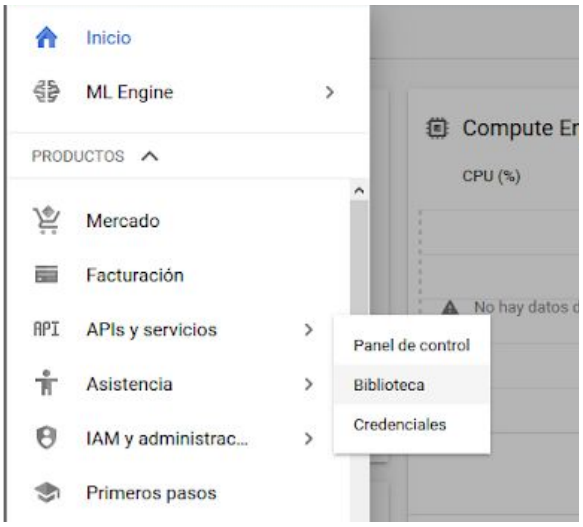

Figura 19: Habilitar APIs en Biblioteca.

En la biblioteca pueden buscarse las APIs para su habilitación.

Para poner en funcionamiento Cloud Datalab deben seguirse los siguientes pasos:

1. Se comienza por iniciar sesión en Google Cloud Platform y pulsar sobre el icono de *Activar Cloud Shell* para que ésta se despliegue.

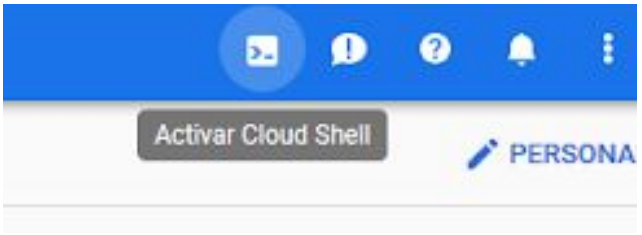

Figura 20: Activar Cloud Shell.

2. Una vez se haya abierto la consola, se ejecuta el siguiente comando para obtener el comando de gcloud más reciente:

```
$ gcloud components update
```

3. Se procede a instalar la componente datalab de gcloud con el siguiente comando:

```
$ gcloud components install datalab
```

4. Ahora ya puede crearse la instancia de datalab de la siguiente forma:

```
$ datalab create nombre-instancia-datalab
```

5. En la consola se pide introducir el número de una zona, en este caso se escogerá, por ejemplo, la 10.

6. La instancia ha sido creada. Para poder acceder a ella, se introduce el siguiente comando:

```
$ datalab connect nombre-instancia-datalab
```

7. Cuando finalice de cargar, se debe cambiar de puerto para poder usar Datalab. Para ello se debe pulsar sobre el icono que se muestra en la Figura 21 > pulsar *Cambiar puerto* > introducir el 8081 > pulsar *Cambiar y obtener vista previa*.

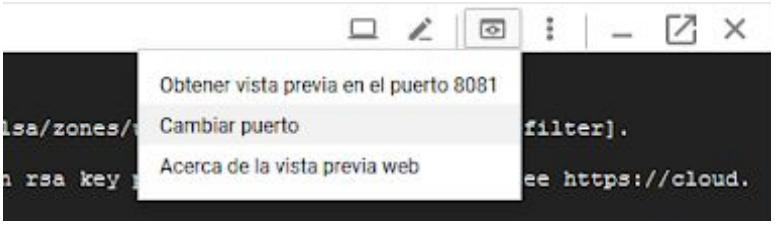

Figura 21: Cambiar puerto.

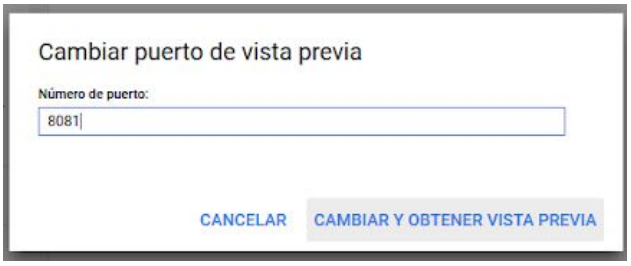

Figura 22: Obtener vista previa.

8. Finalmente, puede empezar a utilizarse Datalab como si fuera un Jupyter Notebook normal. Para agregar librerías a la instancia de Cloud Datalab debe ejecutarse una celda con el siguiente formato:

```
!conda install -y nombre-lib
```

si quiere realizarse desde conda, o

```
!pip install nombre-lib
```

si se prefiere usar pip (también puede ejecutarse este mismo comando sobre pip3, dependiendo de la versión de Python sobre la que se ejecute el Notebook).

Para más información, consultar la documentación[48].

[48] Documentación Cloud Datalab: https://cloud.google.com/datalab/docs/how-to/adding-libraries

9.1.2 Guardar modelo en Firebase Storage

Tras crear el modelo, entrenarlo y testearlo debe desplegarse en ML Engine[49] para poder versionarlo y analizarlo en producción. Para ello, el modelo debe almacenarse en *Firebase Storage*, para que luego ML Engine pueda utilizarlo en la siguiente sección. Esta tarea se realiza ejecutando en el mismo Notebook en el que se encuentra el modelo, los siguientes fragmentos de código:

Para poder utilizar la cuenta de Firebase:

```
import firebase_admin
from firebase_admin import credentials
from firebase_admin import firestore

# Use a service account
if (not len(firebase_admin._apps)):
      cred = credentials.Certificate(r'service_account.json')
      firebase_admin.initialize_app(cred)

db = firestore.client()
```

Para el siguiente fragmento de código se necesita el id del proyecto. Puede encontrarse iniciando sesión en Firebase > escogiendo el proyecto > *Configuración del proyecto.*

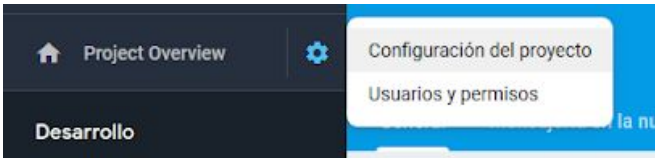

Figura 23: Configuración del proyecto.

Allí se encuentra un apartado que indica cuál es el *ID del proyecto*.

```
from sklearn.externals import joblib
from firebase_admin import storage

joblib.dump(clf, 'model.joblib')
bucket = storage.bucket(name='id-proyecto-gcp.appspot.com')
b = bucket.blob('model-v1/model.joblib')
b.upload_from_filename('model.joblib')
print('model uploaded!')
```

Ahora podemos comprobar que el archivo se ha creado con éxito verificando en *Firebase > Storage > directorio especificado anteriormente* que el modelo se encuentra exitosamente almacenado.

[49] Sitio web ML Engine: https://cloud.google.com/ml-engine/?hl=es-419

9.1.3 Desplegar modelo en ML Engine

Ahora que el modelo está guardado en *Google Cloud Storage* puede conectarse a ML Engine.

Para ello, se debe comenzar por habilitar las siguientes APIs:

Figura 24: Habilitar Cloud Machine Learning Engine.

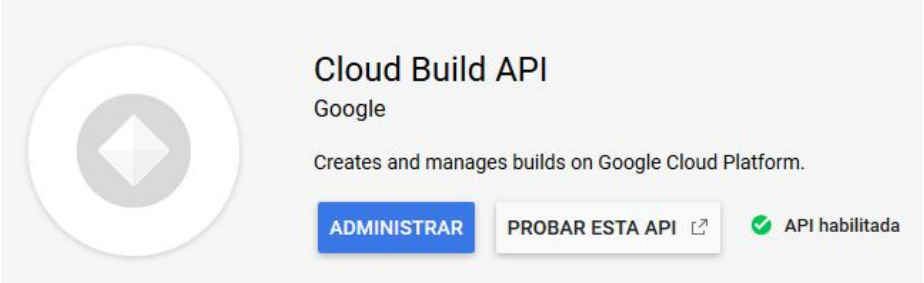

Figura 25: Habilitar Cloud Build API.

Después hay que dirigirse a *ML Engine* > *Modelos*.

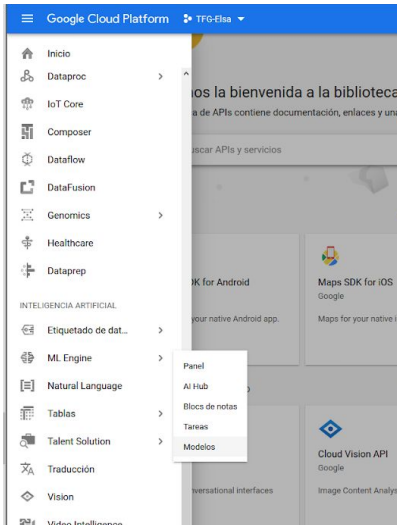

Figura 26: Modelos en ML Engine.

Una vez dentro, se pulsa sobre *Nuevo modelo* y se introducen los datos. Tras ello, se crea una versión de este modelo que apuntará al archivo *.joblib* guardado con anterioridad.

Hacer click sobre el modelo, pulsar sobre *Nueva versión* y proceder a rellenar los datos. Es importante que la versión de Python escogida sea la misma que se utilizó para entrenar el modelo. En framework en este caso se escoge scikit-learn ya que ha sido el empleado. A la hora de especificar la versión del framework, si no se está seguro puede comprobarse ejecutando el siguiente fragmento en el Notebook.

```
import sklearn

print('The scikit-learn version is {}.'.format(sklearn.__version__))
```

A la hora de escoger la *Versión de tiempo de ejecución de ML* debe escogerse la recomendada, actualmente es la 1.9. Se deja el tipo de máquina por defecto y se especifica la ruta al archivo *.joblib*. Para ello, se pulsa sobre *Browse* y se selecciona la carpeta en la cual se encuentra el archivo. Es importante seleccionar la carpeta, no el archivo. El resto de campos se dejan tal y como están por defecto y se guarda. En ese momento se desplegará la instancia del modelo en ML Engine.

9.1.4 Desplegar función como API

Una vez se ha creado la función que accede al modelo, solo resta subir la función a Firebase para desplegarla como API.

Para subir la función a Firebase, se ejecuta el siguiente comando:

```
$ firebase deploy --only functions
```

Una vez haya terminado de cargarse, se obtiene una URL a través de la cual estará accesible la función, la cual puede encontrarse iniciando sesión en Firestore, en el apartado *Functions*, debajo de *Solicitud* en letra más pequeña. Ekitaldi puede acceder a esta función y hacer uso de ella. La integración se ha realizado con éxito tal y como se muestra en el acceso a las funciones desde la aplicación en el apartado 6.1.2 de Implementación de la memoria Ekitaldi y en las pruebas realizadas disponibles en el apartado 8.2 de Pruebas.

9.2 Ciclo de vida

Una vez la solución ha sido desplegada, el producto ya está disponible para utilizarse desde Ekitaldi. Sin embargo, esta solución no es definitiva debido a que el proyecto en sí mismo forma parte de una prueba de concepto de algo mayor, un emprendizaje, por lo que se espera que sea continuamente mejorada. Por ello, es importante planificar un ciclo de vida de la solución que permita dichas mejoras una vez la red social EkitApp se encuentre en explotación y genere datos.

Una vez llegada esa fase del emprendizaje, los datos generados por el uso de la aplicación serán empleados para entrenar los modelos creados (o sus versiones mejoradas) de modo que las

predicciones se acercarán mucho más al contexto real en el que se desenvuelve la red social, obteniendo así unos resultados dirigidos al uso de ésta.

Esta extrapolación de los modelos a nuevos datos es realmente sencilla gracias a la arquitectura diseñada en este proyecto. Únicamente deben de pasarse los datos existentes en Firestore a CSV[50], tal y como se muestra en el apartado 9.2.1, tras ello, se emplean los datos obtenidos para entrenar el modelo, y finalmente se crea una nueva versión del modelo actual en ML Engine, de esta forma, no debe crearse una nueva función API, si no que el modelo se actualiza automáticamente para ser accesible a través de la misma, y así no se interrumpiría la experiencia de usuario durante las actualizaciones del modelo.

9.2.1 Desde Firestore a CSV

Tal y como se ha comentado, en el futuro se pretende emplear los datos generados por EkitApp en explotación para entrenar los modelos, para ello, en este apartado se explica cómo extraer los datos desde Firestore e introducirlos en un CSV.

El primer paso es importar las librerías correspondientes y disponer de credenciales en GCP[51].

```
import firebase_admin
from firebase_admin import credentials
from firebase_admin import firestore

cred = credentials.Certificate('service-account.json')
app = firebase_admin.initialize_app(cred)
db = firestore.client()
```

Después debe de especificarse la colección de la cual quieren extraerse los documentos, en este caso *comments*:

```
comm = db.collection('comments')
docs = comm.get()
```

Tras ello, solo resta recorrer todos los documentos extraídos e insertarlos en un DataFrame de Pandas, desde el cual finalmente se introducen en un CSV:

```
data = []
for doc in docs:
   data.append(doc.to_dict())
   df = pd.DataFrame(data)

df.to_csv('data.csv', index=False)
```

[50] Los archivos CSV (del inglés comma-separated values) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla.

[51] Más información sobre Service Accounts en https://cloud.google.com/iam/docs/service-accounts.

10. Gestión del proyecto

En el capítulo actual se muestra la trayectoria del proyecto y cómo ésta ha sido gestionada. Se realiza un análisis de la gestión del alcance y los principales cambios que ha sufrido durante el desarrollo del proyecto respecto a la versión original. Por otro lado, se realiza una revisión de la gestión de la colaboración, pilar fundamental de este proyecto, además del número de horas dedicadas a cada uno de los paquetes de trabajo.

10.1 Gestión del alcance

Al comienzo del proyecto se estableció un alcance inicial, el cual ha ido variando durante el desarrollo del proyecto. El desconocimiento de las herramientas empleadas y la inexperiencia en proyectos de esta magnitud en el campo de la ciencia de datos ha propiciado que las estimaciones de dedicación por paquete no fueran realistas.

El objetivo de este proyecto era proporcionar funciones adicionales relacionadas con la analítica de datos a la red social EkitApp. Debido a que las redes sociales actualmente incluyen numerosos funcionalidades inteligentes y que la idea del proyecto abarca más de lo factible en un Trabajo de Fin de Grado, el alcance inicial fue acotado al tiempo y recursos disponibles. Sin embargo, algunas funcionalidades realizadas durante este proyecto han excedido el alcance inicial; entre ellas se encuentran desarrollar una función adicional en la que se combinan dos clasificadores permitiendo la detección de Spam multi-idioma, la brevedad en el tiempo de respuesta de las funciones desplegadas como API, la sencillez con la que puede extrapolarse el trabajo realizado a datos reales generados en un futuro por la red social cuando se encuentre en explotación.

Por otro lado, por la peculiaridad de tratarse de un Trabajo de Fin de Grado colaborativo, el principal riesgo a destacar de este proyecto es que la aplicación Ekitaldi no fuera funcional y finalmente no pudieran realizarse las pruebas de integración de ambas partes. Al comienzo del proyecto se establecieron el alcance y los objetivos de forma que, si la integración entre ambas partes no pudiera llegar a realizarse, ésta no afectara a ninguno de los proyectos. Finalmente, la integración ha sido realizada con éxito, tal y como se comenta en el apartado 9.1.4 de Despliegue, y se han realizado pruebas de dicha integración (apartado 8.2 de Pruebas).

10.2 Gestión de la colaboración

En la actualidad son escasos los contextos en los cuales un proyecto se desarrolla sin la colaboración de miembros pertenecientes a distintas áreas del conocimiento. Saber cooperar e integrar distintos enfoques en un mismo producto es fundamental a la hora de realizar proyectos a gran escala. Este proyecto ha sido realizado con la colaboración de dos Trabajos de Fin de Grado: el de Julen Miner (Ekitaldi) y el presente (Smart). La comunicación entre los dos interesados colaboradores del proyecto ha sido un aspecto importante a tener en cuenta a lo largo del desarrollo de éste. Para gestionar la comunicación de manera adecuada el sistema de información compartido (ver punto 10.2.1) ha sido fundamental.

Numerosas partes del proyecto requerían trabajar de forma conjunta, en estos casos se ha procurado colaborar simultáneamente y de manera presencial o mediante videoconferencia para así avanzar de la forma más ágil y efectiva. En aquellos casos en los que no ha podido ser así, se ha comunicado o consultado a la otra parte toda propuesta y modificación realizada.

Adicionalmente, se ha realizado un seguimiento contínuo de ambos proyectos para así adoptar una perspectiva de proyecto conjunto que persigue un objetivo común. Esto ha facilitado la colaboración y propuesta de ideas a la otra parte que ha resultado en una mayor integración de ambos desarrollos.

10.2.1 Sistema de información

Con el el objetivo de facilitar la colaboración, creación de contenido conjunto y el acceso a ambas partes del proyecto se ha empleado un sistema de información conjunto con Ekitaldi que permite el almacenaje de datos, documentación y otros activos generados durante el desarrollo del Trabajo de Fin de Grado.

Para el diseño y organización del sistema de información, nos hemos apoyado en la Estructura de Descomposición del Proyecto, la cual puede verse en la sección 3.3. Esto nos permite ubicar de forma sencilla el contenido que generado en cada paquete del EDT. Google Drive es el servicio que da respaldo a nuestro sistema de información compartido, ya que proporciona una gran integración de múltiples servicios, como el de edición de documentos o de tablas de cálculo, además de comodidad a la hora de compartir documentos. A continuación se muestra la estructura de directorios (Figura 27) con las correspondientes descripciones:

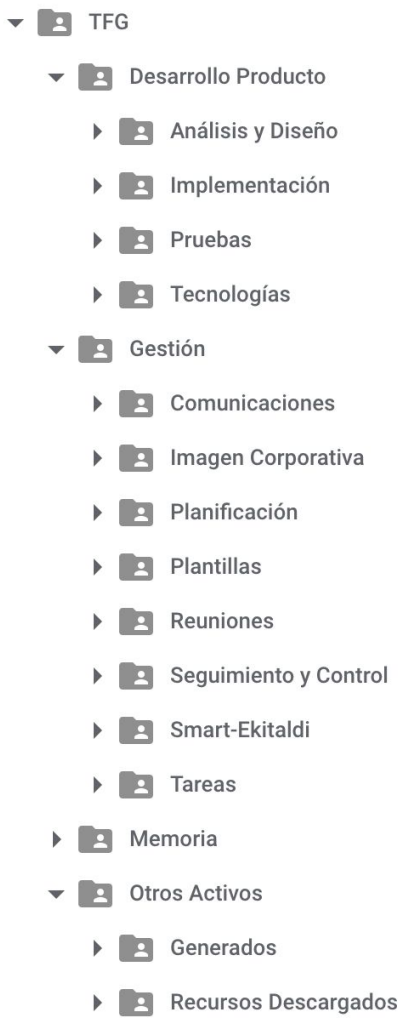

Figura 27: Esquema del sistema de información conjunto.

- **Raíz**: se divide en los siguientes directorios: *Desarrollo Producto*, *Gestión*, *Memoria* y *Otros Activos*.
- **Desarrollo Producto**: contiene toda la información relativa al producto y a su desarrollo. Este nodo se subdivide a su vez en cuatro secciones principales: *Análisis y Diseño*, *Implementación*, *Pruebas* y *Tecnologías*.
- **Análisis y Diseño**: incluye toda la información relativa al análisis previo al desarrollo del producto y a su diseño. La información relativa al producto que no sea parte del propio producto se encuentra en este nodo.
- **Implementación**: se encuentran exclusivamente los activos que forman parte del producto.
- **Tecnologías**: contiene los documentos relacionados con las tecnologías empleadas en el producto, así como los pasos seguidos para sus implementaciones.

- **Gestión**: incluye toda la información en relación a la gestión del proyecto. Este nodo se subdivide a su vez en nueve secciones principales: *Comunicaciones*, *Económica*, *Imagen Corporativa*, *Planificación*, *Plantillas*, *Reuniones*, *Seguimiento y Control*, *Smart-Ekitaldi* y *Tareas*.
- **Comunicaciones**: se encuentra toda la información relativa a las comunicaciones realizadas.
- **Imagen Corporativa**: se encuentran elementos de la imagen corporativa como el logo y los encabezados.
- **Planificación**: incluye toda la información relacionada con la planificación del proyecto.
- **Plantillas**: contiene los documentos que sirven como plantillas, para así estandarizar la creación de documentación entre los dos Trabajos de Fin de Grado (Smart-Ekitaldi). Se encuentran las plantillas de documentos normales además de las actas de las reuniones.
- **Reuniones**: se encuentra toda la información relativa a las reuniones realizadas a nivel interno, tanto a nivel directivo, como de equipo.
- **Seguimiento y Control**: contiene toda la información relativa al monitoreo y control del proyecto (tiempos de trabajo, cumplimiento de plazos y objetivos...).
- **Smart-Ekitaldi**: incluye los documentos de integración de ambos proyectos, así como los acuerdos tomados durante el desarrollo.
- **Tareas**: se encuentra toda la información relacionada con las tareas que se llevan a cabo en el proyecto, así como el estado de las mismas.
- **Memoria**: contiene los documentos que formarán parte de la memoria.
- **Otros Activos**: incluye todos los activos que se han obtenido o generado durante el proyecto que por sí mismos no forman parte del producto pero que tienen valor o pueden tenerlo en un futuro. Este nodo se subdivide a su vez en dos secciones principales: *Generados* y *Recursos Descargados*.
- **Generados**: se encuentra todo activo que se haya generado durante el proyecto que no pertenezca al producto ni a las tecnologías utilizadas.
- **Recursos Descargados**: contiene todo activo de utilidad descargado que no forma parte del producto.

Para diferenciar los documentos que pertenecen a este trabajo y los que pertenecen a Ekitaldi dentro del sistema de información compartido, se sigue el siguiente comportamiento dependiendo del volumen de documentos dentro de cada directorio:

- Si hay 4 documentos o menos, se diferencian escribiendo en el nombre de cada archivo ".X", siendo X la letra J si el archivo forma parte de Ekitaldi o la letra E si el archivo forma parte de Smart. Por ejemplo, el documento de Antecedentes de Smart se llamaría *TFG.Antecedentes.E*.
- Si hay más de 4 documentos, el directorio se subdivide en dos, uno con el nombre Smart y otro con el nombre Ekitaldi, en el que irán los archivos de cada una de las partes.

Además, el encabezado de cada página dentro de los documentos contiene el nombre del autor.

10.3 Dedicaciones

Durante el desarrollo del proyecto se ha llevado la contabilidad de las horas dedicadas a cada paquete definido en la Estructura de Descomposición del Proyecto. El resultado de este seguimiento puede verse reflejado en la Tabla 18.

Paquete de trabajo	Horas dedicadas
P.Gestión.**Planificación**	8
P.Gestión.**SeguimientoYControl**	8
P.Gestión.**IntegraciónSmart-Ekitaldi**	15
P.Producto.**Tecnologías**	20
P.Producto.AnálisisYDiseño.**FuncionesAPI**	6
P.Producto.AnálisisYDiseño.ClasificaciónSpam.**TrabajosAnteriores**	20
P.Producto.AnálisisYDiseño.ClasificaciónSpam.**Dataset**	81
P.Producto.AnálisisYDiseño.ClasificaciónSpam.**MétodosDeClasificación**	38
P.Producto.AnálisisYDiseño.ClasificaciónSpam.**Selección**	14
P.Producto.AnálisisYDiseño.**BaseDeDatos**	3
P.Producto.Implementación.**Documentación**	2
P.Producto.Implementación.**FuncionesAPI**	11
P.Producto.Implementación.**IntegraciónIA-API**	16
P.Pruebas.**PruebasSmart-Ekitaldi**	3
P.Pruebas.**PruebasFuncionesAPI**	2
P.Académico.**Defensa**	20
P.Académico.**Memoria**	112
Dedicación total:	**379**

Tabla 19: Dedicaciones a cada paquete de la EDT.

Estas dedicaciones han sido contabilizadas desde enero de 2019 hasta junio de 2019, siendo la memoria el paquete de trabajo que más dedicación ha precisado. Por otro lado, la creación de datasets ha llevado un tiempo mayor al esperado, debido a que requiere de un análisis y diseño

exhaustivo, además de largos tiempos de espera en el caso de ejecutar modificaciones sobre datasets de gran tamaño, los cuales a menudo provocan complicaciones en tiempo de ejecución.

A continuación se muestra una gráfica que permite visualizar las diferencias en la dedicación a los distintos paquetes de trabajo.

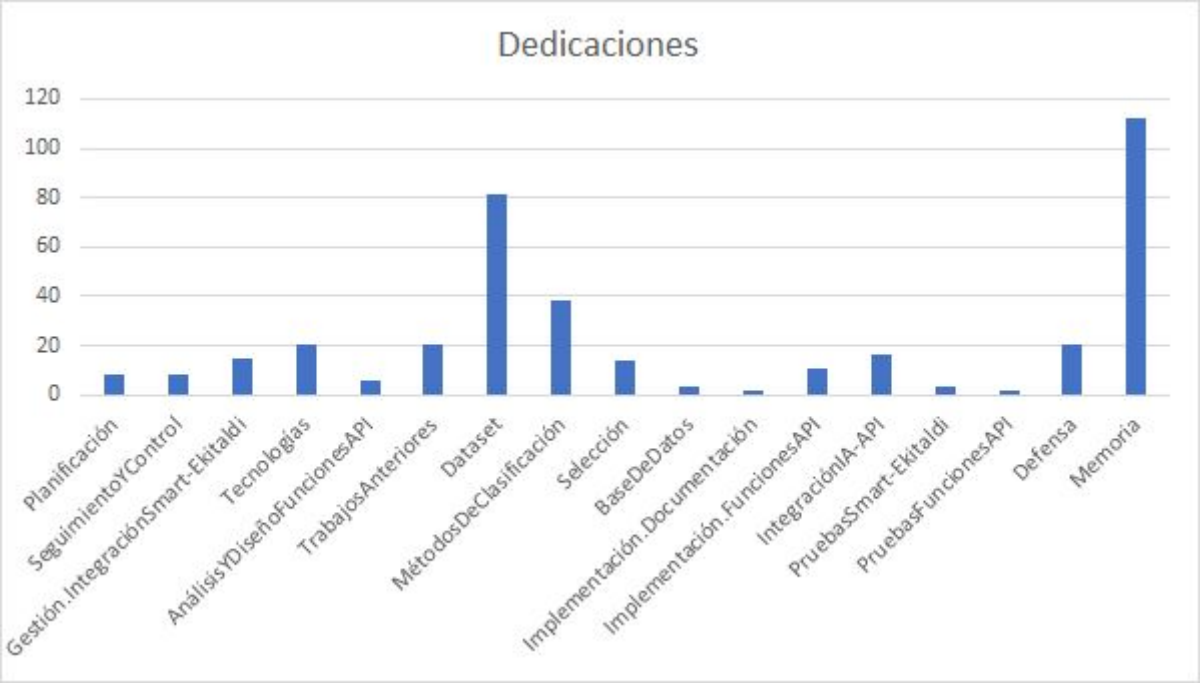

Figura 28: Gráfico de dedicaciones.

11. Conclusiones

En esta sección se encuentran agrupadas las conclusiones extraídas tras la realización de este proyecto. Entre ellas, se comenta la utilidad de las herramientas empleadas y su posible integración en el programa universitario, y pretenden ser, además, un anticipo de las líneas de continuación de este proyecto.

11.1 Tecnologías y metodologías utilizadas

En este proyecto se han empleado tecnologías tan novedosas como los servicios en la nube de Google: Cloud Translation API, Natural Language API, Firebase, Datalab, ML Engine. El uso de estas tecnologías ha permitido adquirir una perspectiva actual de las tecnologías disponibles en el sector de la analítica de datos. Por otro lado, se han empleado servicios open-source, entre los que destaca VADER por su alto rendimiento en el análisis de sentimiento en textos de redes sociales.

ML Engine junto con Firebase ha simplificado la labor del despliegue de modelos de Machine Learning en la nube y ha permitido adoptar una metodología de trabajo profesional de cara al futuro. Un claro ejemplo es el versionado de modelos. Cuando las funciones API que dan acceso al modelo se encuentran en estado de explotación es importante cuidar la metodología empleada para realizar actualizaciones en el modelo o en las propias funciones. Esta tarea se ve enormemente facilitada al emplear las herramientas recién mencionadas, ya que basta con crear una nueva versión del modelo en ML Engine y, de esta forma, no debe crearse una nueva función API, sino que el modelo se actualiza automáticamente para ser accesible a través de la misma, y así no interrumpir la experiencia de usuario durante las actualizaciones del modelo.

De cara a la colaboración Smart-Ekitaldi ha resultado ser un acierto el optar por utilizar las tecnologías de Google en la nube. El utilizar este ecosistema ha propiciado un flujo de trabajo integrado y eficiente. Esta decisión ha contribuido a la integración exitosa de ambos proyectos, logrando así una colaboración efectiva y funcional.

11.2 Problema abordado y solución dada

Una de las principales problemáticas de seguridad dentro de las redes sociales es la detección de Spam en tiempo real. Durante los últimos años los ataques de Spam han cambiado su entorno más habitual de los correos y SMS por uno cada vez más utilizado por la gente de a pie, las redes sociales. A pesar de que numerosos estudios han tratado de solventar esta problemática, todavía no se ha alcanzado un consenso que determine la técnica más efectiva para realizar este filtrado en tiempo real.

En este TFG, se desarrolla un filtro de comentarios Spam en tiempo real, el cual se integra exitosamente con EkitApp, una red social de eventos universitarios fruto de la idea de un emprendizaje en colaboración con Julen Miner. Esta solución combina algunas de las técnicas más efectivas halladas hasta el momento, y proporciona un enfoque práctico y completo de lo que significa realizar un producto en el sector del Data Science, desde la recogida de datos hasta su despliegue posibilitando su uso desde cualquier parte del mundo.

Además, se ha logrado implementar una función adicional en la que se combinan dos clasificadores permitiendo la detección de Spam multi-idioma. Por otro lado se ha alcanzado brevedad en el tiempo de respuesta de las funciones desplegadas como API, y finalmente, se ha obtenido un sistema que permite extrapolar con sencillez el trabajo realizado a datos reales generados en un futuro por la red social cuando se encuentre en explotación.

Cabe destacar las dificultades encontradas durante el análisis de datasets públicos, ya que en la mayor parte de casos los datasets disponibles con relación al Spam son de correos o SMS, los entornos en los que hasta el momento ha sido más frecuente encontrar este tipo de contenido. Sin embargo, no es posible emplear estos mismos datasets para la detección de Spam en redes sociales debido a que el formato del contenido y el tipo de vocabulario empleado en este marco es distinto. Por otro lado, los estudios realizados entorno a la detección de Spam en redes sociales a menudo no publican los datasets empleados, lo cual dificulta el progreso en esta área de la seguridad cada vez más comprometida.

También se han encontrado dificultades a la hora de hallar servicios para la extracción de sentimiento a partir de texto en español. Tras probar las escasas herramientas que prometen un análisis de sentimiento en este idioma, se ha determinado que actualmente no existe una herramienta o servicio que pueda realizar esta tarea con eficacia y que para garantizar la seguridad ante contenido Spam en redes sociales será necesario desarrollar modelos específicos por cada idioma, o decantarse por técnicas alternativas cuyo coste de obtención sea menor.

Tras la realización de los distintos modelos en los que se ha experimentado con varios algoritmos, se ha concluido que para este tipo de problemas Random Forest y Decision Tree obtienen mejores resultados, mientras que, por otro lado, Naive Bayes no es un algoritmo apropiado debido a su bajo rendimiento. Este razonamiento podría estar justificado además por los resultados obtenidos con Plino, servicio para la clasificación cuyo modelo es una implementación customizada de Naive Bayes, ya que al emplear dicho servicio como *feature* adicional no se ha logrado mejora en los resultados. Estos resultados podrían asimismo atribuirse a los datos con los que ha sido entrenado dicho modelo, ya que ha sido entrenado con correos y la forma en la que se escriben los correos es muy

distinta de la forma en la que se escribe en la redes sociales. Por lo que el no estar optimizado para comentarios de redes sociales, puede traducirse en resultados peores.

En cuanto al modelo seleccionado respecta, para las predicciones de Spam en inglés se ha superado los resultados propuestos en Ezpeleta[3] reduciendo el FPR al mismo tiempo que aumentando el accuracy un 13%.

11.3 Respecto al grado

Durante el grado se ha tenido la oportunidad de cursar varias asignaturas que han sido de ayuda para la realización de este proyecto. Una de ellas es MLNN (Machine Learning and Neural Networks), aún siendo una asignatura optativa se aprenden conceptos de gran importancia relacionados con la analítica de datos, que me han facilitado el proceso de investigación y comprensión de artículos científicos. Además, en esta asignatura se aprende a utilizar librerías de gran utilidad como lo son SKLearn y Pandas, las cuales han sido empleadas para la creación de los modelos en este proyecto, y se explica el funcionamiento de Jupyter Notebook[52], del cual hice uso mediante Google Datalab.

Por otro lado, con el conocimiento adquirido en el grado sobre bases de datos SQL y la información encontrada en internet se ha podido diseñar la base de datos noSQL en Firestore, a pesar de que este tipo de bases de datos no se enseñe en el grado de forma explícita.

Adicionalmente, la asignatura de Sistemas Web ha contribuido a realizar un correcto control de versiones a través de la herramienta Git.

Finalmente la asignatura de Gestión de proyectos, ha sido de gran ayuda para adoptar un sistema de información común con TFG-Julen, además de gestionar correctamente la cohesión de ambos proyectos, sea en las partes individuales, que en las de integración.

11.4 ML en la nube, posibilidades de inclusión

A menudo en la asignatura de MLNN debían entrenarse modelos que tomaban en ocasiones varias horas hasta terminar, esto ralentiza mucho el ritmo de trabajo y es un problema que puede encontrarse frecuentemente si se sigue trabajando en el campo del Data Science, especialmente entrenando redes neuronales con datos de gran tamaño. Para evitar esa inversión de tiempo y dotar al alumnado de herramientas para eventuales situaciones similares hubiera sido de gran utilidad incluir en el temario el entrenamiento de modelos en la nube a través de máquinas virtuales con muchos más recursos que las que se tienen normalmente al alcance.

Además, otro aspecto interesante y de utilidad a enseñar es cómo desplegar los modelos creados en clase como APIs para que cualquier persona pueda hacer uso de ellos, ya que se incide en la teoría, pero no tanto en la practicidad de lo que se crea en clase.

[52] Jupyter Notebook es un entorno de trabajo interactivo que permite desarrollar código en Python

11.5 Futuro de SMART y del clasificador de Spam

En el futuro, Smart será mucho más que un filtro de Spam, se hará cargo de todas aquellas funcionalidades que requieran de inteligencia, como por ejemplo un sistema de recomendaciones, un filtro de contenido sensible, un calculador de puntuación de perfiles inteligente...

Respecto al clasificador, éste contará con un modelo customizado para la detección de sentimientos en castellano y se agregarán los idiomas más populares a la función multi-idioma para así continuar mejorando la seguridad de la red social. Sin embargo, debido al coste que supone la implementación de modelos para la detección de sentimientos en numerosos idiomas, se continuará investigando *features* alternativos a añadir al modelo cuyo coste de obtención sea menor.

Además, en el futuro los modelos serán entrenados con los datos reales de la red social y por lo tanto se ofrecerán predicciones enfocadas al contenido generado en EkitApp.

Por otro lado, una línea de continuación a través de la cual podría mejorarse los resultados del filtro, es mediante el análisis de los enlaces incluidos en los comentarios. A menudo los comentarios Spam van acompañados de un enlace malicioso, por lo que el comprobar si éstos figuran en alguna lista negra[53] de enlaces maliciosos facilitaría la identificación de este tipo de comentarios.

[53] Google Safe Browsing: https://transparencyreport.google.com/safe-browsing/search

12. Referencias bibliográficas

[1] C. Chen, J. Zhang, X. Chen, Y. Xiang and W. Zhou, "6 million spam tweets: A large ground truth for timely Twitter spam detection," 2015 IEEE International Conference on Communications (ICC), London, 2015, pp. 7065-7070.

[2] H. Gupta, M. S. Jamal, S. Madisetty and M. S. Desarkar, "A framework for real-time spam detection in Twitter," 2018 10th International Conference on Communication Systems & Networks (COMSNETS), Bengaluru, 2018, pp. 380-383.

[3] Ezpeleta E., Iturbe M., Garitano I., de Mendizabal I.V., Zurutuza U. (2018) A Mood Analysis on Youtube Comments and a Method for Improved Social Spam Detection. In: de Cos Juez F. et al. (eds) Hybrid Artificial Intelligent Systems. HAIS 2018. Lecture Notes in Computer Science, vol 10870. Springer, Cham

[4] c. Yang, R. Harkreader, and G. Gu, "Empirical evaluation and new design for fighting evolving twitter spammers," Information Forensics and Security, IEEE Transactions on, vol. 8, no. 8, pp. 1280-1293,2013.

[5] Ezpeleta, E., Zurutuza, U., Gómez Hidalgo, J.M.: Does sentiment analysis help in Bayesian spam filtering? In: Martínez-Álvarez, F., Troncoso, A., Quintián, H., Corchado, E. (eds.) HAIS 2016. LNCS (LNAI), vol. 9648, pp. 79–90. Springer, Cham (2016).

[6] C. Grier, K. Thomas, Y. Paxson, and M. Zhang, "@spam: the underground on 140 characters or less," in Proceedings of the 17th ACM conference on Computer and communications security, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 27-37.

[7] Hutto, C.J. & Gilbert, E.E. (2014). VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text. Eighth International Conference on Weblogs and Social Media (ICWSM-14). Ann Arbor, MI, June 2014.

[8] C. Lee and Y. Zheng, "SQL-to-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems," 2015 IEEE International Conference on Systems, Man, and Cybernetics, Kowloon, 2015, pp. 2022-2026.

13. Anexos

Anexo 1: Diagrama de la base de datos

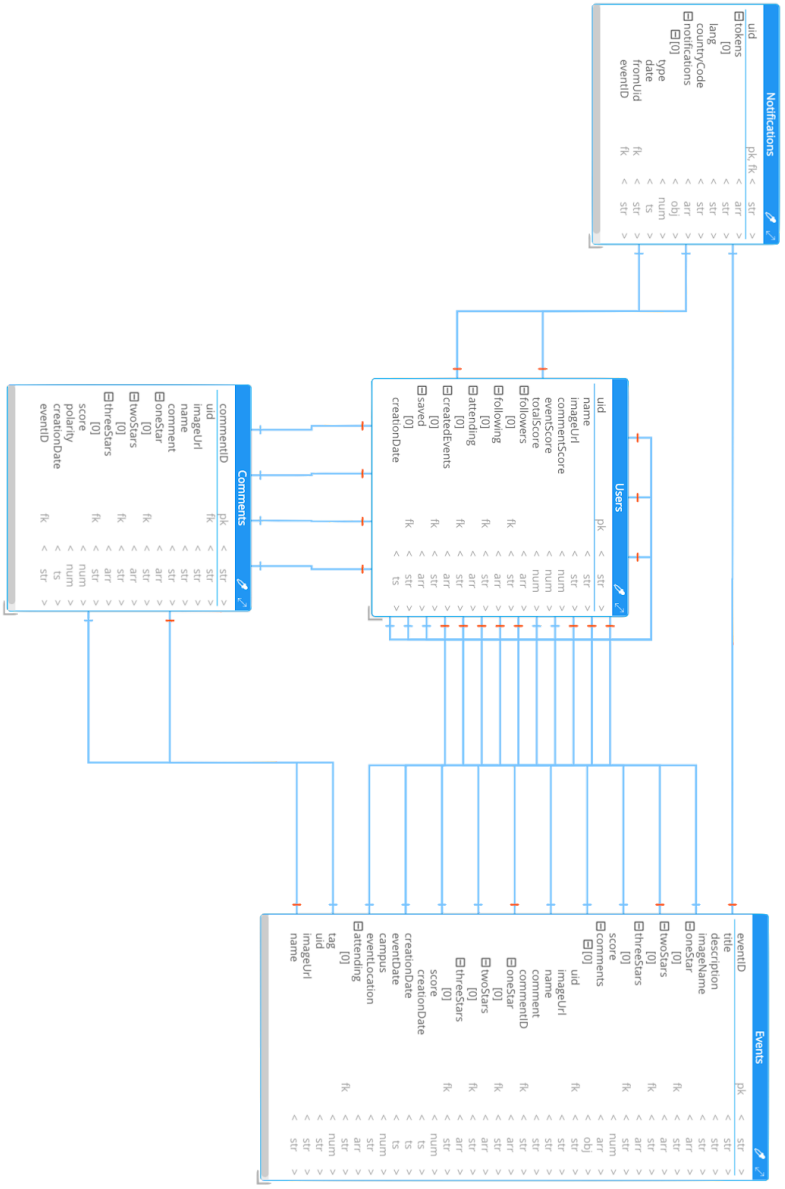

Figura 29: Diagrama de la base de datos.