Visibilizando injusticias laborales publicadas en Twitter a través de una plataforma descentralizada basada en blockchain

Giving visibility to workplace injustices published on Twitter via a decentralized blockchain-based platform

TRABAJO DE FIN DE GRADO

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Roberto Asunción López Pablo Imbert Fernández
Julián Moreno Bellaneda Raquel Pérez González de Ossuna

Doble Grado en Ingeniería Informática y Matemáticas

Javier Mulero Martín Ángela Ruiz Ribera

Departamento de Ingeniería del Software e Inteligencia Artificial

Directores: Samer Hassan Collado y Jorge Saldívar

Curso académico 2021-2022

Convocatoria de Junio

Distribución de la memoria e imágenes del proyecto

Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)

Usted es libre de:

- Compartir: copiar y redistribuir el material en cualquier medio o formato
- Adaptar: remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.

Bajo los siguientes términos:

- Atribución: Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante.
- CompartirIgual: Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la la misma licencia del original.

Distribución del código del proyecto

GNU General Public License v3.0

Todo el código desarrollado en este proyecto se distribuye bajo la licencia GPL-3.0.

Resumen

En nuestro día a día, se producen numerosas situaciones de **injusticia** en el **entorno laboral**. Sin embargo, muchas veces, los medios tradicionales de denuncia no son suficiente para visibilizarlas. En los últimos años, con la aparición y expansión de las **redes sociales**, muchas personas han recurrido a ellas para hacerse oír. En particular, Twitter ha jugado un papel muy importante a la hora de visibilizar iniciativas y unir a gente con intereses comunes.

En este contexto, percibimos que, aún significando un avance importante, pueden existir **problemas** si se usan exclusivamente las redes sociales como medio de denuncia. Por ejemplo, la denuncia puede ser censurada por medio de su eliminación, denuncias muy mediáticas pueden eclipsar a otras, o las denuncias pueden quedar ocultas entre la gran cantidad de contenido que hay en la red social.

Es por ello que proponemos crear una herramienta de **análisis** y **visualización** dedicada, exclusivamente, a este tema. La aplicación *Injustweet*, haciendo uso de estadísticas y gráficas, muestra datos relacionados con denuncias laborales en Twitter. De forma que pueda servir como un punto de **conocimiento**, **apoyo** y **denuncia** para personas que estén interesadas, tanto a nivel personal como profesional, en el tema.

El desarrollo de la aplicación aprovecha los últimos avances **metodológicos** y **tecnológicos** para llevarla a cabo. Por un lado, el desarrollo del *frontend* se lleva a cabo cuidando el **diseño** gracias a *React*. Por otro lado, con el objetivo de evitar la **censura**, el *backend* hace uso de tecnologías distribuidas, como la *blockchain*. Finalmente, destacamos que con el fin de detectar, entre la gran cantidad de **datos**, aquellas que sean denuncias, se aplican técnicas de análisis de lenguaje natural.

Palabras clave: blockchain, dashboard, análisis de lenguaje natural, denuncias laborales, Twitter, MERN, Python.

Abstract

There has always been different kinds of **injustice** in the **workplace**. However, many times, it is hard to be heard when using the traditional ways of launching complaints. During the past years, due to the rapid growth of **social media**, many have started using them as a way of communicating those injustices. In particular, Twitter has played a very important role. It has become a platform that gives visibility to many initiatives and where people can get together and share common interests.

Even though social networks have partially solved the **problem**, they alone do not seam to be the best solution. For instance, a report made in Twitter can be censored by deleting the tweet. The denunciations can be eclipsed by the gigantic amount of data or by others which get more spotlight.

That is way we propose to develop an application dedicated, exclusively, to this topic. *Injustweet* will show data related to workplace injustices by using graphs and statistics. Therefore, everyone who is interested personally or professionally in those situations could use it as a way of **consulting** and **sharing** information.

During the development of this project, we have used the latest **technologies** and **methodologies** available. On one hand, we were able of creating an attractive **design** and developing the frontend thanks to *React*. On the other hand, we used *blockchain* as a way of avoiding **censorship**. Finally, it should be noted that we have used language processing techniques to obtain **data** related to workplace injustices.

Key words: blockchain, dashboard, language processing, workplace complaints, Twitter, MERN, Python.

Agradecimientos

Queremos dedicar unas palabras para agradecer a todos aquellos que nos han acompañado y apoyado, no sólo durante el desarrollo de este TFG, sino a lo largo de toda la carrera.

En primer lugar, nos gustaría agradecer a *Samer Hassan* y *Jorge Saldívar*, codirectores del TFG, por su motivación y ayuda. Nos ha permitido sacar adelante un proyecto del que sentirnos orgullosos, así como aprender sobre nuevas tecnologías y su aplicación a problemas sociales reales.

Por otro lado, agradecer a todas aquellas personas que de forma voluntaria participaron en la evaluación del proyecto, lo que nos permitió entender mejor la calidad de los resultados obtenidos.

Finalmente, a nivel personal, agradecer a los compañeros de carrera que nos han acompañado durante estos años y con los que hemos compartido vivencias académicas. Así como a los familiares y amigos, que nos han apoyado y animado durante las épocas de exámenes y las largas horas de estudio.

Índice general

Capítulo 1

Introducción

Motivación

Siempre se han dado situaciones de **injusticia** en el **entorno laboral** de origen muy diverso, e independientemente del ámbito profesional. Algunas de ellas están relacionadas con las condiciones salariales y horarias en las que se trabaja, otras son de carácter físico o mental. Por ejemplo, observamos algunos datos obtenidos por el *Instituto Nacional de Estadística (INE)* [38] en donde se aprecian dichas prácticas. Ver las gráficas 1.1 y 1.2.

Sin embargo, hoy en día, con el auge de las tecnologías y las **redes sociales**, muchas de estas situaciones están saliendo a la luz y dándose a conocer. Dichas plataformas están siendo utilizadas como un medio de denuncia, comunicación y apoyo. Especialmente, por aquellos trabajadores que no tienen un claro método de denuncia y buscan una comunidad que pueda ayudarles.

Destacamos algunas **iniciativas** como el movimiento *#FightFor15* [109] en Twitter. Fue llevado a cabo por millones de trabajadores estadounidenses que buscaban, y consiguieron en muchos estados, que se aumentase el salario mínimo a 15$ la hora. Otro ejemplo, que tuvo lugar en España, fue el caso *#MalpasoPagaYa* [110], contra una editorial que no pagó a los autores de los libros que publicaba.

Todo esto nos lleva a preguntarnos por la **cantidad** y el **tipo** de denuncias que se llevaban a cabo, la posible **relación** que puede haber entre ellas o con el contexto en el que se producen, si algunas pueden estar pasando **desapercibidas** frente a otras con mayor atractivo mediático, si se están publicando casos **falsos** (*fake news*) o si las denuncias realizadas pueden estar siendo **silenciadas**.

Proponemos, por lo tanto, crear una aplicación que permita dar **visibilidad** y **unificar** en un único lugar todas las denuncias. De forma que esté diseñada para que todo el que quiera pueda acudir a **consultar**, **compartir** o visibilizar dichas situaciones.

Frente a la gran cantidad de **datos** que se encuentran en las redes sociales, planteamos

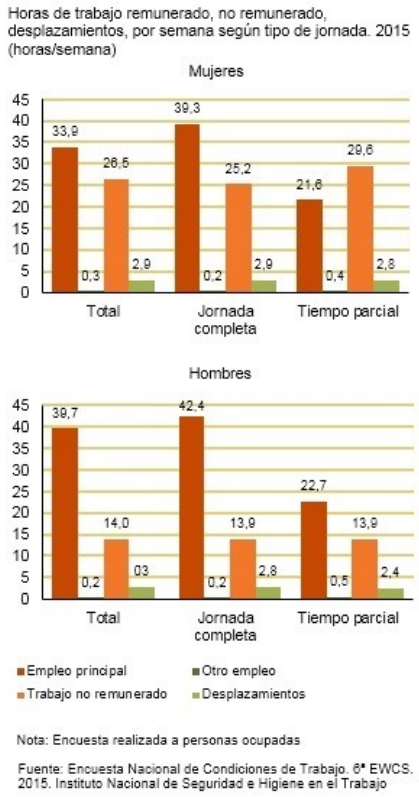


Figura 1.1: Horas de trabajo no remunerado y comparación por sexos.

Figura 1.2: Varias profesiones superan la cantidad legal de horas laborables en España

hacer uso de técnicas de **análisis de lenguaje natural** para obtener aquellos casos que se correspondan a denuncias laborales. Además, con el fin evitar la **censura** por medio de la eliminación o manipulación del contenido de la denuncia, proponemos gestionar los datos utilizando **tecnologías distribuidas**.

Objetivos

En vista de la situación expuesta, nuestros objetivos en el desarrollo de dicha aplicación son los siguientes:

- Dar visibilidad a situaciones de injusticia laboral denunciadas en redes sociales, así como al contexto en el que tienen lugar. En particular, de personas hispanohablantes.
- Proporcionar un lugar de apoyo y denuncia para ese tipo de situaciones.
- Desarrollar una interfaz que pueda ser utilizada y entendida por un usuario general.
- Aprender y utilizar técnicas de análisis de lenguaje natural para obtener las denuncias.
- Aprender e implementar tecnologías distribuidas que permitan evitar la censura del contenido que se muestra.

- Realizar el proyecto de forma que se incentive el desarrollo y uso de software libre.

Estructura del documento

En esta memoria reunimos todo el contenido necesario para poder entender el proyecto. A continuación, describimos brevemente la información recogida en cada capítulo.

- **Capítulo 1:** Presenta el contexto que motivó la creación de este proyecto, qué objetivos se perseguían y cómo se organizó el proyecto.
- **Capítulo 2:** Recoge la información anterior en inglés.
- **Capítulo 3:** Proporciona una visión teórica, y un contexto, sobre las tecnologías y líneas de trabajo que vamos a utilizar.
- **Capítulo 4:** Expone las metodologías de trabajo usadas, cuál ha sido la planificación y contribuciones realizadas, y qué tecnologías se han utilizado para el desarrollo del proyecto.
- **Capítulo 5:** Muestra el trabajo que se llevó a cabo, antes de que se iniciase el proyecto, con el fin de aprender y entender las tecnologías que iban a utilizarse.
- **Capítulo 6:** Muestra la fase de investigación que se llevó a cabo para conocer y entender mejor, tanto al usuario de la aplicación, como las iniciativas puestas en marcha que pudiesen estar relacionadas. Además, recoge los requisitos que determinamos debía cumplir nuestra aplicación.
- **Capítulo 7:** Desarrolla el proceso seguido durante el diseño e implementación del proyecto.
- **Capítulo 8:** Muestra la arquitectura y resultados finales de nuestra aplicación.
- **Capítulo 9:** Muestra los resultados obtenidos tras evaluar la aplicación desde dos puntos de vista; interfaz de usuario y calidad de los datos mostrados.
- **Capítulo 10:** Plantea el posible trabajo a futuro que podría realizarse y la perspectiva que presenta la aplicación.
- **Capítulo 11:** Recoge la información anterior en inglés.

Recursos

En el siguiente enlace a GitHub, se pueden encontrar los repositorios de este proyecto:

https://github.com/injustweet-tfg

El acceso a la web de Injustweet se puede hacer desde el siguiente enlace:

https://dashboard-twitter.herokuapp.com/dashboard

Capítulo 2

Introduction

Motivation

It is known that there has always been different kinds of **injustice** in the **workplace**. Some have to do with the salary conditions or working hours, others involve physical and psychological violence. But at the end of the day, all of them have one thing in common, they affect the health and well-being of workers. The *Spanish National Institute of Statistics* obtained some data related to this topic that can be seen in graphs 1.1 and 1.2.

In our day to day, due to the recent advances in technology, many of this situations are been known and listened to. **Social media** has become a platform where those in need can report injustices, communicate with others in the same situation as them or get help. It has been specially useful for workers that do not have the means to report and are looking for a community who can give them support.

As an example, we would like to highlight some **initiatives** that have taken place during the past years. First of all, the American political movement "The Fight for 15". They created a hashtag in Twitter called *#FightFor15* which helped them achieve a raise of the minimum wage in many states. On the other hand, in Spain took place a movement against an editorial who did not pay their workers. They made use of the hashtag *#ElpasoPagaYa*.

All of this raises some questions. For instance, **how many** and **what kind** of workplace related denunciations are being reported. It would also be interesting to know if they are **related** to one another and in which context are they being made. Another concern may be if the most popular ones are **eclipsing** the rest, if there are **fake** ones or if they are being **silenced**.

Therefore, we propose a web application which can give **visibility** and **gather** all of this reports in one place. This would mean that everyone who is interested could **consult**, **share** and give visibility to those situations.

Due to the gigantic amount of **data** that can be found in social media, we propose to use **natural language processing** techniques (NLP). This would allow as to find and obtain data

related to workplace denunciations. Moreover, with the final purpose of avoiding **censorship**, we bring forward the idea of using the **blockchain**.

Goals

In view of the situation exposed, we develop our application with the following goals in mind:

- To give visibility to workplace related denunciations in places where Spanish is their main language. We would like to show, not only the reports, but also the context in which they take place.
- To provide a place where people can share, see and report situations of injustice, as well as support one another.
- To develop an interface that can be used and understood by everyone.
- To learn and use natural language processing techniques in order to acquire data.
- To learn and use the blockchain as a way of avoiding censorship.
- To develop a project using and encouraging the use of free software.

Structure of this document

This paper contains the information required to properly understand the project. We now show the contents of each chapter.

- **Chapter 1:** It presents the reasons why we started this project and the motivations behind it.
- **Chapter 2:** Chapter 1 in English.
- **Chapter 3:** It gives a theoretical point of view of the technologies used and the lines of work that have inspired us.
- **Chapter 4:** It explains the work methodologies and technologies used, as well as the planning made and how each person contributed to the project.
- **Chapter 5:** Before starting the project, we took some time to learn and develop smaller projects which are shown in this section.
- **Chapter 6:** It explains the investigation phase that took place, as well as all the requirements we had to meet are shown in this section.
- **Chapter 7:** It presents the process followed during the design and implementation phase. It made us better understand the market we were entering to and the users we had to target.

- **Chapter 8:** The full software architecture of our project can be found here.
- **Chapter 9:** After finishing the project, a phase of evaluation took place. It had two points of view; user experience and NLP.
- **Chapter 10:** It presents the results and conclusions of our project. It also proposes the future lines of work that could be followed.
- **Chapter 11:** Chapter 10 in English.

Resources

The project can be found in GitHub using the following link:

https://github.com/injustweet-tfg

Access *Injustweet* using the fillowing link:

https://dashboard-twitter.herokuapp.com/dashboard

Capítulo 3

Marco teórico y estado del arte

3.1. Marco teórico

Dedicamos esta sección a presentar los principales temas que vamos a tratar, introduciendo el contexto tecnológico y social en el que se desarrolla nuestro proyecto. Vamos a contarlo en tres partes.

En primer lugar, nos centraremos en ver cómo las **redes sociales** se han utilizado como forma de **organización social** para denunciar injusticias laborales y sociales, así como a entender cómo de importante es la **colectivización** de personas para obtener resultados.

A continuación, dedicamos una sección a ver cómo se pueden aprovechar técnicas de **análisis de lenguaje natural** para detectar y recoger las situaciones que sean de interés.

Finalmente, nos acercaremos a conocer qué es la tecnología **blockchain**, un término muy utilizado durante los últimos años y que se está introduciendo cada vez más en nuestra sociedad. Entenderemos los conceptos básicos, un poco de su historia, y en qué momento nos encontramos actualmente.

3.1.1. Redes sociales y organización social

Hoy en día, gran parte de la comunicación que realizamos a través de internet es gracias a las redes sociales. Entendemos por redes sociales aquellos servicios web o aplicaciones que nos permiten conectarnos con otras personas, para compartir e intercambiar información, creando así redes de contactos y comunidades. Las redes que se forman pueden ser privadas o públicas. Pero son éstas últimas las que han promovido, durante los últimos años, que las redes sociales no sólo sean un lugar donde compartir experiencias con tus contactos, sino también un medio de difusión masiva de información.

Todo esto ha conducido a que las redes sociales se utilicen, en particular, como **medio de organización y movilización ciudadana**. Esto se debe a que ayudan a aumentar la parti-

cipación y el sentimiento de pertenencia a colectivos. Además, dan voz a iniciativas que, por los medios tradicionales, podrían no ser escuchadas.

Un ejemplo de uso de redes sociales con este fin, es la **movilización de los trabajadores** de Walmart de 2010. Walmart es una cadena multinacional de supermercados estadounidenses, con una plantilla global de 2.2 millones de trabajadores, de los cuales 1.4 eran trabajadores por horas[121]. La organización "OUR Walmart" (*Organisation United for Respect at Walmart*), fundada por la UFCW (*United Food and Commercial Workers*), fue creada para combatir la amenaza que suponía la compañía para la organización de los trabajadores [115]. A los trabajadores se les impedía organizarse, incluso eran sancionados si se les veía hablar de esos temas durante las jornadas laborales, por lo que las redes sociales supusieron un canal de comunicación seguro y directo, donde exponían sus situaciones y se apoyaban mutuamente. OUR Walmart se convirtió, de esta manera, en un movimiento con bastante repercusión en los medios. Gracias a él, consiguieron aumentar el salario inicial 10 dólares la hora, cambio que afectó a 500,000 trabajadores [115].

Otro ejemplo, fue el **movimiento #FightFor15** surgido en Twitter en 2012, con el fin de aumentar el salario mínimo a 15 dólares la hora a los trabajadores de un restaurante de comida rápida. Gracias a las redes sociales, obtuvieron un nivel de exposición mucho mayor al esperado, debido a la gran cantidad de usuarios que compartían el contenido. Esto evitó que tuviesen que depender de los medios tradicionales para hacerse oír. La iniciativa tuvo tanta repercusión que, gracias a ella, se consiguió la aprobación de leyes, en numerosos estados estadounidenses, que aumentaban el salario mínimo, acercándose o superando los 15 dólares la hora.

Por lo tanto, se ha visto que las redes sociales son muy útiles a la hora de unir a personas en distintas situaciones de injusticia y visibilizar, a gran escala, problemas que son difíciles de difundir haciendo uso de medios tradicionales. En particular, una de las redes más utilizadas para denunciar este tipo de situaciones es *Twitter*, gracias a la gran actividad que llevan a cabo sus usuarios, y al uso de *hashtags*.

Twitter nace en 2006, fundada por Jack Dorsey y sus socios en San Francisco [85]. Propone un servicio en el cual sus usuarios pueden compartir mensajes de texto, llamados *tweets*, de hasta 280 caracteres (previamente 140). Los usuarios también pueden reaccionar a otros tweets, compartiéndolos mediante *retweets*, o marcándolos como favoritos con *likes*. Además, el uso de *hashtags*[1], permite agrupar tweets relacionados entre sí.

Además de los movimientos mencionados anteriormente, hay muchas otras iniciativas que han utilizado Twitter como medio de difusión. Por ejemplo, *#MeToo* [54], nacido en 2017, para denunciar los abusos y agresiones sexuales por parte de productores de Hollywood, o *#BlackLivesMatter* [9], que denuncia la opresión de la comunidad negra ejercida por el estado y las autoridades. Twitter ha permitido y contribuido a la creación de espacios seguros donde

[1]Palabras sin espacios precedidas del símbolo #

las personas se unen para denuncias, crear colectivo y apoyarse mutuamente.

3.1.2. Análisis de lenguaje natural y web scraping

Hemos hablado de la gran cantidad de datos e información útil que pueden encontrarse en redes sociales, debido a la participación activa de una gran cantidad de gente. Nos planteamos cómo extraer y analizar esos datos, la mayor parte de los cuáles se encuentran en formato texto.

3.1.2.1. Análisis de lenguaje natural

El **procesamiento del lenguaje natural**, o NLP por sus siglas en inglés, es un campo en la intersección entre la inteligencia artificial y las ciencias del lenguaje que permite a la informática analizar, interpretar y comprender el lenguaje humano. Entre las distintas aplicaciones de esta tecnología podemos encontrar las siguientes.

Clasificación de textos

La clasificación de textos [59, 49, 21] consiste en, a partir de una muestra, extraer información relevante que permita clasificarla en una clase de entre un conjunto de ellas. Dependiendo de cómo se lleve a cabo esta clasificación, existen 3 principales sistemas.

- **Rule-based systems**

  La clasificación de textos se realiza utilizando un conjunto de reglas lingüísticas hechas a mano. Estas reglas tienen en cuenta una serie de elementos del texto que les permiten identificar a qué clase pertenecen. Sin embargo presentan grandes desventajas:

  - Por una parte requieren un amplio conocimiento sobre los temas que se están clasificando, para saber qué palabras se usan en cada contexto.
  - Requieren de muchas pruebas y cambios para que funcionen correctamente, lo que se traduce en mucho tiempo de desarrollo.
  - Son difícilmente escalables, puesto que las nuevas reglas pueden llegar a afectar a las ya existentes.

- **Machine learning-based systems**

  A partir de textos de ejemplo preseleccionados, también conocidos como corpus, los algoritmos de *machine learning* aprenden a clasificar textos gracias a métodos estadísticos que les permiten crear su propias reglas. La principal ventaja de este tipo de sistemas es precisamente su escalabilidad, no requieren de añadir reglas manualmente porque basta con ampliar el corpus. Además, son generalmente más precisos que los sistemas basados en reglas, ya que no requieren un profundo conocimiento del tema. Lamentablemente, son menos explicables que estos últimos, un pequeño cambio en el corpus de entrenamiento puede alterar completamente el funcionamiento del sistema sin conocer

su causa. Finalmente, observamos que se debe tener en cuenta que requieren de corpus preseleccionados que cumplan con las características necesarias para entrenar al modelo, dependiendo de los textos necesarios puede hacerse difícil encontrar uno adecuado y crear uno a mano requiere de mucho tiempo y esfuerzo.

- **Hybrid systems**

  Los sistemas híbridos son una complementación de los anteriores sistemas.

Machine Translation

La Machine Translation (MT) [87, 31, 47] es la traducción automática entre textos de habla humana. Destacamos las siguientes variantes.

- **Rule-based MT**

  Se basa en el empleo de un conjunto de reglas diseñadas por expertos lingüistas. El proceso de análisis parte de extraer información de morfemas, categoría gramatical de las palabras que forman el texto (*part of speech*), *named entity* (es decir reconocer y clasificar nombres propios de personas, organizaciones, fechas) y *word sense disambiguation* (reconocer cuál de las posibles acepciones que una palabra tiene es la que se está usando). Tras este proceso, se lleva a cabo una traducción de las palabras raíz del idioma original a las palabras raíz del idioma a traducir, y luego se traducen los sufijos. Finalmente, se lleva a cabo una corrección del género y número de las palabras.

- **Statistical MT**

  Emplea modelos estadísticos que asumen que la traducción de todas las palabras del lenguaje al que se traduce están correctamente traducidas con una cierta probabilidad. Cuanta mayor sea la probabilidad mejor es la traducción de esa palabra. Para ello se usan textos anteriormente traducidos y combinan las palabras en todas las posibles posiciones, lo que resulta en una base de datos de traducciones.

- **Neural MT**

  Utiliza redes neuronales combinadas, normalmente, con Statistical MT para obtener mejores resultados. Son las más complejas pero también las más efectivas.

- **Syntax-Based MT**

  Son una subcategoría de las Statistical MT pero llevan a cabo traducciones de unidades sintácticas en lugar de palabras.

Natural Language Generation

Es un proceso que permite transformar datos en textos comprensibles por los seres humanos. Distinguimos las siguientes etapas.

- **Content Determination**: Se prioriza qué parte del contenido es el que se va utilizar.

- **Data interpretation**: Se lleva a cabo una interpretación, con el fin de reconocer los datos que se quieren mostrar.
- **Document planning**: Los datos se organizan para poder crear una estructura narrativa acorde al ámbito en el que se va a utilizar el algoritmo.
- **Sentence Aggregation**: Se agregan aquellas frases que tengan cierta cohesión contextual entre ellas.
- **Grammaticalization**: Se revisa la gramática, ortografía y los signos de puntuación.
- **Language Implementation**: Finalmente, se muestra el texto en los formatos adecuados dependiendo de los usuarios a los que va destinado.

Tokenization, Lematización y Stemming

Debido a la riqueza usual en los lenguajes como las distintas partes de una oración (verbos, sustantivos, adverbios, adjetivos), la existencia de géneros o incluso los distintos tiempo verbales, es necesario el procesamiento de las palabras para poder reducirlas a un formato común más sencillo [45, 88].

La **Tokenización** es un proceso mediante el cual las palabras de una sentencia se dividen en espacios individuales, de esta forma se pueden obtener las palabras que conforman una frase o porción de texto.

La **Lematización** reduce una palabra a su lema, que es la forma en que esa palabra está representada en el diccionario. Para poder llevar a cabo este proceso se emplean diccionarios en diferentes lenguas y análisis morfológicos sobre estas palabras, con el fin de encontrar su similitud con las palabras del diccionario.

El **Stemming** elimina los sufijos de las palabras, pero no se debe confundir con la Lematización, puesto que este proceso extrae el sufijo de las palabras atendiendo a una lista con sufijos y prefijos comúnmente usados y aplicando sobre las palabras análisis morfológico. Observamos que no siempre funciona correctamente en todas las ocasiones, ya que puede detectar como sufijo o prefijo un conjunto de letras que pueden formar parte de la raíz de la palabra.

Junto con estos procesos existen otros que permiten eliminar palabras de los textos que puedan entorpecer la clasificación de los mismos. Uno de los más famosos es el *Stopword removal*. El **Stopword Removal** [118] es un proceso en el que se eliminan un serie de palabras sin significado alguno dentro de una oración, las palabras vacías o *stopwords*. Son palabras ampliamente usadas que no contienen un significado mayor a darle un sentido de construcción de las frases, son eliminadas a fin de reducir el tamaño del texto a analizar (lo que implica menor procesamiento). Una gran cantidad de bibliotecas NLP emplean estos procedimientos casi de manera invisible para el usuario.

En la figura 3.1 ilustramos un ejemplo de aplicación de algunos de estos procesos.

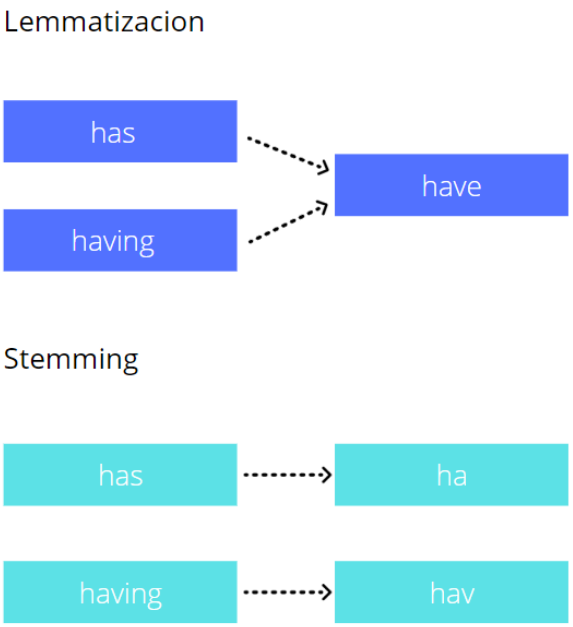


Figura 3.1: Comparación entre Lemmatization y Stemming

Evaluación de clasificación de textos

Para poder evaluar el desempeño de un sistema de clasificación de textos se emplean matrices de confusión [45]. La **matriz de confusión** es una tabla que representa los valores obtenidos frente a los valores reales. Enfrenta lo que el sistema ha clasificado frente a lo que realmente es.

Como se puede ver en la figura 3.2, es un matriz 2×2 en la que las filas representan los valores predichos por el algoritmo y las columnas los valores reales de la clasificación (es decir, los valores que debería predecir). Los valores numéricos que se situaran en la matriz representarían el número de muestras que han sido predichas de una manera frente a lo que realmente son dependiendo de la posición en la que se situase.

Poniendo como ejemplo de sistema de clasificación de textos el nuestro, en el que se busca reconocer si un texto es una denuncia o no, los valores representados serían los siguientes:

- **Esquina superior izquierda:**

  Número de verdaderos positivos (VP), es decir, número de muestras que han sido predichas como denuncias y realmente lo eran.

- **Esquina superior derecha:**

  Número de falsos positivos (FP), es decir, número de muestras que han sido predichas como denuncias erróneamente.

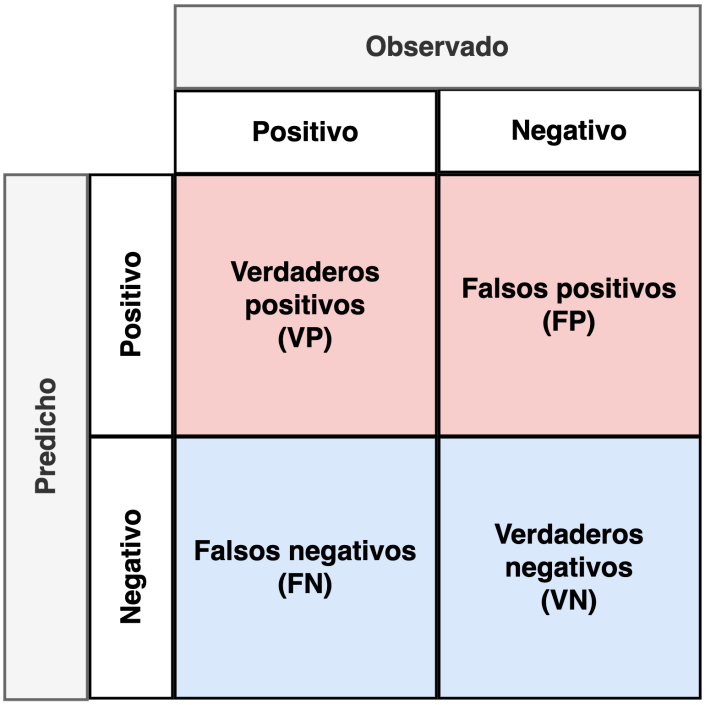


Figura 3.2: Matriz de Confusión

- **Esquina inferior izquierda:**

  Número de falsos negativos (FN), es decir, número de muestras que han sido predichas como no denuncias erróneamente.

- **Esquina inferior derecha:**

  Número de verdaderos negativos (VN), es decir, número de muestras que han sido predichas como no denuncias y han acertado.

Cuanto mayores sean los valores de los verdaderos positivos y negativos y menores los de los falsos positivos y negativos mejores estadísticas poseerá nuestro algoritmo. A continuación detallaremos cuáles son las principales métricas, que se obtienen a partir de la matriz de confusión, para determinar el desempeño de los sistemas de clasificación de texto.

- **Exactitud** (*accuracy)*

  Accuracy mide el porcentaje de muestras que fueron identificadas correctamente, es decir verdaderos positivos y negativos, entre todas las clasificaciones.

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN} \tag{3.1}$$

  Esta métrica puede parecer fundamental pero no se usa en clasificación de textos. La razón es que no es adecuada si el número de clases predichas es muy distinto entre si. Por ejemplo, si nuestro algoritmo predijera un número de denuncias mucho menor que

los que predice como no denuncias el *accuracy* sería muy alto, pero eso no significaría que nuestro algoritmo fuera capaz de reconocer cuáles son denuncias.

- **Precisión** (*precision*)

  Precision mide el porcentaje de muestras que el sistema predijo como positivo y fueron realmente positivo, lo que sería el porcentaje de verdaderos positivos entre todos los positivos.

$$Precision = \frac{VP}{VP + FP} \tag{3.2}$$

- **Exhaustividad** (*recall*)

  Recall mide el porcentaje de muestras que fueron correctamente identificadas por el sistema, es decir verdaderos positivos y negativos.

$$Recall = \frac{VP}{VP + FN} \tag{3.3}$$

  Se obtiene dividiendo los verdaderos positivos entre la suma de verdaderos positivos y falsos negativos. De esta manera cuantos más falsos negativos haya menor será la métrica, lo que indicará que no es capaz de reconocer qué no es una denuncia.

- **Medida F** (*F-measure*)

  Ambos *Precision* y *Recall* representan la capacidad que tiene el algoritmo para reconocer qué es una denuncia y qué no, al contrario que *Accuracy*. Sin embargo mantener ambos valores altos no es algo sencillo, puesto que cambiar el algoritmo para aumentar la *Precision* implica que el *Recall* se resienta.

  Su fórmula viene dada por la siguiente expresión:

$$F_\beta = \frac{(\beta^2 + 1)P \cdot R}{\beta^2 P + R} \tag{3.4}$$

  Donde el parámetro β representa cuál de ambas métricas pesa más en la valoración. Valores mayores a 1 favorecen al *Recall*, mientras que los menores a la *Precision*. Por ello F_1 es la más equilibrada y la más usada. Esta formula se obtiene a partir de una media armónica entre el *Recall* y la *Precision*. La razón de usar la armónica en vez de la aritmética se debe a que la media armónica halla valores más cercanos al mínimo de los valores sobre los que se halla que la aritmética.

3.1.2.2. Web scraping

Para utilizar las técnicas de procesamiento de lenguaje natural comentadas previamente, es necesario disponer de gran cantidad de datos. En muchas investigaciones, estos datos se obtienen de distintas redes sociales, ya que muchas de ellas proporcionan una API para facilitar la interacción con la información de estas redes. Sin embargo, cuando los datos no se

obtienen de redes sociales, sino de otras páginas webs, o cuando la API que proporcionan tiene restricciones, es necesario encontrar otros mecanismos para conseguir extraer esta información.

El **web scraping** [52, 122] es una herramienta que permite obtener un gran número de datos de sitios web, en un breve tiempo, sin necesidad de usar un API. Consiste en el uso de un script o un bot para simular a un usuario humano, descargar páginas web y buscar en ellas información específica.

En un inicio, se utilizaban para hacer una comparación de precios de un servicio que se ofertaba en distintas páginas web y también para recopilar direcciones de correo electrónico y crear listados a los que enviar publicidad [113]. A pesar de los posibles usos controvertidos que puede tener, es una herramienta muy potente para recoger datos y eso puede tener resultados muy positivos. Obsérvense a continuación algunos ejemplos.

- Permite a las empresas crear nuevos productos e innovar más rápido aún.
- En una época en la que la transparencia de las empresas ha cobrado tanta importancia, muchos datos han sido publicados en páginas web y navegar a través de ellos puede resultar muy complicado y abrumador. Utilizando el web scraping además se puede comparar con otras fuentes de información lo que aumenta la fiabilidad de los datos.
- Permitir a los negocios tener una clara visión de cuál es la imagen que hay suya en la red, ya que se puede extraer datos de reviews que han hecho los consumidores online.
- Facilita la automatización de procesos, la toma de decisiones de negocio y la creación de bases de datos con una gran cantidad de información útil

3.1.3. Blockchain como tecnología descentralizada

Finalmente, debido a la sensibilidad de los datos que nos resultan de interés, buscamos la descentralización en el almacenaje de los mismos, lo que nos lleva a introducir las tecnologías descentralizadas.

Una **cadena de bloques** o en inglés *blockchain* es una estructura de datos cuya información se agrupa en conjuntos a los que se les añade información relativa a otro bloque de la cadena anterior. Así, gracias a algoritmos criptográficos muy complejos la información almacenada en un bloque solo puede ser modificada editando todos los bloques anteriores.

La idea de blockchain surgió por primera vez en 2008 cuando el anónimo individuo Satoshi Nakamoto publicó un artículo titulado "Bitcoin: A Peer-to-Peer Electronic Cash System" [60]. En este artículo, define un nuevo sistema para realizar transacciones financieras sin requerir de una autoridad central. Bitcoin funciona bajo una **red P2P** (*Peer-to-peer*), es decir, una red donde un grupo de usuarios o máquinas participan de forma completamente **descentralizada**. De esta forma, no hay un punto central de conexión o control, y los usuarios actúan de forma

autónoma respondiendo bajo un consenso común.

A pesar de que la Bitcoin funciona bajo una tecnología muy moderna y novedosa, las aplicaciones descentralizadas no son algo reciente. Desde hace más de 20 años tenemos aplicaciones P2P muy conocidas como eMule o BitTorrent [78]. Estas aplicaciones mantenían la información guardada en una red de ordenadores que forman parte de su ecosistema. Al conectar un ordenador a la red de eMule o BitTorrent se podía acceder y descargar los contenidos incluidos en ellas.

Bitcoin prometía la posibilidad de realizar transacciones libres de impuestos, basándose en firmas digitales y criptomonedas en lugar de usar monedas emitidas por gobiernos. Todas las transacciones están registradas en un libro de registro, la **blockchain**. Ésta, como su nombre indica, es una implementación de un libro mayor, el cual es una simple base de datos de transacciones de monedas entre diferentes direcciones. Una transacción en un libro mayor es un intercambio o transferencia de dinero desde una dirección a otra. Los participantes son propietarios de monedas, y las monedas tienen asignada una dirección, un par de claves criptográficas y un valor monetario. En una transacción, estas monedas son las que se intercambian, habiendo algunas monedas de entrada y otras de salida.

Una parte de los usuarios aportaban a la red de Bitcoin su poder de computación personal para mantener la blockchain en funcionamiento, resolviendo problemas criptográficos que garantizan su seguridad. Como recompensa de esta aportación, recibían dinero en forma de Bitcoin. Estos individuos son los mineros (**nodos activos**), y existen en muchas más redes blockchain.

Para que la red funcione correctamente se requiere que todos los nodos activos **verifiquen la misma transacción** y compartan su libro de registro (blockchain) con todos los demás usuarios de la red (al resto de nodos activos, y también a los nodos pasivos, que son aquellos que no aportan poder computacional). Esto mantiene la **descentralización y transparencia de la red** y evita que pueda verse comprometida. Este tipo de sistemas descentralizados están basados en un consenso entre los nodos de la red. En este enfoque, los nodos de la red votan por lo que cada nodo considera ser la verdad, de manera que pueden cambiar su opinión si reciben información actualizada de otros nodos.

La cadena de bloques de Bitcoin necesita que todos los nodos verifiquen una transacción. Se considera muy complicado *hackear* o corromper la cadena de bloques ya que para que sucediera, la mayoría de los nodos tendrían que estar de acuerdo para ver una transacción maliciosa como correcta. Por esto se considera que la blockchain es una tecnología con mucho potencial y muy fiable para el futuro. Bitcoin funciona bajo un protocolo de **prueba de trabajo** (*proof of work*), que consiste en un protocolo de consenso en el que el nodo tiene que probar que ha ofrecido su potencia computacional para generar el bloque siguiente, resolviendo un problema criptográfico que sólo se puede solucionar a base de fuerza bruta.

Bitcoin tiene una cantidad máxima de 21 millones de monedas que puedan ser creadas o minadas. Una vez llegue a ese límite, no se podrán crear mas monedas de Bitcoin y los mineros empezaran a cobrar tarifas de transacción como recompensa de su trabajo. Es por esto que, a medida que se crean más Bitcoins, existen eventos como el *Halving* en la que la recompensa que reciben los mineros por proporcionar su potencia computacional se reduce a la mitad.

Las transacciones no se procesan como se reciben, sino que se procesan en lotes llamados **bloques**. Cada bloque de Bitcoin contiene miles de transacciones. Los bloques son generados cronológicamente y con marca de tiempo. Cada bloque está encadenado al anterior, es decir, incluye en él el *hash* del bloque anterior. Un *hash* es el resultado de una **función hash**, la cual es una operación criptográfica que genera identificadores únicos e irrepetibles a partir de una información dada. El propósito de esta cadena de bloques es realizar transacciones inmutables. Cualquier cambio en un bloque anterior hace que el *hash* del bloque siguiente quedé invalido y por lo tanto los nodos no lo aceptarán como válido.

En la Figura 3.3 podemos ver la estructura de los bloques en la blockchain. Vemos como en cada bloque aparte de las transacciones, también hay referencia al código *hash* asociado al bloque anterior. De esta forma, si se intenta modificar alguna información de un bloque previo, se deben calcular los *hashes* de todos los bloques posteriores, lo que conlleva un gran coste computacional, y garantiza la seguridad de la blockchain.

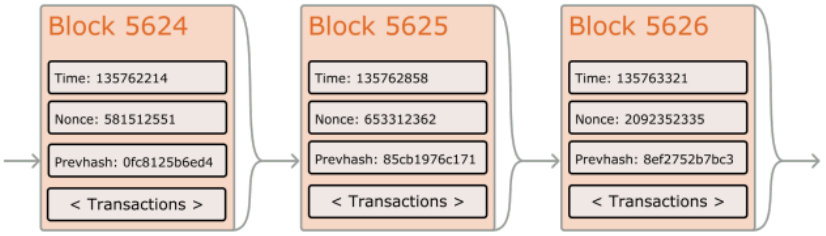


Figura 3.3: Esquema de bloques en la blockchain [26]

Los mineros trabajan de forma que toman un lote de transacciones e intentan calcular el siguiente bloque que se transmitirá a la red. Todos los mineros verifican que los bloques recibidos de otros nodos sean correctos y válidos. Esta manera de "minar" es la que tiene la red de Bitcoin, pero existen diferentes protocolos de prueba en otras blockchains que utilizan otras criptomonedas.

Después de todo el despliegue de Bitcoin surgieron nuevas monedas criptográficas con sus respectivas blockchains, con mejoras frente a la moneda que Satoshi Nakamoto creó. Surge así Ethereum [5], que además de tener el propósito de crear una nueva criptomoneda, es crear un protocolo para crear aplicaciones descentralizadas (dApps). Utiliza un token denominado ether (ETH). Se utiliza para pagar la ejecución de contratos y recompensar a los mineros para extraer nuevos bloques para la cadena de bloques.

Ethereum posee la infraestructura y herramientas para crear aplicaciones descentralizadas gracias a los contratos inteligentes. Un contrato inteligente, o **smart contract**, es un tipo especial de instrucciones y de código que es almacenado en la blockchain y que además tiene la capacidad de ejecutarse dentro de ella. Además, los smart contracts nos permiten eliminar intermediarios para simplificar procesos y ahorrar dinero para el consumidor y funcionan de forma inmutable, transparente y completamente segura. Por otro lado, son visibles por cualquier persona y se puede observar su código públicamente. Éstos viven en la blockchain en un formato específico de Ethereum llamado *bytecode* Ethereum Virtual Machine (EVM). Esta máquina vitual es capaz de ejecutar este tipo de código.

Ethereum esta formada por cuentas, en la que cada una tiene una dirección de 20 bytes. Existen dos tipos de cuentas, cuentas con dueño externo y cuentas de contrato. Además están programados mediante el lenguaje de programación *Solidity* con computación Turing. Que tenga computación Turing quiere decir que tiene un poder computacional equivalente a lo que se denomina maquina de Turing, es decir, un sistema que puede realizar cualquier tipo de cálculo y esto lo consigue gracias a la EVM.

Finalmente, las aplicaciones descentralizadas (dApps) mencionadas con anterioridad, son herramientas o sistemas que no están gestionados por una empresa y donde los usuarios se relacionan directamente sin intermediarios entre ellos a través de la blockchain. Estas aplicaciones son muy seguras debido a que la principal diferencia con una aplicación corriente, es que en vez de ejecutarse sobre un servidor central o varios, se ejecuta en una red formada por numerosos ordenadores o nodos. Además, la mayoría de aplicaciones y proyectos blockchain están basados en software libre y permite que exista una gran comunidad detrás de desarrolladores para dar servicio y soporte a la red.

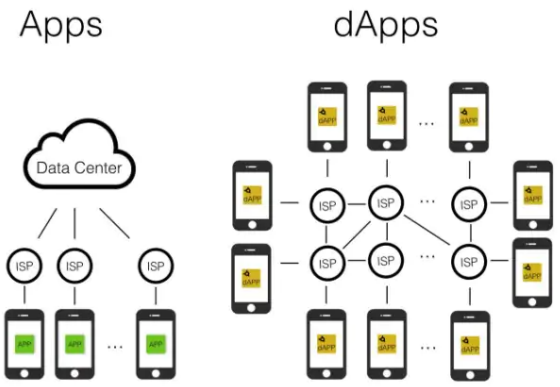


Figura 3.4: Comparación de plataformas de aplicaciones corrientes y dApps

En la Figura 3.4, se observa como las aplicaciones convencionales necesitan una figura central que se encarga de la gestión de los datos. Esto crea un factor de dependencia en las personas que han desarrollado la aplicación, que tienen todo el control. En cambio, las dApps

no tienen esa limitación, de forma que el control no recae sobre alguien específico, sino sobre toda la red que la gestiona.

3.2. Estado del arte

Existen otras iniciativas que buscan, al igual que nuestro proyecto, visibilizar o hacer frente a situaciones de injusticia laboral. Por ejemplo, la cooperativa *Smart* [90], que pretende ayudar a emprendedores a minimizar los riesgos y dificultades que enfrentan, o la aplicación *jobstice* [43], que proporciona una plataforma para hacer denuncias anónimas. Sin embargo, no hemos encontrado ninguna que haga uso de denuncias y plataformas ya existentes (redes sociales), o que aproveche las ventajas de la blockchain para evitar censura. Dedicamos, por lo tanto, esta sección a presentar distintos trabajos existentes donde se hace uso de los conceptos que se han presentado previamente.

3.2.1. Redes sociales como fuente de investigación

Como hemos visto previamente, en plataformas como las redes sociales se desarrollan distintos movimientos donde las personas se organizan y visibilizan situaciones de injusticia social. Sin embargo, a pesar de estas buenas acciones, las redes pueden acabar siendo un reflejo del mundo *offline*, por lo que también encontramos publicaciones promoviendo **discursos de odio** por motivos de raza, género u orientación sexual, además del acoso hacia otros a través del medio digital (*ciberbullying*). Que estas situaciones sean trasladadas a las redes sociales no sólo es fruto de reflejar este tipo de opresiones en otro medio, sino que el anonimato que proporcionan las redes ayuda a que las personas que difunden los mensajes muchas veces no tengan consecuencias.

Durante los últimos años, han aumentado los estudios sobre discursos de odio en redes sociales. El artículo de Ariadna Matamoros-Fernández y Johan Farkas [53] hace una revisión de 104 artículos que tratan el racismo y estos discursos en redes. Muestran que la mayoría de estos estudios utilizan Twitter como red social a estudiar, posiblemente debido a que su API proporciona un acceso a los datos de manera sencilla. También resaltan que Estados Unidos y Europa son las regiones más estudiadas, por lo que ponen de manifiesto la necesidad de investigaciones más inclusivas, para evitar reproducir los privilegios existentes dentro de este tipo de estudios.

Las redes sociales también se utilizan como medio de información, lo que ha conllevado a que cualquier usuario sea un potencial divulgador, y pueda publicar información sin ningún tipo de verificación o necesidad de referenciar las fuentes. Esto provoca que las personas que leen las publicaciones, sean susceptibles a creer dicha información, sea verdadera o falsa, por lo que las redes se convierten también en un **medio de desinformación**.

En el caso de la salud y la desinformación en redes sociales, se han realizado numerosos

estudios sobre el tema, y van a aumentando cada año, siendo las enfermedades infecciosas el tema más investigado [116]. Por ejemplo, en el transcurso de estos últimos años, se ha expandido mundialmente la enfermedad infecciosa COVID-19, y ha habido gran cantidad de información sobre este tema en redes sociales, sobre todo en Twitter, donde se comparten muchas publicaciones en situaciones de crisis [114]. En el trabajo de Mahsa Dalili Shoaei y Meisam Dastani [18] , podemos ver un estudio de varias investigaciones en relación al uso de Twitter durante la pandemia. En él se muestra cómo muchos usuarios publicaban sus experiencias con el COVID-19, pero también políticos y científicos publicaban información importante sobre la enfermedad, por lo que destacan la importancia de la presencia de organizaciones públicas para evitar la desinformación.

Las redes sociales saben de este problema que es la desinformación, y es por ello que durante el desarrollo de la pandemia, la mayoría implementaron mecanismos para avisar de la necesidad de verificar la información y combatir la desinformación [61]. Además, también se han realizado varios trabajos relacionados con este contexto, donde se visualiza la información publicada en redes sociales en forma de *dashboard*[2]:

- *USC Melady Lab* [89] presenta un dashboard donde hacen un seguimiento de las publicaciones relacionadas con el COVID-19 en Twitter. En él se analizan cuáles son los países que más publicaciones tienen, y dónde hay más desinformación, basándose en *fact-checkers* para verificar la información. Así como cuáles son los temas donde hay menos fiabilidad. También realizan un análisis del sentimiento de los tweets para mostrar la percepción de las personas en el transcurso de la pandemia, haciendo uso de técnicas de procesamiento de lenguaje natural.
- El *Portal COVID19MisInfo.org* [33] es una iniciativa cuyo objetivo es estudiar la desinformación relacionada con la pandemia, y proporcionan distintas herramientas para ello. Una de ellas es un dashboard de información obtenida de Twitter, que monitoriza tweets relacionados con la palabra "covid" o "coronavirus", para evitar la desinformación. Muestra información sobre los tweets o las cuentas que han sido eliminadas o suspendidas, así como las cuentas más contribuyentes o los links más compartidos.

3.2.2. Blockchain para el beneficio social

Ahora que se han introducido los conceptos básicos para entender qué es y cómo funciona la blockchain, vamos a comentar algunos aspectos en los que puede ser muy útil el uso de este tipo de tecnologías.

La tecnología blockchain es muy innovadora y tiene multitud de usos. A pesar de que la mayoría de estos son de tipo financiero, poco a poco se está extendiendo hacia otros marcos, como el de la discriminación en situaciones sociales [16], que a nosotros nos va a interesar especialmente.

[2]Un *dashboard* es una herramienta donde se visualizan y analizan distintas métricas y datos de manera visual.

El uso de esta técnica favorece que muchas denuncias puedan hacerse sin miedo a que puedan ser censuradas, eliminadas o reportadas posteriormente. Las aplicaciones y proyectos desarrollados para el bien social, buscan un cambio sistémico hacia hacia una distribución más equitativa de la toma de decisiones y beneficios fuera de la cadena de bloques. Además, la mayoría de las tecnologías blockchain son de código libre, de esta manera muchos más usuarios pueden comprobar, auditar y comprender todo el contenido del código y de los protocolos. Esto es crucial para explotar esta tecnología, ya que es fundamental para el beneficio social, comprobar la calidad de lo que está codificado, las arquitecturas que se adoptan y por último la seguridad que tienen. Otro de los beneficios que tiene por ser *open source* es que muchos usuarios pueden construir y desarrollar a partir de trabajos existentes, y así permitir avances en la democratización de proyectos blockchain.

Además, hay otro inconveniente muy común hoy en día, es el tema de la privacidad de los datos. Cada vez más, diferentes empresas centralizadas almacenan nuestros datos, para su uso o venta. Por lo tanto, otro de los objetivos de la blockchain, es recuperar los controles democráticos en la producción y uso de datos dentro de las redes sociales. Esto se consigue gracias a las técnicas de criptografía, que ofrecen únicamente la información necesaria para una determinada interacción digital. La finalidad de todo esto es democratizar cada vez más todo con el paso de los años aprovechando esta tecnología.

Capítulo 4

Metodología y tecnologías

4.1. Metodología

Antes de comenzar el proyecto, se creó el plan de trabajo que se iba a seguir durante su desarrollo. De esta forma, pudimos determinar cómo se iba a realizar la gestión de los recursos, a nivel de tiempo, humanos y tecnológicos. Para ello resultó esencial establecer la metodologías que se iban a utilizar.

Dedicamos esta sección, por lo tanto, a mostrar las metodologías de diseño y software empleadas, haciendo especial hincapié en el uso de software libre. Además, mostramos el plan de trabajo seguido y las contribuciones al proyecto de cada miembro del equipo.

4.1.1. Metodología de diseño y software

El objetivo de definir una metodología era el de poder establecer un marco de trabajo que nos permitiese estructurar, planificar y controlar todo el proceso de diseño y desarrollo de nuestro proyecto.

Elegimos seguir un **modelo de diseño** inspirado tanto en el diseño centrado en **actividad** (DCA) como en el diseño centrado en **usuarios** (DCU) [119]. Ambos forman parte del diseño centrado en humanos [17], que tiene como objetivo hacer que los sistemas interactivos sean usables y útiles para las personas. Su diferencia reside en que el primero se centra en la actividad que los usuarios podrán desarrollar con la tecnología, mientras que la segunda se centra en el usuario y sus necesidades. El diseño DCA basa sus fundamentos teóricos en la *Teoría de la Actividad* [8], cuya estructura consiste en detectar un motivo, realizar una investigación, ejecutar un plan de trabajo controlado y finalizar con una corrección del mismo. Actualmente, aunque es un campo muy estudiado a nivel teórico, no se ha desarrollado un proceso que permita su implementación, por lo que suele utilizarse a nivel de modelo y no tanto de desarrollo. A diferencia del anterior, DCU sí que está muy extendido en diseño de aplicaciones. Es por ello que existe un proceso claramente definido que permite su implementación; investigación, recogida de requisitos, diseño y evaluación.

Nuestro proyecto, partiendo de una motivación inicial (capítulo 1), buscaba implementar una serie de tareas llevadas a cabo por un usuario general. Decidimos seguir la estructura del diseño DCU, pero focalizando la participación del usuario al final del desarrollo. De esta manera, las fases iniciales pudieron centrarse en estudiar el contexto y las actividades que queríamos implementar para cumplir con nuestros objetivos. El proceso, por lo tanto, consiste en cuatro fases que se van iterando hasta obtener el resultado final. Observamos que no necesariamente tienen que hacerse todas las fases en cada iteración, dependerá del momento del proyecto en el que se esté y los resultados obtenidos hasta entonces.

- **Investigación**: Entender el contexto y las características del usuario.
- **Requisitos**: Establecer requisitos que debe cumplir la aplicación y modelizarla.
- **Diseño** e **implementación**: Diseñar e implementar las soluciones.
- **Evaluación**: Evaluar las soluciones desarrolladas.

Por otro lado, la metodología software utilizada ha sido **iterativa** e **incremental** de forma que, partiendo de un mínimo producto viable, pudiésemos ir añadiendo funcionalidades. De entre todas las metodologías ágiles, nosotros optamos por seguir un modelo simplificado de **scrum** [104]. Éste se caracteriza por tener iteraciones que duran 2-4 semanas (*sprints*) y reuniones tanto al principio como al final de la misma. Cada iteración, comienza con una definición de requisitos y objetivos, y termina con una evaluación de resultados. De esta forma, se controla, de forma colaborativa, la cantidad y calidad de trabajo que se va realizando. Como se verá más adelante en el plan de trabajo 4.1.3, comenzamos realizando un sprint inicial que permitió preparar el proyecto. El resto de sprints contaron con reuniones de seguimiento en las que se comenzaba repasando y evaluando los objetivos del sprint anterior, y posteriormente se establecían los del siguiente.

4.1.2. Metodología de software libre

El proyecto se ha desarrollado intentando fomentar el uso y creación de software libre. Es por ello que no sólo hemos intentado utilizar tecnologías libres, sino que también hemos escogido licencias para el proyecto que lo fuesen. De esta manera, nuestro código está disponible para toda la comunidad *open source*, permitiendo que tanto durante el progreso del proyecto como al final del mismo, cualquier persona sea libre de usar, modificar, distribuir y publicar versiones modificadas y mejoradas. El objetivo es conseguir un código más seguro y mantenible a lo largo del tiempo [69].

En nuestro caso, hemos utilizado *GitHub* para publicar el código fuente y para poder trabajar de forma colaborativa. El código está bajo la licencia **GNU General Public License v3.0** [51]. Tomamos esta decisión por el carácter vírico que posee, el cuál fuerza a que cualquier modificación del código tenga que ser distribuida con la misma licencia, fomentando y expandiendo el uso de software libre.

El código fuente del proyecto se puede encontrar en el siguiente repositorio de *Github*:

https://github.com/injustweet-tfg

De la misma manera, para la distribución de la memoria y del contenido creado en la aplicación web, hemos escogido una licencia **Creative Commons Attribution 4.0 International** [50], que permite copiar y distribuir la obra utilizando la misma licencia (incluido el uso comercial).

La aplicación web del proyecto se puede encontrar en el siguiente enlace:

https://dashboard-twitter.herokuapp.com/dashboard

4.1.3. Plan de trabajo

La planificación y organización del proyecto se llevó a cabo utilizando un *diagrama de Gantt*. Debido a la magnitud del proyecto, esto nos permitió llevar un seguimiento de las tareas que se iban realizando y las dependencias existentes entre ellas. Se puede observar en la Figura 4.1.

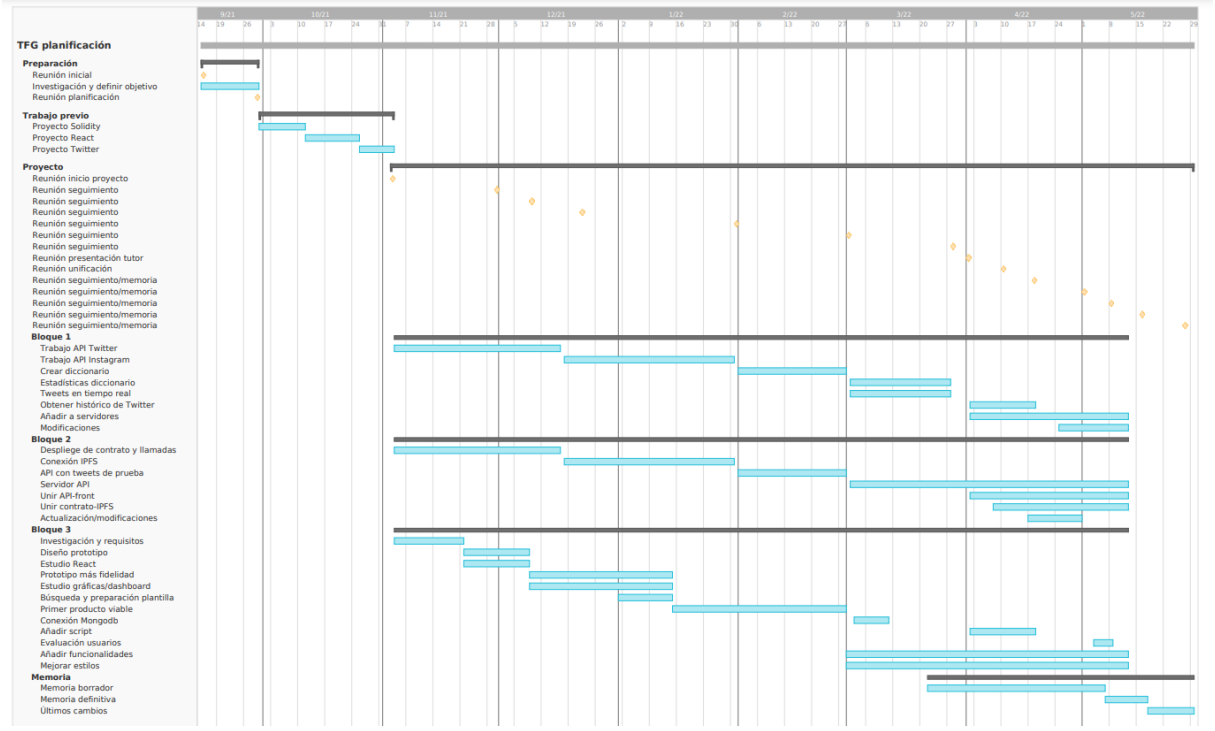


Figura 4.1: Diagrama de Gantt de la planificación

En primer lugar, se estableció un periodo de **trabajo conjunto** entre los 6 miembros del grupo. Este periodo sirvió para aprender algunas de las principales tecnologías que se iban a utilizar; React, Solidity y la API de Twitter. Realizamos pequeños proyectos que pusimos en común y sirvieron para afianzar conceptos. Esto garantizó que todo el equipo tuviese los conocimientos necesarios para entender partes del proyecto en las que no fuese a trabajar

directamente.

A continuación, se llevó a cabo una fase de **investigación** y recogida de **requisitos**, que permitió definir mejor el problema que queríamos abordar con nuestro TFG (visibilizar situaciones de precariedad laboral en lugares hispanohablantes) y cómo íbamos a llevarlo a cabo.

Posteriormente, dividimos el proyecto en tres bloques que trabajaron en paralelo en el **diseño** y la **implementación** de la aplicación. A partir de este momento, tuvimos reuniones periódicas en las cuáles mostrábamos nuestro progreso y garantizábamos que los tres bloques fuesen coherentes. Gracias a ellas pudimos realizar un seguimiento cercano y colaborativo de los objetivos que se iban cumpliendo. Inicialmente, tuvieron lugar cada 2-4 semanas laborables, y al final del proyecto, cada 1-2. Explicamos, a grandes rasgos, en qué consistió cada uno:

- Bloque 1 (**Recoger datos**): Pablo I.F y Raquel P.G
  Se encargó de la recogida de datos, obtenidos a través de la API de Twitter, y del categorización de los mismos, para garantizar que fuesen denuncias laborales.
- Bloque 2 (**Guardar datos**): Roberto A.L y Julián M.B
  Se encargó de recoger los datos proporcionados por el bloque anterior y almacenarlos en la blockchain. También se encargó de hacérselos llegar al siguiente bloque.
- Bloque 3 (**Mostrar datos**): Javier M.M y Ángela R.R
  Se encargó de mostrar los datos obtenidos en un dashboard. Además, gestionó los accesos a la blockchain y cómo alimentar a la página web.

Finalmente, se estableció un periodo que sirvió para unificar el proyecto y alcanzar un **mínimo producto viable**. A partir de este momento, se realizaron mejoras sobre el mismo. El proyecto terminó con una **evaluación** de resultados que sirvió para plantear los posibles **trabajos futuros**. Además, durante los últimos meses, se **documentó** el proyecto.

4.1.4. Contribuciones al proyecto

Finalmente, utilizamos esta sección para mostrar el trabajo realizado por cada miembro del grupo. Comenzamos viendo cómo se ha organizado y redactado la memoria.

Estructura de la memoria, descripciones de las secciones y revisión final: **Ángela y Javier**

0. Resumen y abstract: **Ángela y Roberto**
1. Introducción: **Ángela**
2. Introduction: **Ángela**
3. Marco teórico y estado del arte
   3.1 Marco teórico

- 3.1.1 Redes sociales y organización social: **Javier**
- 3.1.2 Análisis de lenguaje natural: **Pablo**
- 3.1.3 Web Scraping: **Raquel**
- 3.1.4 Blockchain como tecnología descentralizada: **Roberto**

3.2 Estado del arte: **Javier** y **Roberto**

4. Metodología y tecnologías

4.1 Metodología

- 4.1.1 Metodología de diseño y software: **Ángela**
- 4.1.2 Software libre: **Javier**
- 4.1.3 Plan trabajo: **Ángela**
- 4.1.4 Contribuciones: **Todos**

4.2 Tecnologías: **Todos**

5. Trabajo previo: **Ángela y Javier**
6. Investigación y recogida de requisitos: **Ángela y Javier**
7. Diseño e implementación

8.1 Bloque 1: **Pablo y Raquel**

8.2 Bloque 2: **Julián y Roberto**

8.3 Bloque 3: **Ángela y Javier**

8. Arquitectura

9.1 Bloque 1: **Pablo y Raquel**

9.2 Bloque 2: **Roberto y Julián**

9.3 Bloque 3: **Ángela y Javier**

9. Evaluación

10.1 Evaluación usuarios: **Ángela y Javier**

10.2 Evaluación análisis lenguaje natural: **Pablo y Raquel**

10. Conclusiones y trabajo a futuro:

11.1 Conclusiones: **Ángela y Roberto**

11.2 Trabajo a futuro: **Todos**

11. Conclusions and future work: **Ángela**

A continuación, cada miembro del equipo desarrolla en profundidad su aportación al proyecto y el trabajo que ha realizado.

Roberto Asunción López

Al inicio del proyecto, comencé a indagar sobre la tecnología blockchain, ya que desde el momento en que elegí realizar este trabajo me pareció un campo muy innovador y con mucha proyección de cara al futuro. Posteriormente, eché un vistazo a algunas redes sociales como Twitter e Instagram, para ver cómo estaba la situación sobre injusticias en el entorno laboral. De esta manera me hice una ligera idea de como podíamos darle visibilidad a este problema. Después de la primera reunión pactamos realizar tres pequeñas prácticas, una para aprender React, otra para Solidity y por último, una para la API de Twitter. Acordamos esto para que cada miembro pudiera ver que parte le gustaba más o con cuál tenía un mejor desempeño. La primera práctica que realicé fue la de Solidity, la cual consistía en hacer un pequeño sorteo de lotería. Además hice un pequeño curso en la página web "Udemy", para aprender este lenguaje. Posteriormente hice la parte de React, cuyo objetivo era hacer una lista de tareas. Ésta tarea me costó mucho ya que no me adaptaba a React y lo notaba muy confuso, además me llevó bastante tiempo y no pude realizar la última práctica. La parte que más me gustó y la que mejor se me daba fue sin duda la de Solidity.

A continuación de que todos terminásemos las prácticas nos dividimos el proyecto en tres grupos de dos personas cada uno. Los dos primeros grupos, se centraban en el back-end de la aplicación y el último, se ocupaba del front-end. El primer grupo integrado por mis compañeros Pablo y Raquel, se sometió a trabajar con la API de Twitter y así obtener denuncias. El segundo grupo, el cual participé junto con mi compañero Julián, se encargaba de recoger los datos que la parte de Twitter recogía, y los almacenaba en la blockchain interactuando mediante un contrato inteligente. Por último, la parte de React, integrada por Ángela y Javier, se ocupó de obtener las denuncias de la blockchain y mostrarlas a través de una página web mediante estadísticas, gráficas, wordclouds, etc.

Una vez tuvimos la reunión donde nos dividimos las partes. Julián y yo comenzamos a desarrollar un smart contract muy simple, que almacenaba (get) y devolvía una variable (set) y así poder probar las funciones del contrato desde fuera de este. Una vez teníamos el contrato nos surgió la duda de cómo podíamos desplegarlo en redes de prueba, ya que creíamos que desde Remix no se podía. Empezamos a usar la herramienta Truffle para desplegar el contrato e interactuar con él. Debido a la cantidad de problemas que nos daba utilizar esta herramienta, pensamos en otras alternativas. Finalmente opté por utilizar el mismo Remix, ya que descubrí que daba la posibilidad de desplegarlo en varias redes de prueba. Me encargué de crear una cuenta de Metamask asociada al proyecto y de introducir ethers de prueba para operar en la red de Ropsten. Además de esto utilicé Infura como infraestructura para las llamadas al contrato. Mediante Remix utilizaba el ABI generado al compilar el smart contract, para crear un archivo JSON y añadirlo al proyecto. A continuación de esto me ocupé de documentarme bien sobre la biblioteca "web3" para interactuar con el contrato e instanciarlo. Tuve bastantes problemas a la hora de llamar a las funciones del contrato ya que dependiendo de cómo estuviera programado, se llamaban de una manera o de otra. Para las funciones que fueran "view"tenía que añadir 'call' y para las que no lo fueran, tenía que añadir 'send' junto con un parámetro 'from' con la dirección de la billetera de metamask que utilizábamos para el

traspaso de ethers. Finalmente logré realizar la conexión con el contrato y llamar así a sus funciones de forma correcta.

A continuación después de otra reunión con nuestros tutores, nos dimos cuenta de que necesitábamos una plataforma para almacenar las denuncias. Guardar tantas denuncias en un smart contract era inviable. Después de investigar diferentes opciones optamos por utilizar IPFS para guardar la información de los tweets.

El siguiente paso a realizar fue el desarrollo de una API para que nuestros compañeros pudieran interactuar con nosotros más fácilmente. Optamos por usar "Express" y realicé dos pequeños métodos GET y POST. El método GET devolvía un array simple con tres tweets de prueba que cogí de ejemplo. El método POST simplemente añadía un 'hola mundo' a la API. Después enlacé el código de la API con la parte que interactuaba con el contrato ("Web3") y funcionaba de forma correcta sin problemas de dependencias. El problema surgió cuando enlacé la API con la parte de IPFS que mi compañero Julián realizó. Aparecieron numerosos errores muy confusos que nos impedían progresar en el proyecto. Me llevó mucho tiempo investigar de donde venían estos problemas y finalmente descubrí que eran por temas de dependencias y versiones de las librerías. Después de una reunión con uno de nuestros tutores, arreglé el problema actualizando todas las librerías y borrando la caché de npm (NodePackageManage) y del proyecto. Resolver este problema fue crucial ya que estuvimos muchos días intentando solucionarlo y nos ralentizó mucho las tareas.

Una vez teníamos la API enlazada con "Web3" y con IPFS empecé a mejorar el smart contract por mi cuenta para que tuviera las funciones que realmente necesitábamos. Añadí estructuras para almacenar los hashes devueltos por IPFS y para almacenar los identificadores de los tweets. Además agregué funciones adicionales para insertar hashes e identificadores, y otra para devolverlos. Aunque finalmente utilizamos el contrato que mi compañero Julián implementó. Probamos para comprobar su correcto funcionamiento.

Después de que tuviéramos el contrato modificamos los métodos POST y GET de la API para que tuvieran el comportamiento real. El método GET fue modificado para que devolviese los hashes de los ficheros almacenados en el contrato y con cada uno de ellos obtener la información de las denuncias de IPFS. Más tarde mi compañero se encargó de comprobar que estuvieran actualizados. Si no estaban actualizados se encargaba de hacerlo. Por otro lado el método POST lo modificamos para que subiera el contenido de las denuncias a la API y a IPFS, además almacenaba el hash devuelto por este último en el contrato.

Posteriormente nos surgió otro problema. La API que desarrollamos estaba gestionada de forma local en nuestro ordenador. Nuestros tutores nos aconsejaron que utilizásemos una plataforma para mantenerlo, ya que en local no era viable. Nos pusimos manos a la obra. Pensé en numerosas opciones tales como "Amazon Web Services (AWS)", "Google Cloud", "Heroku", etc. Finalmente, nos recomendaron "heroku" y me pusé a implementarlo con nuestro proyecto. Me creé una cuenta de heroku y después de varias guías de instalación logré enlazarlo con

nuestra aplicación. Asimismo surgieron errores a la hora de subir nuestro código a "Heroku", ya que no funciona de la misma manera que tener la aplicación en local. Instalé algunas dependencias más que la aplicación requería y pude solucionar estos errores. Una vez hecho esto, Julián y yo podíamos subir nuestra aplicación al servidor de "Heroku" e íbamos actualizándolo según corregíamos y añadíamos funcionalidades.

Los últimos días, considerando que ya teníamos el proyecto casi terminado nos preocupamos de observar y solucionar errores que surgían cuando enlazábamos todas las partes de la aplicación. Tuvimos algunos problemas debido a que algunos tweets no se subían a la API de la forma correcta. Pero, buscamos soluciones y Julian los solventó. También tuvimos el inconveniente de que nos estábamos quedando sin ethers de prueba, debido a que la red de Ropsten incrementó el coste en gas. Tuve que averiguar qué páginas suministraban estos tokens, ya que muchas de las páginas que usamos al principio ya no estaban operativas. Por último me centré bastante en redactar mis partes de la memoria, partes generales y la gran mayoría de las partes de blockchain, así como de mantener muchas reuniones con el resto de compañeros para asegurarnos de que todo iba por buen camino. Tuvimos un problema de última hora, ya que la red de Ropsten empezó a tener cantidades desorbitadas de gas para ejecutar las transacciones. Me ocupé de cambiar la red a Rinkeby y desplegar el contrato en ésta junto con mi compañero Julián.

Pablo Imbert Fernández

Una vez confirmamos nuestra participación en este proyecto decidí investigar de los lenguajes que íbamos a utilizar, ya que me interesaba programar en un lenguaje nuevo que fuera ampliamente utilizado. Habiéndonos asignado una serie de prácticas para poder tener una visión más global del proyecto, realicé la práctica de *Blockchain* y de *React*, la práctica de *Twitter* no pude realizarla por falta de tiempo. Inicialmente estaba más interesado en la parte de *React*, a pesar de haber encontrado dificultades sabía que era un framework bastante usado, o la parte de *Blockchain*, que fue la que primeramente me atrajo al proyecto. Sin embargo, al haber utilizado anteriormente *Python* conjuntamente con la *API* de *Twitter* en una asignatura del grado, Raquel y yo decidimos encargarnos de la parte de recolección de datos, ya que pensamos que así ahorraríamos tiempo y podríamos ayudar al resto de bloques. Tiempo después descubrimos que no podríamos ayudar en otras partes del proyecto, dada la longitud de nuestro bloque.

Inicialmente desarrollamos un pequeño prototipo con el que poder utilizar las funcionalidades de la *API* de *Twitter*, empleamos distintas tecnologías recurrentes en el campo de *NLP* (*tokenization, lemmatization, sentiment analysis*). La creación del diccionario a partir de publicaciones de *Instagram* usando la *API* de *Instagram*, así como la obtención de sus estadísticas fueron realizadas en su mayoría por mi compañera Raquel, debido a que por problemas de librerías no fui capaz de ejecutar el código, por lo que mi participación estuvo sujeta a que Raquel pudiera compartir pantalla y ayudarla en lo posible. Sin embargo las decisiones tomadas sobre las palabras que conformaron el diccionario final fueron realizadas conjuntamente.

Tras tener ciertos problemas con la revelación de tokens de *Twitter* y cuentas y contraseñas de *Instagram* en *Github* incorporé una serie de modificaciones al código, con el objetivo de mantenerlos a salvo. Para ello agregé un sistema de manejo de tokens basado en archivos .env, los cuales no son subidos a *Github* gracias a la creación de un archivo *.gitignore* (que permite ignorar ciertos archivos). En el caso de la cuenta de *Instagram* agregé un manejo de claves que empleaba el *keyring* del sistema *Linux* (llavero).

Una vez el diccionario fue terminado, retomamos el desarrollo de la interacción con el *API* de *Twitter*. Tras detectar ciertos problemas para recoger el histórico de datos, decidimos emplear la librería de *tweepy* para recoger los *tweets* a tiempo real. En mi caso dediqué una gran cantidad de tiempo a la puesta en marcha del *Listener*, así como conseguir que obtuviera los campos necesarios de cada *tweet*. Para poder hacer que funcionara correctamente toda la lógica del programa, fue necesaria la incorporación de una Base de datos de *Mongo*, así como su conexión desde el código. Esta parte fue desarrollada por mi, primeramente en local y más tarde con un servidor en la nube.

Tras descartar utilizar la *API* de *Twitter* para la recolección de datos con cierta antigüedad, nuestro codirector nos sugirió emplear un *Scraper Web*, *snscrape*. El código de esta nueva modalidad fue desarrollado a partir del existente por mi compañera.

Paralelamente traté de subir nuestros códigos a los servidores de *Heroku*. Sin embargo, y tras muchos intentos, no conseguimos incorporar el código al servidor. Se nos ocurrió la idea de utilizar los famosos servidores de *Amazon Web Services*, por lo que creé una cuenta *student* para poder utilizar las instancias *EC2*, desde la que poder ejecutar nuestro código. La puesta en funcionamiento supuso una gran cantidad de tiempo, debido a las dificultades que presentaban la instalación de la versión de *Python* con la que habíamos desarrollado todo el proyecto y la cantidad de errores de lo más variopintos por la instalación de librerías y sus dependencias. También fue necesario modificar las reglas de conexiones *HTTP, HTTPS* y *SSH* para obtener los datos del *API* de *Twitter* y , posteriormente, la *BDD*. Debido a los recursos limitados de las instancias que utilizamos, pensamos en dejar de usar una base de datos local, por lo que llevé a cabo la migración de *MongoDB* a *MongoDB Atlas*, cuyo servidor también es proporcionado por *AWS*. Todo esto supuso cambiar la dirección de la BDD a la que se conectaba el código, junto a la creación de usuarios, habilitación de IP´s y demás. Es necesario mencionar que conté con la ayuda de mi compañera durante todo este proceso.

El método de ejecución de la *API*, proporcionada por el equipo de *Blockchain*, desde *Python* fue desarrollada conjuntamente con mis compañeros Raquel y Julián. La puesta en funcionamiento en los servidores fue realizada principalmente entre Julián y yo.

También ha habido varias modificaciones y soluciones de errores en el código que han sido realizadas tanto por Raquel, como por mí, pero su poca importancia en el funcionamiento general del programa implica su no mención en esta sección de la memoria.

Conjuntamente con el código he redactado numerosas partes de la memoria, mayormente aquellas que tenían que ver con mi campo en el proyecto.

Julián Moreno Bellaneda

En un principio, al igual que el resto de mis compañeros, empecé experimentando con las nuevas tecnologías con las que ibamos a tener que trabajar a lo largo de estos meses. Para empezar, como no tenía mucho conocimiento sobre el mundo de las tecnologías descentralizadas, ni del blockchain me apunté a un curso online de FUNDAMENTOS DE LA TECNOLOGÍA BLOCKCHAIN impartido en EDx. Aprendí con el curso, junto con información proporcionada al principio, una práctica que nos recomendaron hacer y documentación de las páginas oficiales, como funcionaban las tecnologías descentralizadas, como programar en Solidity y a familiarizarme con el entorno de programación llamado *Remix*. Al mismo tiempo, centré en aprender React gracias a otra de las prácticas que nos sugirieron hacer. Nos contaron que React era un framework basado en Javascript, pero no tenía conocimientos previos de ninguno de ambos, así que estuve una semana de forma intensa dedicado a comprender el funcionamiento de esta tecnología y las que la rodean, como Node.js, NodePackageManager (npm) o Javascript. Algo que tenía muy en mente y se habló desde el principio con el resto de mis compañeros era que todos queríamos aprender de todas las tecnologías, por lo menos lo fundamental.

Tras una de las reuniones, se procedió al reparto de contenido y se me asignó la parte de *blockchain* junto con otro de mis compañeros, Roberto. Empezamos creando un contrato básico que almacenaba una variable para probar funciones y más o menos que estructura queríamos llevar. Tras investigar un poco sobre el desarrollo de aplicaciones descentralizadas y hablar con nuestro director del TFG, llegamos a la conclusión de que almacenar todo en la blockchain iba a ser inviable dada la cantidad de contenido que pretendíamos manejar y nos propuso utilizar otro tipo de tecnologías descentralizadas para el almacenamiento. Escogimos IPFS como zona de almacenamiento de contenido y comencé a explorar sobre el tema.

Continuando con la línea de investigación de las aplicaciones descentralizadas, tras haber practicado con React y Solidity, encotré un framework llamado Truffle que combinaba estas dos tecnologías y le propuse a mi compañero Roberto utilizarlo para desarrollar la aplicación. En un principio nos venía muy bien tener una interfaz donde probar las funciones que añadíamos, por lo que empecé creando una aplicación que se conectase a IPFS y fuera capaz de subir contenido. Con la subida de contenido, IPFS devuelve un hash para poder buscar el contenido que ha sido almacenado, este hash lo mostraba por pantalla para poder buscarlo en un buscador de IPFS y hacer pruebas sobre si se subía bien y que clase de contenido podía llegar a almacenar. Para conectarme a IPFS estuve investigando sobre diferentes formas y gracias a mi compañero Roberto descubrí Infura, él lo estaba usando para conectarse al contrato, pero esta infraestructura es muy amplia y tras investigar, resultó tener métodos para conectarse a IPFS. Más adelante, programé la capacidad de descargarse el contenido de IPFS, en un principio esto resultó más complejo de lo que pensaba, puesto que la información es devuelta en chunks

y tuve que averiguar exactamente como se almacenaba para poder pasarlo a texto legible. Al principio esto daba problemas con la letra "ñz con letras con acentos, pero más adelante lo solucioné usando métodos de String en vez de pasando la información de enteros a char.

Mientras realizaba la parte de IPFS, y tras la decisión de utilizar Truffle, junto con Roberto tratamos de avanzar en la parte de blockchain. Investigué sobre Web3, cuyo uso principal iba a ser conectarnos con Metamask, en esta parte tuvimos que arreglar un problema que venía porque el uso de gas no estaba bien limitado.Tras solucionar estos problemas, conectamos el código de IPFS con el de las llamadas a funciones del contrato desarrollado por Roberto. Una vez hecho esto, me di cuenta de que tener una interfaz no iba a ser lo más óptimo por la dirección que estaba tomando el trabajo. Tras una reunión con nuestro codirector del TFG decidimos que, en vez de una interfaz, debíamos desarrollar una API para automatizar mejor los procesos y no tener que hacerlo todo de forma manual.

Para crear una API nos recomendaron investigar sobre *Express.js*, un módulo de Node que se encarga de crear API Rest, por lo que me puse a investigar y junto a mi compañero desarrollamos un pequeño código que desplegaba el servidor con el puerto. Tras informarme del funcionamiento de este módulo, me di cuenta de que, por defecto, con un navegador las peticiones son HTTP GET, que era perfecto para quienes necesitasen obtener la información de la API, pero también necesitábamos un método POST para que nos diera la información a almacenar. Me puse a desarrollar dos códigos, uno en el que desplegaba un servidor con dos métodos, uno POST para subir contenido a la API y otro GET para comprobar que era correcto y otro para poder subir el contenido que deseara, puesto que no quería hacerlo con una web ya que en un futuro no tendríamos esa opción. Investigué sobre cómo realizar peticiones POST y encontré que con la función FETCH se podían modificar los parámetros de la petición, por lo que desarrollé el código con eso que se encargaba de enviar archivos JSON, pero también tuve que parsear las entradas a la API para asegurarme de que lo que le llegaba eran JSON.

Aunque no fue fácil, tras muchos errores, conseguí implementar el código que había en Truffle con las conexiones al contrato de Solidity y las conexiones a IPFS, para ello tuve que eliminar todo el código que quedaba en React para mostrar el contenido, puesto que la API funciona únicamente por consola y no tiene interfaz. Otro de los problemas que arreglé fue juntar dos tipos de módulos distintos que se importaban de formas diferentes, esto era culpa de los módulos ES6. Por último, para que esto funcionara, cambié el código que llamaba a la cuenta de Metamask, porque al no haber un navegador ya no podíamos usarlo y teníamos que pasarle directamente la cuenta con la que debía hacer llamadas a las funciones, para esta parte me ayudó Roberto, que tenía una cuenta con las ciptomonedas para la red de prueba que usamos. Este último código se modificó más adelante para tener las direcciones más protegidas usando otro tipo de cartera y almacenando las claves en un fichero más seguro de entorno.

Cuando junté todo el código, desarrollé el código en Solidity que hemos usado, tanto el inicial que usamos para guardar los hashes de los ficheros y acceder a ellos, como las versiones posteriores que implementaban otras estructuras y funciones pensadas para poder actualizar

los datos, como se cuenta más adelante. Tras varias iteraciones en las que lo he modificado para ajustarlo a las necesidades de nuestros compañeros, por ejemplo, cambiando los tipos de estructuras de memoria o la organización de las mismas. Para estas tareas, intenté usar el mayor número de funciones VIEW en dónde fuera posible, ya que no gastan ethers desde dónde las llamamos, también se utilicé estructuras como arrays y mapas, estos últimos muy optimizados en Solidity. En los casos en que este tipo de funciones no eran posibles porque modificaban el estado de la blockchain intenté desarrollarlo de la forma más óptima posible buscando un equilibrio entre el coste de llamar a las funciones, realizar la mayor parte de código fuera de este contrato para evitar un uso de gas innecesario y poder modificar la información subida sin eliminar contenidos almacenados para garantizar inmutabilidad de los datos obtenidos y garantizar que no se produciría censura. En estos casos me ayudé de un mapa para poder hacer más eficiente el acceso a contenidos del array principal en el que se almacenan los hashes de IPFS cada uno con un array que contiene los identificadores de los tweets, evitando bucles y búsquedas innecesarias.

Una vez hice el código del smart contract, realicé cambios al código de Javascript que desarrollé previamente para almacenar y descargar información en IPFS y recuperarla en base a las nuevas necesidades que habían surgido. Un ejemplo de esto es que, para descargar el contenido, primero se obtienen cada uno de los ficheros subidos y voy concatenando los tweets que contiene en función de si la información está actualizada para mostrarlos en la respuesta. También se realicé correcciones en cuanto a posibles objetos del array vacios, o caracteres como "," que pudiera haber en medio entre objetos. Otro de los cambios fue a la hora de subir contenido, puesto que tenía que guardar los identificadores de los tweets en el contrato de Solidity para ver que contenido de los ficheros subidos a IPFS iba a ser correcto más adelante en la descarga, puesto que se podría actualizar.

Tras todo esto, realicé el código que es encarga de actualizar el contenido de la blockchain. Implementé un método GET, porque no hay que pasarle información, que se descarga los tweets de IPFS y por cada uno se obtiene la información de la API de Twitter que puede haber cambiado para comprobar si está actualizado, en caso de no estarlo se modifica. Todos los que hayan sido cambiados se vuelven a subir en un archivo a IPFS y se pasa al smart contract junto con los identificadores modificados. Aquí hubo un problema en la subida de contenido porque los IDs eran muy grandes como enteros y lo cambié a string tanto en Javascript como en Solidity, para esta modificación me ayudaron mis compañeros Pablo y Raquel. Otro de los problemas con los que me encontré a la hora de actualizar el contenido fue que la cuenta de Twitter que se nos concedió, tiene limitado el número de peticiones que le podemos hacer con una restricción de 300 tweets cada 15 minutos. Tras adaptar el código para que se pueda llamar de forma periódica, desarrollé un cron en javascript, un código que se encarga de llamar a la función de actualizar cada un periodo de tiempo y de actualizar los tweets en tandas que la API de Twitter puediera manejar.

Una vez teníamos un código funcional, Roberto y yo desplegamos el código en Heroku tras aprender cómo funcionaba, de forma que la API estuviera en todo momento accesible para las

otras partes. Tras esto he ido actualizando tanto el código de la API como usando los logs del servidor para comprobar que todo funcionaba correctamente y configurándolo para almacenar las claves de las distintas API que usamos de forma segura.

También ayudé a mis compañeros encargados de la parte de recolección y procesado de información con la conexión de su parte del código con la nuestra. Junto con Pablo y Raquel, nos encargamos de buscar una forma de poder ejecutar el código que desarrollé en Javascript en su máquina integrándolo en el código en Python que tenían desarrollado. También les ayudé a conectar el servidor en el que tienen implementado su código con nuestra API para que pudieran subir el contenido que procesan. Junto con otros de mis compañeros he redactado partes de la memoria.

Javier Mulero Martín

Durante los primeros meses del proyecto, estuve investigando acerca de las tres principales tecnologías que íbamos a utilizar: React, Solidity y la API de Twitter. Como todos los miembros del grupo queríamos aprender acerca de todas las partes, y durante el desarrollo posterior del mismo nos íbamos a dividir en tres equipos, realizamos una práctica con cada tecnología. La primera que realicé fue la de Solidity, creando un sorteo de lotería con smart contracts. La siguiente fue la de React, donde me formé en React con HTML, CSS y javascript, y construí una aplicación de lista de tareas. Por último, ya que previamente al proyecto ya había trabajado con la API de Twitter, hice la práctica donde realicé consultas para obtener tweets con algunas palabras relacionadas con la precariedad laboral, y analizarlos para crear un wordcloud con las palabras más frecuentes.

Mientras estaba con la última práctica, también recopilé algunas aplicaciones web existentes que hacían uso de la API de Twitter y donde se visualizaba esa información recogida de Twitter en forma de diferentes gráficas o de diferentes maneras, para poder saber qué tipo de herramientas ya existían en la actualidad. Además, iba guardando cuentas de Twitter o hashtags donde se podían encontrar publicaciones relacionadas con denuncias de algún sector laboral, pues nuestro propósito del proyecto era visualizar este tipo de contenido.

Posteriormente a los meses que dedicamos a introducirnos y aprender las nuevas tecnologías, en el equipo nos dividimos en 3 grupos; el primero se encargaría de recopilar denuncias haciendo uso de la API de Twitter, el segundo se encargaría del back-end, utilizando la blockchain para almacenarlas, y el tercero realizaría la parte del front-end de la aplicación, para visualizar las denuncias. Es en este último en el que entré yo, junto con Ángela, y en el que trabajamos durante el resto del proyecto.

En esta primera fase, dediqué una parte del tiempo a realizar una investigación de aplicaciones que pudieran estar relacionadas con nuestro proyecto, ya fuera de forma parcial o total. También pensé en el tipo de usuario que podría usar nuestra aplicación y qué requisitos podía tener. A partir de ahí, y tras varias reuniones con el equipo, establecimos unas bases sobre las

que comenzar a montar la aplicación.

Antes de empezar con la web, lo primero que hicimos fue investigar un poco más a fondo React acerca de como estructurar un dashboard, y qué librerías existían para visualizar datos de distintas formas. Tanto Ángela como yo buscamos por separado algunas librerías y luego las pusimos en común, para decidir así cual se ajustaba más al proyecto. Entretanto, también nos dedicamos a hacer bocetos del dashboard con distintos niveles de fidelidad, aumentándolo después de las reuniones que teníamos con el grupo completo para actualizarnos acerca de cómo avanzaba el proyecto.

Cuando tuvimos un primer prototipo, pudimos comenzar a diseñar la web. Ya habíamos estado probando a utilizar gráficas de distintas librerías. En concreto, yo practiqué con *visx*, *react-chart-js* y *React-vis*, y utilizamos plantillas de dashboard con licencias libres para estructurar las distintas gráficas. En esto punto tuvimos que decidir si empezar el proyecto desde cero o basarnos en una plantilla. Probé a hacer alguna prueba con *bootstrap* y *react-bootstrap*, pero descubrimos *Material UI*, y como se ajustaba más a nuestra idea de diseño y existían plantillas con licencia MIT que nos iban a permitir avanzar con más rapidez, escogimos una de ellas sobre la que comenzar a trabajar.

Para empezar a trabajar en el desarrollo de la aplicación, ambos nos quedamos varios días en la facultad para trabajar juntos y estructurar el front-end. También trabajamos la mayor parte del tiempo con Live Share, la extensión de Visual Studio Code que nos permitía trabajar de manera online y colaborativa. Y al hacer uso de Git, nos permitió trabajar cada uno en el proyecto y mantenerlo actualizado y con control de versiones.

Nos dividimos de forma equitativa las distintas gráficas que íbamos a implementar (algunos datos globales, los top usuarios y hashtags, y algunas gráficas temporales). Yo me encargué principalmente de AppTopHashtags, AppTweets y AppHeatmap, aunque posteriormente tanto Ángela como yo hicimos cambios en todos los componentes.

Durante el proceso de la implementación del dashboard nos seguíamos reuniendo todo el equipo y poniendo nuevos objetivos. Uno de ellos fue crear una "memoria caché" para que el front-end de la aplicación no tuviera problemas de esperas de tiempo al pedir la información a IPFS, con lo que nos tuvimos que encargar de ello. Tuve que aprender acerca de cómo funcionaba una base de datos NoSQL, para lo que realicé un curso de Udacity sobre *MongoDB* e investigué la documentación de la web de *Mongodb*. Una vez supimos unas bases, comenzamos con el proyecto de cache-twitter, que íbamos a gestionar a parte del dashboard.

Lo primero que hice fue crear una cuenta y una base de datos en MongoDB Atlas, un servicio de base de datos en la nube, de modo que Ángela y yo podíamos trabajar sobre ella. Al principio solo teníamos una consulta que hacer: un método *get* para obtener todos los tweets entre dos fechas y utilizarlos en el dashboard. En esta parte me encargué más de preparar el componente que iba a implementar el filtro de tiempo, y estuve controlando las renderizaciones

que realizaban las componentes, mediante el uso de hooks.

A falta de terminar el dashboard, sólo nos quedaba crear un script (que sería el proyecto update-cache-twitter) que actualizara dos veces al día los datos de nuestra caché con los datos de IPFS. La primero que pensamos fue que, además de actualizar nuestros datos de la caché, también se deberían actualizar algunos metadatos sobre los tweets ya recogidos en IPFS (como la cantidad de retweets o likes). De modo que propusimos crear un script que se encargará de ambos trabajos, pero finalmente todo el grupo decidimos que era mejor que la parte de actualización la realizaran los de backend. Así, comenzamos a crear el script, donde cogemos los tweets de IPFS con el método *get* de su API y hacemos un *bulk save* en nuestra caché, es decir, vaciamos nuestra caché y guardamos los nuevos, ya que no modificamos ningún dato. En esta parte, implementamos juntos las funciones *add* y *delete*. También investigué cómo podíamos ejecutar el programa en un tiempo determinado del día, y encontré *node-schedule*, que nos permitía esa funcionalidad. Y a pesar de que la actualización de los metadatos comentada previamente se hiciera desde el back, implementé una función *update* que se encargara de actualizarlos, teniendo en cuenta la restricción que impone la API de Twitter de obtener 300 tweets cada 15 minutos. Así, en caso de algún problema o alguna decisión de cambio de diseño, estaría preparada y lista para su ejecución.

Las últimas semanas del proyecto fueron observar el funcionamiento tanto de la caché como del script que la actualiza mientras terminábamos el dashboard y redactábamos la memoria. Al dashboard le añadimos algunas funcionalidades más, como una búsqueda de palabras en los tweets y de usuarios que habían escrito tweets, además de una gráfica más donde se observa la correlación de las palabras más usadas en el tiempo. Yo me encargué del componente donde se realizaba la búsqueda e implementé el hook *useFetch* para gestionar mejor las consultas a *mongodb*. También agregué los *Skeleton* de *Material UI* para indicar cuándo está cargando un componente.

Estas últimas semanas fueron bastante intensas, pues aumentamos el número de reuniones para asegurarnos del correcto funcionamiento final de nuestra aplicación, y revisábamos lo que íbamos escribiendo en la memoria. También aprovechamos para hacer la página de información en nuestro dashboard mientras retocaba los últimos detalles de la interfaz. En estas semanas, también preparé y realicé la evaluación de 3 usuarios sobre nuestra aplicación, y revisé la memoria para corregir los últimos detalles.

Raquel Pérez González de Ossuna

Mi participación en este proyecto comenzó con una extensa fase de investigación sobre cada una de las tecnologías que se trataban en el mismo. Comencé investigando con el tema *Blockchain*. Para esta parte comencé leyendo algunos libros recomendados por los directores del TFG. Una vez consideré que tenía unos conocimientos suficientes sobre *Solidity*, realicé la práctica que se nos planteó como trabajo previo. Investigando sobre esta temática encontré que esta práctica se solía recomendar a los principiantes en este lenguaje, descubrí varias pro-

puestas similares con pequeñas modificaciones que decidí realizar para poner a prueba mis conocimientos.

La siguiente tecnología que abarqué fue la aplicación para crear y eliminar listas de *React*. Ya que esta tecnología está en auge, decidí dedicarle el mayor tiempo, investigando distintas formas de realizarla, leyendo una extensa bibliografía y visualizando varios tutoriales.

Finalmente, como preparación para la parte de la *API* de *Twitter*, consideré lo más importante entender qué tipo de denuncias se podrían encontrar en esta red social. Para ello, realicé una búsqueda exhaustiva sobre *hashtags* con denuncias, como por ejemplo sobre denuncias en el sector artístico dónde les habían robado sus diseños. De esta parte saqué en claro que tipo de vocabulario veríamos más adelante durante el desarrollo.

Una vez adquiridos unos conocimientos que consideramos base, realizamos la división del trabajo. Pablo y yo nos encargaríamos de la parte de recolección de datos, pues ya habíamos trabajado con anterioridad con la *API* de *Twitter* y creíamos que está familiaridad nos ayudaría a realizarlo de manera más efectiva y nos permitiría ayudar al resto de compañeros con otras partes del proyecto.

Más adelante nos dimos cuenta que al tratarse está parte del motor del trabajo, requeriría una cantidad de trabajo que no pudimos prever. Esto se debe a que, si el algoritmo para recoger denuncias no era eficiente o si tanto el contenido como la forma en la que lo que almacenábamos creaba cualquier discordancia, afectaría directamente a la viabilidad del producto final.

Por eso, lo primero que hice fue investigar sobre *dashboards* y *wordclouds*. Esto tiene importancia porque al ser una tecnología que sabía que mis compañeros del *front-end* iban a usar, quería conocer que tipo de información podrían necesitar y en que formato para que Pablo y yo pudiéramos programar de forma eficiente. De este modo, pretendíamos evitar cambios mayores una vez tuviéramos el algoritmo ya avanzado.

Tanto Pablo como yo hemos realizado el trabajo prácticamente de forma conjunta, pues siempre que era posible quedábamos para ayudarnos ya fuera con la resolución de *bugs* o para compartir nuestros avances. Sin embargo, sí hay algunas distinciones sobre qué hizo cada uno.

Nuestro primer trabajo fue la búsqueda de un corpus de denuncias en *Twitter*. Como habíamos aclarado con nuestro codirector íbamos a usar una cuenta de *Twitter* llamada *mierdaJobs* para extraer los *tweets*, realizar el análisis de sentimientos y crear el diccionario.

El primer problema con el que nos encontramos fue ver que la mayoría de usuarios utilizaban imágenes para compartir conversaciones u otro tipo de contenido multimedia que veían relevantes. Nuestra primera idea para solucionarlo fue, mediante el uso de librerías de reconocimiento de texto en imágenes, leer las conversaciones de las fotos para después realizar el *sentiment analysis* del texto entero. Esta parte la realicé principalmente yo con el apoyo

continuo de mi compañero, pues por problemas técnicos de dependencias entre librerías que eran necesarias, a él le resultaba imposible ejecutar el código en su portátil. Nuestra forma de trabajar consistía en compartir mi pantalla mientras él me iba comentando posibles mejoras o errores en el código.

Nos dimos cuenta pronto de la inviabilidad de hacer esto, pues el texto del *tweet* solía ser irónico y porque las conversaciones contenían caracteres que no terminaban de reconocer bien. Por esto, hablando con nuestro codirector, llegamos a la conclusión de pasar a la *API* de *Instagram* pues encontramos una cuenta con denuncias reales de múltiples usuarios sobre distintas empresas. De esta parte también me encargué yo principalmente por los motivos citados anteriormente. Aquí hubo una curva de aprendizaje para entender como funcionaba la *API* de *Instagram*, pero una vez hecho este trabajo, extrajimos un gran número de publicaciones y creamos el diccionario a partir de la frecuencia con la que se usaba cada palabra en cada texto.

Como queríamos que el diccionario fuera lo más fiel a la realidad, limpiamos los textos de caracteres especiales, hicimos *stemming* y *lemmatization* además de limpiar nombres propios de empresas. Después ambos limpiamos bien el diccionario, realizando de forma conjunta las tres iteraciones para ver cual sería nuestro diccionario final. Una vez nuestro codirector nos dio el visto bueno, comenzamos a calcular si éste era o no eficaz. Por eso, busqué varios corpus de tamaños similares a las publicaciones que no habíamos llegado a extraer y saqué la matriz de confusión además de varias métricas que nos hicieron ver que el algoritmo funcionaba.

Una vez realizado este trabajo, volvimos a *Twitter*. Lo primero que intenté llevar a cabo fueron consultas a la *API*, pero estas devolvían un pequeño número de denuncias y de una antigüedad máxima de una semana. Por eso, mi compañero Pablo decidió centrarse en investigar como funcionaban los *Listeners* para poder recoger *tweets* en *Streaming*.

En esta parte se centro principalmente él, pues yo estaba teniendo problemas con mi acceso a la *API* de *Twitter* tras realizar demasiadas peticiones con el primer método. También se encargó él de conectar el algoritmo a *MongoDB*, primero de forma local, que me enseño además a mí a utilizar, para que pudiéramos ejecutar los dos el código con distintas fechas y recoger así más denuncias. Esto mismo lo realizó luego en *MongoDB Atlas*, que gestionó él con sus claves.

Yo en está parte serví principalmente de apoyo para la resolución de errores que iban apareciendo a cada paso. Una vez montamos esto, vimos que recogíamos muy pocas denuncias, por lo que hablando con el codirector decidimos hacer uso de *scrapers* para recoger el histórico de *tweets*, ya que mediante la *API* de *Twitter* no teníamos acceso al mismo.

De esta parte me encargué principalmente yo, pues mi compañero estaba empezando a estudiar cómo podríamos subirlo a un servidor de forma gratuita. El mayor problema que se nos presentaba era que muchas de las versiones de prueba solo permiten el uso de un hilo y, por el diseño del *streaming*, ya se usaban dos. De todas formas, mi compañero estuvo ayudándome siempre que le necesité para resolver dudas, ya fueran de concepto o de código.

Finalmente utilizamos *AWS*. Para subirlo aquí, mi compañero dedicó una gran cantidad de tiempo, pues al utilizar una instancia con una versión antigua de *Debian* hubo muchos problemas de incompatibilidades ya fuera con *Python* o con distintas conexiones. Él tuvo que realizar un gran trabajo de investigación para que funcionará.

Ya que teníamos que utilizar una misma cuenta y para *AWS* se utilizan los datos bancarios, creímos que era mejor si solo uno usaba la cuenta. Por eso, yo en esta parte me dediqué a observar y ayudar a mi compañero cada vez que encontraba algún fallo que llevaba mucho tiempo. Además, hubo muchos problemas con el servidor para la habilitación de puertos y de *IPs*. En estos fallos pude ayudar más a mi compañero, pues por temas de trabajo estaba familiarizada con este tipo de errores.

Lo último que hemos tenido que realizar fue la *API* para conectarnos con el grupo de *Blockchain*. Julián nos paso el código a Pablo y a mí y hemos tenido que investigar como implementarlo. Hemos pasado mucho tiempo los tres intentando solucionar cada error que iba surgiendo y viendo como podíamos conectarla con *Python*. Además, ha llevado mucho tiempo entender porque funcionaba en local y no en el servidor, o porque funcionaba al realizar la llamada a la *API* desde la terminal pero no desde el código. Esta ultima parte nos ha llevado incontables horas a cada uno, yo intentado averiguar como hacer que funcionara en local, Pablo intentándolo en el servidor, y Julián estudiando los *logs* que devolvía para ver que podía estar fallando y *debuggear* así el código.

Cabe mencionar también el gran trabajo de investigación para cada una de las tecnologías que usamos además del tiempo dedicado a la resolución de errores o de incompatibilidades entre las distintas librerías y módulos. Esto fue realizado por los dos por igual.

Ángela Ruiz Ribera

Comencé definiendo, junto con el resto de mis compañeros, el objetivo que iba a perseguir nuestro TFG, y cómo íbamos a llevarlo a cabo. Para ello, busqué información sobre iniciativas que tratasen temas de injusticia laboral (*Smart*, *#FightFor15*...), y participé activamente en la planificación realizada. En particular, durante el desarrollo del proyecto, me encargué de llevar acta de las reuniones que tenían lugar y organizar las sesiones para que se tratasen todos los temas y se fijasen objetivos.

A continuación, dediqué un periodo de tiempo a aprender sobre los tres campos en los que íbamos a trabajar; desarrollo web, blockchain y análisis de lenguaje natural. Desarrollé una aplicación en *React* que permitía crear y eliminar listas de tareas. Esto me permitió familiarizarme con *javascript*, *CSS* y *HTML*. Además, aprendí los dos paradigmas de programación en *React*; los componentes clase usados antiguamente y los funcionales, introducidos en la versión 16.8. Por otro lado, implementé un *Smart Contract* usando *Solidity* y *Remix* que imitaba una lotería. Nunca antes había usado dicha tecnología, por lo que dediqué la mayor

parte del tiempo a documentarme. Finalmente, leí e hice un repaso de conceptos relacionados con análisis de lenguaje natural. Todo ello lo puse en común con mis compañeros con el fin de afianzar conceptos. Gracias a este trabajo, creamos una base de conocimiento que nos permitió entender partes del proyecto en las que no íbamos a trabajar de forma directa.

Llegados a este punto, decidimos distribuirnos el trabajo en función de los tres campos que he mencionado con anterioridad. Un grupo se encargó de la recogida de datos con análisis de lenguaje natural, otro del almacenaje de los mismos usando tecnologías distribuidas y, finalmente, un último en diseñar y construir el frontend de la aplicación. Por lo tanto, a partir de este momento, trabajé junto con Javier Mulero Martín en el desarrollo del frontend de la aplicación.

Dediqué una fase de trabajo a investigar y definir requisitos, ya con el objetivo del TFG determinado (implementar una página web que mostrase datos sobre situaciones de precariedad laboral, sin censura y en forma de dashboard). Esto me permitió entender mejor al usuario objetivo, así como el tipo de datos que queríamos mostrar y las características que debía tener nuestra aplicación.

Posteriormente, comenzamos con las fases de diseño e implementación, que ocuparon la mayor parte del proyecto. Diseñamos bocetos que fueron incrementando su grado de fidelidad (en papel y *Balsamiq*), a la vez que aprendíamos sobre desarrollo web. Cree un proyecto en *React* en el que fuí probando distintas librerías de visualización gráfica (*recharts*, *visx* o *nivo*) y diseño web (*Bootstrap*, *Material UI*...). Además, haciendo uso de plantillas y bibliografía, profundicé mis conocimientos sobre enrutamiento dentro de una página, componentes funcionales y uso de *Hooks*. Todo ello lo puse en común con Javier, quién compartió conmigo las pruebas que había realizado él.

Una vez establecido el diseño inicial, y tomando como referencia una plantilla, realizamos la primera implementación del dashboard. Cabe destacar que la mayor parte del desarrollo que hicimos a partir de este momento fue conjunto, haciendo uso de la herramienta *LiveShare* de *Visual Studio Code*, que nos permitía trabajar en el mismo proyecto en tiempo real.

El primer producto que creamos contaba con los siguientes componentes; 4 tarjetas para mostrar totales, 1 wordcloud, 2 tarjetas para mostrar rankings, 1 visualizador de tweets, 1 gráfica temporal y 1 línea de tiempo de doble eje. La estructura del proyecto, y los 4 componentes de totales, los implementamos juntos. El resto, los repartimos equitativamente. Yo me encargué del wordcloud, un ranking y la línea de tiempo. Para comprobar su funcionamiento, utilizamos un fichero de datos de prueba. Posteriormente, dichos componentes sufrieron modificaciones en las que participamos tanto Javier como yo.

El siguiente paso, fue ver cómo obtener los datos que estaba almacenando otro grupo en *IPFS*. En ese momento, nos dimos cuenta de que dichos accesos podían resultar muy costosos, por lo que decidimos construir una base de datos que funcionase como una "memoria caché". Es decir, esta BBDD se actualizaría cada dos días con datos de *IPFS* y sería la encargada de

proporcionar información a la página web. Por lo tanto, a partir de este momento, comenzamos a desarrollar tres proyectos en paralelo; la página web (*dashboard-twitter*), uno encargado de hacer consultas a la BBDD (*cache-twitter*) y un último encargado de actualizarla con datos de *IPFS* (*update-cache-twitter*).

La BBDD decidimos tenerla en *MongoDB Atlas*. Implementamos el proyecto (*cache-twitter*) para que, haciendo uso de *express.js*, pudiese conectarse con ella y realizar consultas. Dediqué un periodo de tiempo a aprender y familiarizarme con dichas tecnologías, para lo cuál fueron de especial utilidad los manuales de *MongoDB*. En este momento, dimos funcionalidad a un nuevo componente en el dashboard; los filtros por fecha. Aprendí cómo realizar e implementé consultas filtradas a la BBDD, así como peticiones *HTTP* que permitiesen conectar los proyectos. Además, profundicé mi conocimiento en el *Hook UseContext*, lo cuál me permitió utilizarlo para dar funcionalidad a los filtros que afectaban a todos los componentes de la página web.

Por otro lado, implementamos el script (*update-cache-twitter*) para que llamase a la función *get* de la API creada por el otro grupo, y obtener así los datos de *IPFS*. Este script se encarga de vaciar la "memoria caché" y meter los datos actualizados. Por lo tanto, en ese momento, implementé las llamadas *HTTP* al proyecto *cache-twitter* y añadí las consultas necesarias en él para modificar la BBDD.

Desplegamos los tres proyectos en *Heroku*, de forma que estuviesen en servidores y no en local. En este momento, todos funcionaban perfectamente y ya estábamos mostrando datos de *IPFS*. A partir de este momento, decidimos aumentar funcionalidades en el dashboard y realizar cambios de estilo.

Me encargué de incluir un componente que permitiese visualizar correlaciones a lo largo del tiempo de las palabras más utilizadas en denuncias. Además, añadimos un buscador al componente encargado de visualizar los tweets. Éste último permitía búsquedas por usuario y por palabras dentro de los tweets, y requería que los datos le llegasen ordenados. Por lo tanto, implementé las consultas necesarias sobre la BBDD, haciendo uso de expresiones regulares para el tratamiento del texto. Otras tareas que se llevaron a cabo fueron; realizar la página de información de la iniciativa, arreglar implementaciones realizadas que podían mejorarse con los nuevos conocimientos adquiridos, incorporar funcionalidades en las gráficas (zoom, enlaces a *Twitter*, personalización del contenido...), realizar un estilo amigable de la página web etc.

Una vez obtuvimos el producto final, llevamos a cabo un periodo de evaluación con usuarios objetivo. Creamos un plan de evaluación y realizamos entrevistas. Los resultados obtenidos, sirvieron para plantear posibles trabajos futuros. Además, durante esta última parte del proyecto, se llevó a cabo la redacción de la memoria. Formé parte, no sólo de su redacción, sino también de su organización y revisión.

4.2. Tecnologías

Dedicamos esta sección a exponer en detalle los recursos utilizados durante el desarrollo del proyecto, y cuál ha sido su finalidad. En la tabla 4.1 encontramos un resumen de las mismas, organizadas por la funcionalidad que les dimos, junto con su bibliografía.

Objetivo	Tecnología	Acceso documento	Averigua más
Edición y control de versiones	Diagramas de Gantt	4.2.1.1	[98]
	Overleaf	4.2.1.2	[70]
	GitHub	4.2.1.3	[30]
	PyCharm	4.2.1.4	[73]
	Visual Studio Code	4.2.1.5	[14]
Diseño	Balsamiq Wireframes	4.2.2.1	[120]
	GIMP	4.2.2.2	[29]
	diagrams.net	4.2.2.3	[20]
Desarrollo	API de Twitter	4.2.3.1	[106]
	Twitter for Websites	4.2.3.2	[107]
	API de Instagram	4.2.3.3	[41]
	Python y paquetes	4.2.3.4	[76]
	Metamask	4.2.3.6	[55]
	Infura y Etherscan	4.2.3.7	[39]
	Remix (Solidity)	4.2.3.5	[83, 92]
	Javascript y paquetes	4.2.3.8	[19]
	React	4.2.3.9	[79]
	Node.js y npm	4.2.3.10	[66, 68]
Almacenaje de datos	IPFS	4.2.4.1	[42]
	MongoDB Atlas	4.2.4.2	[57]
Despliegue	Heroku	4.2.5.1	[36]
	AWS	4.2.5.2	[3]
	Ropsten y Rinkeby	4.2.5.3	[86, 84]
Comunicación	HTTP	4.2.6.1	[37]

Cuadro 4.1: Tecnologías

4.2.1. Edición y control de versiones

4.2.1.1. Diagramas de Gantt

Es una herramienta gráfica que nos ha permitido organizar el tiempo de dedicación previsto para cada una de las tareas que teníamos que realizar. Además, también nos ha servido para visualizar la relación existente entre ellas y poder mejorar la planificación.

4.2.1.2. Overleaf

El desarrollo de la memoria se ha realizado utilizando el lenguaje LaTeX[48]. La edición y control de versiones se ha gestionado, colaborativamente, usando *Overleaf*.

4.2.1.3. GitHub

GitHub es un servicio basado en la nube que hemos utilizado para trabajar de forma conjunta en el proyecto. Nos ha permitido, no solo guardar el código y los cambios que se producían en él, sino también poder realizar un seguimiento y control de versiones.

4.2.1.4. PyCharm

PyCharm es un IDE de *Python* que nos ha permitido integrar el servicio de *GitHub* para subir y actualizar de forma eficiente los cambios. Cabe destacar la facilidad con la que se pueden descargar los paquetes de *Python* sin necesidad de comandos.

4.2.1.5. Visual Studio Code

Como editor de código, hemos utilizado también *Visual Studio Code*. En particular, ha resultado ser muy cómoda la funcionalidad *Live Share*, que nos ha permitido editar código colaborativamente, y en tiempo real, desde varios dispositivos.

4.2.2. Diseño de la aplicación

4.2.2.1. Balsamiq Wireframes

Balsamiq Wireframes es una herramienta desplegada en la web que hemos utilizado para diseñar los *wireframes* que componen la interfaz de nuestra página web. Nos ha permitido crear prototipos de baja fidelidad.

4.2.2.2. GIMP

GIMP es un editor de imágenes que nos ha permitido aumentar la fidelidad de los bocetos de nuestra aplicación.

4.2.2.3. diagrams.net

Para el diseño de la arquitectura, su estructura y funcionamiento hemos hecho uso de *diagramas.net*, una aplicación web.

4.2.3. Desarrollo de la aplicación

4.2.3.1. Twitter API

La API de Twitter consiste en una serie *endpoints* de programación que pueden ser usados para entender o crear conversaciones en Twitter. Esta se utiliza para recoger, interactuar y

analizar datos de Twitter. Algunos de estos son los tweets, usuarios, mensajes directos, listas y *trends*. Para poder acceder a ella, es necesario tener una cuenta de *Twitter developer*, posteriormente hay que registrar una aplicación y solicitar los *tokens* y *keys* correspondientes.

Para hacer uso de la API en *Python* hemos usado *Tweepy*, una librería que cuenta con numerosas formas de interactuar con la API (*stream*, *API*, *Client*...). Nosotros hemos utilizado la primera, para poder obtener tweets en tiempo real filtrados por nuestras necesidades. Observamos que, al no poseer una cuenta *Research*, no se ha podido utilizar esta librería para acceder al histórico de tweets.

Otra forma que hemos utilizado para acceder a esta API, ha sido haciendo uso del módulo *twitter-api-v2* [108] de *JavaScript*, para poder actualizar los metadatos de los tweets (retweets, likes y respuestas).

4.2.3.2. Twitter for Websites

Twitter for Websites proporciona una serie de herramientas para incluir contenido de Twitter en páginas web. Por ejemplo, *Web Intents* proporciona un direccionamiento para interactuar con tweets y usuarios concretos desde nuestra página web.

4.2.3.3. Instagram API

Al igual que la API de Twitter, nos permite recoger y analizar datos, aunque esta vez de Instagram. Para acceder a ella, hemos hecho uso de *Instagrapi* [41], una librería de *Python* que, además de facilitarnos el acceso, nos permite navegar tanto con API web pública, anónimamente, como móvil privada. Gracias a ella, hemos podido obtener la sección "descripción" de las publicaciones.

4.2.3.4. Python

Python es un lenguaje de alto nivel de programación interpretado. Debido a su popularidad, cuenta con gran cantidad de librerías. En este proyecto, hemos hecho uso de las siguientes:

- **Snscrape** [91]

  Permite *scrapear* de redes sociales distintos datos. En el caso de Twitter, es posible obtener usuarios, perfiles, hashtags, búsquedas, tweets o tendencias. Esta es la alternativa para acceder al histórico de tweets, al no tener acceso a la cuenta de *Research* de Twitter, y es usada en el programa scrape.py junto con una query formada por algunas palabras del diccionario que permite buscar palabras clave.

- **Pymongo** [74]

  Gracias a ella podemos trabajar con la base de datos *MongoDB* desde *Python*. Nos ha permitido, por ejemplo, solucionar problemas de concurrencia sobre archivos.

- **Stanza** [94]

  Es un paquete de Python de procesamiento del lenguaje natural (NLP) en numerosos idiomas. En este trabajo se utiliza para tokenizar y lemmatizar el texto.

- **Pandas** [71]

  Es una famosa librería de Python, muy utilizada en el campo de análisis de datos, que permite el fácil manejo de datos a través de estructuras de datos, como pueden ser los *dataframes* o las series. Nos permite cargar datos de distintos formatos y poder manipularlos. En nuestro caso ha sido empleado tanto para crear el diccionario a partir de un dataframe de 2 columnas sobre cada palabra extraída del diccionario mediante la función to_csv, como para poder cargar los datos del diccionario, con la función read_csv, cada vez que son requeridos por los programas.

- **Dotenv** [77]

  Facilita el manejo de variables de entorno, usadas para tratar con tokens de acceso de manera segura. Carga las variables de entorno de archivos .env al propio proceso, separando dichas variables del código. Esto lo consigue haciendo uso de las funciones *find_dotenv* y *load_dotenv*. En este proyecto es usado combinado con un .gitignore para no revelar los Tokens de acceso de la cuenta de Twitter Developer, que nos permite utilizar la API de Twitter.

- **Keyring** [46]

  Permite acceder al anillo de claves del sistema desde aquellas aplicaciones que requieren un almacenamiento seguro de contraseñas. Es necesario crear un usuario y una contraseña a través de la función *set_password* para, posteriormente, usarlos con la función *get_password*. Fue usado para el manejo seguro de la contraseña de la cuenta de Instagram utilizada para la API de Instagram.

- **Emoji** [23]

  Permite utilizar emoticonos en textos a partir de los códigos CLDR que los representan. Nosotros lo hemos utilizado, por ejemplo, para limpiar el texto de los tweets para su clasificación.

- **Statistics** [95]

  Permite hallar estadísticas matemáticas de datos numerables. Fue empleado cuando buscábamos un corpus para poder hallar las métricas del diccionario, ya que necesitábamos calcular el tamaño medio de los textos dado un corpus, pues era esencial que tuviera el tamaño y longitud más parecido al que íbamos a usar de denuncias.

- **Certifi** [12]

  Proporciona la colección de certificados raíz de *Mozilla* para poder validar la confianza en el certificado *SSL* mientras se verifica la identidad de los *hosts TLS*.

- **Spacy** [93]

  Permite el procesamiento de lenguaje natural. Lo hemos utilizado para tokenizar.

- **Codecs** [15]

  Define las clases base de codificación en *Python* y da acceso a su registro interno de codificaciones, lo que facilita el manejo de errores. Para este trabajo se usa para manejar el fichero donde se escriben las denuncias.

- **Subprocess** [97]

  Permite crear nuevos procesos, conectar a sus correspondientes tuberías de entrada/-salida/error y obtener sus valores de retorno. Este módulo se usa normalmente para reemplazar múltiples módulos y funciones antiguas.

- **Threading** [100]

  Módulo que permite el manejo de distintos *hilos* de ejecución. Es utilizado en el código *scrape* para obtener los tweets en un *hilo* distinto al que los clasifica, con el objetivo de incrementar la velocidad del programa.

- **Time** [102]

  Módulo con funciones de manejo de tiempos. Es utilizado para poder crear un retardo en la velocidad de recolección de *tweets* del código *scrape*, debido a la acumulación de *tweets* en la base de datos al ser insertados a mayor velocidad de lo que son borrados.

4.2.3.5. Remix y Solidity

Remix es un entorno de desarrollo de *Ethereum* que nos ofrece la posibilidad de compilar y desplegar contratos inteligentes, conocidos en inglés como *smart contracts*. Además también se puede usar para probar las funciones de los contratos, gracias a su maquina virtual de prueba.

La programación de los contratos inteligentes se ha llevado a cabo usando un lenguaje orientado a objetos conocido como *Solidity*.

4.2.3.6. Metamask

Metamask es una extensión para navegadores, o móvil, que actúa como una billetera (*wallet*) de criptomonedas. Utilizamos esta extensión para tener un medio de pago con el que efectuar las transacciones.

4.2.3.7. Infura y Etherscan

Infura es una plataforma que proporciona un conjunto de herramientas e infraestructuras que permiten a los desarrolladores llevar, fácilmente, su aplicación blockchain de la prueba, a la implementación a escala, con un acceso simple y confiable a *Ethereum*. La utilizamos como

infraestructura para nuestro proyecto.

Por otro lado, *Etherscan* [27] es una plataforma para explorar información de la blockchain de *Ethereum* [25]. La utilizamos para monitorizar y observar la información de nuestro contrato, ya que permite ver todas las transacciones que se han ejecutado y comprobar su correcto funcionamiento.

4.2.3.8. JavaScript

Gran parte del proyecto se ha desarrollado en *JavaScript*, un lenguaje de programación o de secuencias de comandos que se utiliza para implementar funciones complejas en web. Es una parte muy importante para nosotros porque con él realizamos, tanto la interacción con el contrato inteligente y la API, como gran parte del desarrollo *frontend*. Este último uso, debido a su magnitud, lo exponemos en el siguiente apartado de forma dedicada. De entre todos los módulos, frameworks y librerías utilizados, destacamos los más importantes.

- **Web3** [117]

  Permite interactuar con un nodo de *Ethereum* de forma local, *HTTP*, *IPC* o *WebSocket*. En nuestro caso, nos ha permitido interactuar con funciones de contratos inteligentes. Además, gracias a ella hemos podido conectarnos con una billetera de *Metamask*, y así poder emplear ether para llamar a las funciones del contrato que necesitan dinero para ejecutar la transacción.

- **Express** [28]

  Es un framework construido para *Node.js* que provee diferentes herramientas web y permite crear APIs. Cuenta con métodos como *post*, *get* o *delete* que nos han servido para conectar APIs de la página web, o tratar con BBDD en la nube.

- **Body-parser** [10]

  Se encarga de *parsear* las peticiones *HTTP*, lo que permite obtener los datos de las mismas en un formato adecuado.

- **Truffle-hdwallet-provider** [35]

  Es un módulo de *JavaScript* que permite crear conexiones con *Ethereum* firmando transacciones de direcciones que se le pasen. Usamos este módulo para poder realizar correctamente las transacciones entre la API creada por nosotros y el contrato desplegado de *Solidity* [92], ya que para poder llamar a las funciones es necesario que interactúen con una dirección válida.

- **Node-fetch** [64]

  Este módulo fue creado para realizar peticiones asíncronas del lado del cliente. Ha sido utilizado para conectar distintas partes del proyecto de forma asíncrona, gracias al *await*, que hace que el flujo de ejecución espere al resultado.

- **Node-Schedule** [65]

  Permite programar funciones para su ejecución en un tiempo determinado, usando el formato tipo *cron*. El formato consiste en una serie de números que representan: día de la semana (0 - 7), mes (1 - 12), día del mes (1 - 31), hora (0 - 23), minuto (0 - 59) y segundo (0 - 59, opcional).

4.2.3.9. React

React es una biblioteca *JavaScript* diseñada para crear interfaces de usuario en aplicaciones de una sola página (SPA). Permite, tanto la creación de vistas, como la renderización de las mismas en el navegador, sin necesidad de interactuar directamente con el *DOM*.

El desarrollo de aplicaciones en *React* se basa en la creación de componentes, trozos de código independientes que permiten separar y organizar el código en piezas lógicas. Existen tres tipos principales; los componentes elementales, de clase y los funcionales. En el desarrollo de nuestro proyecto, nos hemos centrado en los últimos. Se tomó esta decisión porque, actualmente y de cara al futuro, parece que son los más utilizados. Esto se debe a su simplicidad en código, y su capacidad de reutilización y unificación.

Los componentes funcionales son funciones que reciben *props*, datos que permiten definir su funcionamiento, y devuelven un *ReactNode*, un elemento que puede tomar muchos tipos distintos (*HTML*, *String*, *Boolean*...). Además, pueden contener estados, objetos que contienen información sobre el componente y que van cambiando con el tiempo, generando una renderización cada vez que se modifican.

Para poder acceder al estado y al ciclo de vida de los componentes funcionales, se introdujeron los *Hooks* en la actualización de *React* (v16.8.0). En nuestra aplicación, hemos hecho uso de los siguientes:

- *useState*: Permite crear un estado y nos proporciona la función necesaria para modificarlo.
- *useContext*: Permite definir datos que pueden ser utilizados por toda una jerarquía de componentes, sin necesidad de ser pasados por todos los niveles a través de *props*.
- *useEffect*: Permite realizar acciones una vez el componente se haya renderizado. Hemos hecho uso de él para crear nuevos *Hooks* personalizados. Por ejemplo, *useFetch*, utilizado para hacer consultas *HTTP*.
- *useRef* : Permite la manipulación directa de elementos del DOM.
- *useMemo*: Permite memorizar valores.

Para implementar los componentes, y aunque se puede prescindir de él, *React* cuenta con el lenguaje JSX. Es una extensión de *JavaScript* que busca unificar lenguajes de marcado

(*HTML*) y diseño (*CSS*).

Además, cabe destacar que, al estar *React* tan extendido, existen muchas bibliotecas diseñadas para utilizarse con él. Destacamos, a continuación, las más relevantes que hemos utilizado:

- *Material UI* [112]: Permite crear componentes siguiendo el diseño *Material Design* de Google.
- *React-ApexCharts* [6]: Permite la creación de gráficos y diagramas interactivos.

Finalmente, para el desarrollo de nuestra aplicación, hemos tomado como referencia una plantilla de Minimal Free [56]. Además, hemos hecho uso de la extensión de navegador *React Developer Tools* [80] para poder inspeccionar mejor las jerarquías de la aplicación y el procedimiento de renderización de cada componente.

4.2.3.10. Node.js y npm

Node.js es un entorno de ejecución de *JavaScript* de código libre y multiplataforma, pensado para facilitar la escalabilidad de aplicaciones de red, como servidores web que necesitan atender muchas peticiones simultáneas. Al contrario que el modelo de concurrencia basado en hilos, *Node.js* está basado en un solo hilo de ejecución y es orientado a eventos asíncronos, haciendo usos de funciones *callback* en lugar de esperar a funciones de entrada o salida, lo que evita bloqueos y permite seguir ejecutando el código. Se puede interpretar como que *Node.js* "delega el trabajo" a otras partes mientras continúa la ejecución.

Además, *Node.js* incorpora un sistema de gestión de paquetes *npm* (*Node Package Manager*), donde desarrolladores de código libre de cualquier parte del mundo comparten sus módulos para facilitar a la comunidad abstracciones y códigos más sencillos y seguros de usar. En nuestro proyecto, hemos utilizado una gran variedad de ellos [67].

Lo hemos utilizado para ejecutar el código *javascript* tanto en el front-end como en el back-end.

4.2.4. Bases de datos y almacenaje de información

4.2.4.1. IPFS - InterPlanetary File System

IPFS es un sistema de archivos distribuidos *p2p* (punto a punto) que busca conectar todos los dispositivos informáticos con el mismo sistema de archivos.

Por su diseño, está concebido para almacenar una gran cantidad de archivos de forma pública y resistente a la censura. Divide la información en fragmentos, que se almacenan en distintos nodos y a los que se accede a través de un identificador. Si se desea modificar un archivo, crea otro nuevo, por lo que la información guardada es resistente a cambios.

En nuestro proyecto lo hemos utilizado como principal fuente de almacenamiento para guardar la información de los tweets que contienen denuncias de forma distribuida. De esta forma evitamos guardarlo directamente en la blockchain puesto que esto tendría un coste inasumible y haría este trabajo muy poco escalable, además nos garantiza la inmutabilidad de los datos almacenados lo que aplicado a este contexto en el que se guarda información asociada a denuncias y quejas, generalmente hacia empresas que no quieran que esta información sea revelada o esté a disposición de un público mayoritario, hace que aquellos que ya hayan manifestado su opinión no puedan ser presionados a eliminarla.

4.2.4.2. MongoDB Atlas

MongoDB es una base de datos *NoSQL*. Es decir, no guarda los datos en tablas, como hacen las bases de datos relacionales, sino que utiliza estructuras *DBSON* (Binary JSON). Esto permite, por lo tanto, guardar estructuras de datos flexibles, y escalar fácilmente el tipo y la cantidad de datos que se quieren almacenar.

En nuestra aplicación, esto era un requisito fundamental, puesto que necesitábamos poder almacenar estructuras de tamaño variable (*p. ej.*: listas de hashtags), y tener la posibilidad de escalar fácilmente el tipo de datos (*p. ej.*: añadir nuevos campos).

Otras ventajas que tiene, y que también contribuyeron a su elección, fueron, por un lado, el poder usar una estructura de datos parecida al *JSON*, que es la utilizada tanto al recoger los datos, como luego en el front-end de la aplicación. Por otro lado, la gran diversidad de consultas que permite realizar sobre los datos. Finalmente, destacamos el fácil seguimiento que se puede hacer en todo momento sobre los datos que hay en la BBDD utilizando su interfaz.

En particular, nosotros hemos utilizado *MongoDB Atlas*, que proporciona el servicio de base de datos en la nube. Se utiliza para almacenar temporalmente los tweets obtenidos tras su recolección de Twitter, así como para guardar los datos de IPFS con el fin de alimentar a la página web.

4.2.5. Despliegue de la aplicación

4.2.5.1. Heroku

Para desplegar algunas de las partes de nuestra aplicación, utilizamos *Heroku*, una plataforma de servicios en la nube (PaaS) que da soporte a una gran cantidad de lenguajes de programación. Realizamos esta elección, entre otros motivos, porque *Heroku* se encarga de la gestión del hardware y los servidores, lo que nos daba a nosotros la libertad de dedicarnos, exclusivamente, al desarrollo de nuestra aplicación.

4.2.5.2. Amazon Web Services (AWS)

Para desplegar las partes de la aplicación que implementaban hilos en paralelo, utilizamos *AWS*. *Amazon Web Service* es un proveedor de plataformas bajo demanda de computación

en la nube. Estos servicios web ofrecen capacidad de computación distribuida y distintas herramientas software, mediante las granjas de servidores de *AWS*. Para no tener que ejecutar el código localmente, hacemos uso de dos servidores suyos que se encargan de ello.

4.2.5.3. Ropsten y Rinkeby

Son redes de prueba de *Ethereum* que han sido utilizadas para el despliegue de contratos inteligentes. Permite usar *ether* "de prueba" con el fin de no gastar dinero real. Gracias a ello se pueden experimentar y testear prototipos de aplicaciones descentralizadas antes de su posterior despliegue en la *mainnet* de *Ethereum*.

4.2.6. Comunicación dentro de la aplicación

4.2.6.1. HTTP

HTTP (Hypertext Transfer Protocol) es un protocolo de comunicación que permite transmitir información en la World Wide Web. Está orientado a transacciones que siguen un esquema de petición-respuesta entre cliente y servidor. Hemos hecho uso de él para comunicarnos con los servidores de nuestras BBDD y las aplicaciones desplegadas en *Heroku*.

Capítulo 5

Trabajo previo

Como se mencionó en la planificación, la primera parte del proyecto estuvo orientada a entender las tecnologías con las que íbamos a trabajar. Esto lo llevamos a cabo realizando y poniendo en común unos proyectos que recogemos en esta sección.

5.1. Lista de tareas - React

Para comenzar a experimentar con *React*, se propuso crear una aplicación de una página (*single page*) para gestionar listas de tareas pendientes. Se pusieron, como requisitos, poder crear, editar, y eliminar listas de tareas pendientes, además de crear, editar, eliminar y buscar tareas.

La idea de este proyecto era familiarizarnos con el framework de *React*, pues no lo habíamos utilizado previamente en ningún otro proyecto. Así comprendimos los conceptos de *componente*, *estado*, *contexto* y *formulario*, entre otros.

Ejemplo de un proyecto: https://github.com/jjavimu/To-do-lists.

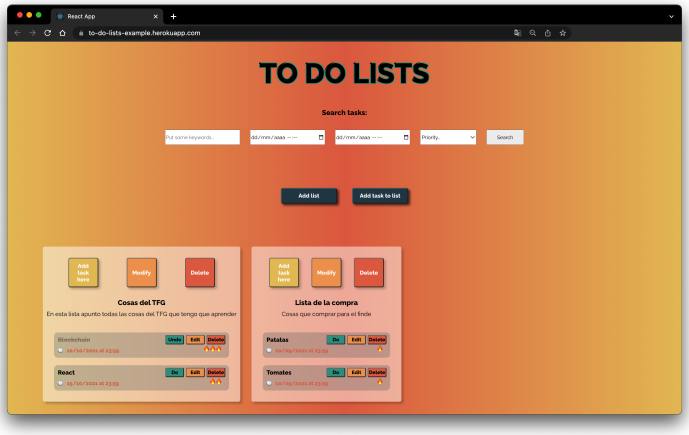


Figura 5.1: Aplicación de gestión de lista de tareas

5.2. Lotería - Blockchain y Solidity

El objetivo de este proyecto era aprender a usar el lenguaje de programación *Solidity*, gestionar el envío y la recepción de criptomonedas (ether) y entender el funcionamiento de la blockchain. Para desplegarlo, utilizamos *Remix*.

Este proyecto consistió en construir un contrato inteligente (*smart contract*) que permitiese realizar un sorteo de lotería. El contrato funciona de forma que, quién quiera participar, paga una cantidad de dinero al contrato. Una vez elegido un ganador al azar, el contrato le enviará a él todo el dinero acumulado.

Ejemplo de un proyecto: https://github.com/angrui02/lotery.git.

5.3. API de Twitter y análisis de lenguaje natural

Este ejercicio tenía como objetivo recolectar publicaciones de Twitter relacionadas con el tema de la precariedad laboral en España, y haciendo uso de la API de Twitter. Además de recordar algunos conceptos relacionados con el procesamiento y análisis del lenguaje natural.

Comenzamos dándonos de alta en su plataforma de desarrolladores. Intentamos conseguir una cuenta nivel *Academic Research*, que nos proporcionaba el acceso al archivo completo de Twitter. Sin embargo, sólo conseguimos *Elevated*, que permite obtener menos tweets al mes y no permite el acceso al archivo completo.

A continuación, elaboramos un vocabulario con algunas palabras claves relacionadas con el tema de la precariedad laboral. Esto nos permitió extraer un conjunto de tweets en español, sobre los cuáles obtuvimos su polaridad (la mayoría tenían negativa o neutra). Visualizamos las palabras que más se repetían, tras una limpieza del texto en la que se eliminaron símbolos, enlaces, palabras vacías etc.

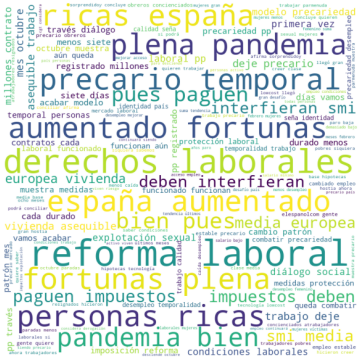


Figura 5.2: Wordcloud con las palabras más frecuentes de los tweets obtenidos

Capítulo 6

Investigación y recogida de requisitos

En este momento, comienza el desarrollo de nuestro proyecto propiamente dicho. Comenzó con una fase de investigación, seguida del establecimiento de los requisitos que debía cumplir la página web. Mostramos ambos en esta sección.

6.1. Investigación

Estuvo orientada al objetivo que pretendíamos alcanzar con nuestra aplicación; construir una página web que muestre datos sobre situaciones de precariedad laboral, sin censura y en forma de dashboard. La investigación consistió en determinar las características y objetivos del usuario, así como la situación del mercado al que pretendíamos entrar. Es por ello que realizamos un análisis de la competencia, tanto total como parcial, que nos permitió entender mejor cómo diseñar nuestro producto.

Características del usuario

En primer lugar, determinamos que el usuario que va a hacer uso de nuestra aplicación puede presentar características muy diversas. Las más relevantes:

- Persona que trabaja en un sector/empresa, o va a empezar a trabajar en él, y quiere conocer qué situaciones de injusticia se dan, le estén pasando o no a ella.
- Persona que presenta una predisposición a situaciones de injusticia, sea por su campo profesional, o por sus características físicas/psicológicas.
- Persona de autoridad que quiere conocer las situaciones que se dan en un sector/empresa. Puede ser con el objetivo de solucionar la injusticia o intentar silenciarla.

Análisis de la competencia

Por otro lado, el análisis de la competencia se centró en estudiar qué productos existían en el mercado. Estudiamos dashboards existentes, estuviesen o no relacionados con Twitter,

y buscamos organizaciones que llevasen a cabo proyectos con un objetivo similar al nuestro. A continuación, indicamos las principales herramientas que encontramos y que consideramos más resaltables para nuestro desarrollo.

- **TweetViz** [96]

  Herramienta web donde se muestran datos de tweets relacionados con una palabra (o palabras). Busca tweets que contengan dichas palabras y realiza un análisis del sentimiento de los tweets. Además, muestra distintas gráficas, temas relacionados, wordclouds... Los diferentes datos aparecen en diferentes pestañas, y todas ellas se ven afectadas por ese análisis del sentimiento. En la zona inferior de los gráficos, muestra información general acerca del uso de la web.

- **Twitter Analytics Dashboard** [4]

  Es una funcionalidad proporcionada por Twitter para ver la información acerca de la cuenta personal de cada usuario. El objetivo es hacer un seguimiento de los tweets y la interacción del usuario, con el fin de ver cómo repercute en la audiencia. Muestra información como las impresiones de los tweets, visitas al perfil y resúmenes mensuales del uso de la cuenta, entre otros.

- **Portwiture** [72]

  Proporciona una rejilla con imágenes obtenidas de Flickr que coinciden con palabras más usadas del contenido más reciente de un usuario dado.

- **TrackMyHashtag** [103]

  El uso de *hashtags* en Twitter ayuda a encontrar tweets relacionados con una temática. La herramienta TrackMyHashtag hace un seguimiento de un hashtag proporcionado y muestra estadísticas acerca de él, como las contribuciones al hashtag, impresiones o información acerca del lenguaje de los tweets.

- **Cuentas en redes sociales**

  En las propias redes sociales existen cuentas dedicadas a visibilizar denuncias. Por ejemplo, en Twitter está la cuenta @Jobsmierda [111], que se dedicaba a dar visibilidad a situaciones laborales precarias con un tono de humor. En Instagram encontramos @Trabajosruineros [40], que publica las denuncias que le envían sus seguidores junto con el nombre de la empresa dónde ha tenido lugar el incidente.

- **Aplicación jobstice** [43]

  Es una aplicación, desarrollada para teléfonos móviles, que permite denunciar de forma anónima abusos laborales.

6.2. Requisitos

En esta fase, haciendo uso de la información recogida durante la investigación, determinamos los requisitos que debíamos cumplir con nuestro dashboard. Fue un paso esencial para

poder establecer las funcionalidades que debía proporcionar la aplicación y realizar un diseño que se ajustase a ellas.

Comenzamos llevando a cabo un *brainstorming*, o lluvia de ideas, entre todos los integrantes del grupo. El objetivo era compartir las ideas preconcebidas que teníamos y, utilizando los resultados obtenidos en la fase de investigación, establecer unos requisitos comunes en los que pudiésemos basarnos.

Requisitos funcionales

En primer lugar, se establecieron los siguientes requisitos funcionales que determinan las interacciones que el usuario podrá realizar con nuestra aplicación:

1. Visualizar información sobre situaciones de precariedad laboral.
    a) Contenido de las denuncias.
    b) Cantidad de denuncias.
    c) Características de las denuncias; tipo (horas trabajadas, impago...), localización...
    d) Personas que realizan las denuncias.
    e) Cantidad de personas que realizan denuncias.
    f) Características de las personas que realizan denuncias (edad, género, profesión...)
    g) Reacción que reciben las denuncias por parte de la gente.
    h) Tendencias en las denuncias; a lo largo de los años, en función de la época del año, del tema...
2. Acceder a la fuente original de la información para poder interaccionar con ella.
3. Ordenar la información en función de varias métricas (tiempo, cantidad de denuncias, denuncias que mayor reacción generan...).
4. Filtrar la información que se muestra (tipo de información, tiempo, edad, género, localización...).
5. Descargar la información que se muestra.
6. Realizar búsquedas sobre la información (usuarios, temas...).
7. Poder poner denuncias.

Requisitos no funcionales

Por otro lado, se establecieron los siguientes requisitos no funcionales:

1. Garantizar que no haya censura. Es decir, aunque el contenido original sea eliminado, la página web debe seguir mostrándolo.

2. Mostrar contenido significativo.

3. Realizar un diseño visual, simple y fácil de usar por un usuario general.

4. Crear un sentimiento de comunidad y apoyo.

Para referirnos a estos requisitos, de aquí en adelante, utilizamos las abreviaciones RF (requisito funcional) y RNF (requisito no funcional). Los códigos que acompañan a dichas nomenclaturas se corresponden con la enumeración presentada. Por ejemplo, RF7 indica "Poder poner denuncias".

Capítulo 7

Diseño e implementación

El siguiente paso, consistió en utilizar los requisitos de la fase anterior para realizar el diseño y la implementación de la aplicación. En esta sección, recogemos los resultados obtenidos tras las sucesivas iteraciones que realizamos en esta fase de trabajo. Además, explicamos el desarrollo que nos condujo a la arquitectura y resultados finales de la aplicación.

Comenzamos viendo qué requisitos conseguimos cumplir y cuáles no en el siguiente cuadro 7.1. En el primer caso, mostramos cómo se implementaron y una breve explicación del mismo. En el cuadro 7.2, profundizamos en los requisitos que no pudieron ser cumplidos y el motivo de que esto ocurriese. Además, comentamos aquellos cumplidos pero que podrían ser ampliados.

A continuación, exponemos cómo se llevó a cabo el desarrollo de esta fase y qué proceso se siguió. Debido a la magnitud del proyecto, y la complejidad de cada una de sus partes, lo explicamos siguiendo la división en tres bloques mencionada en el capítulo 4. Cada uno de ellos, expone el proceso en orden cronológico. La arquitectura final obtenida tras este proceso, se puede encontrar en el capítulo 8.

7.1. Bloque 1: Recoger datos

7.1.1. Fase inicial

La idea partía de utilizar la API de Twitter para la obtención de tweets y llevar a cabo un proceso de análisis del sentimiento (*sentiment analysis*) para determinar si eran denuncias laborales. Comenzamos utilizando *textblob* [99] y *googletrans* [32] para traducir las publicaciones y calcular el sentimiento del texto traducido. Sin embargo, realizar tantas consultas a la API de *Google Traductor* resultaba ineficiente, además de que al no traducir el texto de forma literal, se perdían matices que afectaban al tono del tweet. Por ello, optamos por usar *afinn* [1], una librería de *Python* que nos permitía calcular el sentimiento en español.

El primer proyecto obtenía denuncias de cuentas y miraba si su sentimiento era negativo. Desarrollamos un prototipo inicial que recogía tweets de la cuenta *@JobsMierda* [111], que denunciaba en base a capturas de pantalla de conversaciones de *Whatsapp*, *e-mails* u ofertas

Requisitos funcionales	**Implementación**	**Explicación**
RF1a	AppTweets	Tweets que muestran el contenido de las denuncias
RF1b	AppTotalTweets	Total de denuncias realizas
RF1c	No cumplido	-
RF1d	AppTweets	Tweets que muestran quién los redactó
RF1e	AppTotalUsers	Total de usuarios que realizaron denuncias
RF1f	No cumplido	-
RF1g	AppTweets AppTotalFAV AppTotalRT	Tweets que muestran la reacción (comentarios, likes y retweets) que recibieron Total de likes que recibieron las denuncias Total de retweets que recibieron las denuncias
RF1h	AppWordsTime AppWordcloud AppHeatmap AppTimeline AppTopHashtags AppTopUsers	Progresión y correlación de temas denunciados en el tiempo Temas más denunciados Cantidad de denuncias en función de la época del año Cantidad de denuncias y reacción en el tiempo Temas más utilizados Usuarios más activos
RF2	AppTweets AppTopHashtags AppTopUsers	Enlace para escribir un tweet usando el hashtag `#Injustweet` Enlace para escribir un tweet usando el hashtag Enlace al usuario
RF3	AppTweets	Ordena denuncias (fecha ascendente/descendente o en función de la cantidad de likes/retweets)
RF4	FilterSidebar	Filtro temporal
RF5	AppWordcloud	Permite descargar información
RF6	AppTweets	Búsqueda de temas denunciados o usuarios
RF7	AppTweets AppTopHashtags AppTopUsers	Acceso directo a Twitter para denunciar Acceso directo a Twitter para denunciar Contacto directo al Twitter del usuario
Requisitos no funcionales		
RNF1	Uso de la blockchain	Guarda datos descentralizados
RNF2	Uso de Twitter	Plataforma con mucha actividad y datos
RNF3	Estilos y diseño AppInfo Texto explicativo	Usando la plantilla y Material UI, basado en Material Design Información sobre el proyecto Descripción del principal uso del dashboard
RNF4	Iniciativa y uso de un hashtag común: `#Injustweet`	Incentivar la denuncia de situaciones precarias motivadas por el dashboard

Cuadro 7.1: Requisitos cumplidos y cómo se han implementado

de trabajo con condiciones extremadamente precarias.

De esos tweets, mucha de la información relevante de la denuncia estaba en la imagen. Utilizamos los paquetes *pytesseract* [75] y *easyocr* [22] para capturar el texto de contenían las

Requisitos	Motivo
RF1c	Al no disponer de un corpus de documentos clasificados, no pudimos crear un modelo para entrenarlo, por lo que el tipo de las denuncias no aparece recogido. La localización no está disponible en todos los tweets. Además, los tweets se pueden publicar desde cualquier lugar, sin tener garantías de que ese sea el lugar origen de la denuncia.
RF1f	Se definió como requisito no prioritario para futuras implementaciones.
RF1h	Se podría ampliar la información sobre tendencias usando técnicas de *machine learning*
RF3	Se podría ampliar usando métricas como las características de la denuncia o del usuario
RF4	Se podría ampliar usando métricas como las características de la denuncia o del usuario
RF5	Se podría ampliar con una opción de descarga para cada gráfica

Cuadro 7.2: Requisitos no cumplidos o que se pueden ampliar y motivo del por qué

imágenes. Ambos nos plantearon los mismos problemas; iconos como el doble tick o la hora del mensaje formaban parte del texto. Esto no sólo hacía ilegible el mensaje, sino que además hacía que el texto no fuera válido para hallar su sentimiento. Además, las conversaciones les daba estructuradas como si se tratase de una sola línea. Esto rompía el sentido de la misma al no saber cuál de las dos partes dijo qué.

Encontramos también otros problemas. Numerosas publicaciones de esta cuenta no eran ni siquiera denuncias laborales y buscar en hilos tampoco auguraría un mejor resultado. Además, aunque el texto de la imagen fuera una denuncia, muchas veces el comentario que la acompañaba tenía un tono irónico o sarcástico, lo que resultaba en un *sentiment* positivo. Esto también ocurría al analizar conversaciones, puesto que muchos mensajes no tenían connotaciones negativas, como los saludos porque las condiciones laborales precarias redactadas como una oferta de empleo tendían a tener un sentimiento neutro.

Decidimos, por lo tanto, que un proceso de análisis del sentimiento no era lo adecuado para encontrar denuncias. Fue entonces cuando surgió la idea de crear un diccionario de palabras recogidas de denuncias preseleccionadas. De forma que pudiésemos usarlas para ponderar y determinar si un texto es una denuncia laboral.

7.1.2. Diccionario

Instagram

La cuenta de *Instagram @trabajosruineros* [40] contiene contenido que es, casi exclusivamente, denuncias laborales. Éste es redactado por usuarios y va acompañado de una imagen con el nombre de la empresa denunciada. Decidimos utilizarlo para crear el corpus y formar el diccionario ya que, al estar escritas por distintas personas, proporcionaban diversidad tanto en la redacción como en el contenido.

Utilizamos la librería *Instagrapi* de *Python* para poder interactuar con el API de *Instagram*, con el fin de extraer todas las palabras de las publicaciones que sí fueran denuncias laborales, siendo casi 800. De éstas, utilizamos cerca de 650 publicaciones, reservándonos 150 para poder formar un corpus con el que, más adelante, probaríamos el rendimiento de nuestro algoritmo.

Creación del diccionario

Una vez recogidas publicaciones de la cuenta, utilizamos técnicas de *NLP* para quedarnos con la información de la palabra que nos interesaba. Inicialmente, aplicamos procesos de *lemmatization, stemming y tokenization* a las publicaciones. Sin embargo, decidimos prescindir de la técnica *stemming* por la poca explicabilidad del diccionario. Esto se debía a que en ciertas ocasiones las raíces de las palabras no permitían deducir de qué palabra derivada había surgido. Una vez recogidas todas las palabras, llevamos a cabo un proceso de limpieza.

Inicialmente tomamos las palabras con mayor frecuencia, ya que debían ser las más usadas para realizar denuncias. Sin embargo, esto llevó a un diccionario de cerca de 8000 palabras, en dónde las que más se repetían eran verbos y palabras de uso común, que podían usarse en cualquier contexto. Es por esto que, la primera iteración, consistió en deshacernos de ellas, quedando un diccionario de 200 palabras.

En la siguiente iteración vimos que había palabras que estaban directamente relacionadas con temas laborales, pero que podían usarse en muchos más contextos. Decidimos quedarnos con las palabras más específicas, por lo que el diccionario se redujo a 75 palabras. Observamos el mismo en las figuras 7.1a y 7.1b.

Finalmente, realizamos una última iteración, dónde nos quedamos con las palabras que considerábamos necesarias en una denuncia laboral. El diccionario obtenido de 25 palabras se observa en la figura 7.1c. Éste tan sólo lo usamos para realizar consultas a la *API* de *Twitter* y al Scraper Web, con los tweets que nos devuelve, evaluamos con el diccionario de la segunda iteración si son denuncias o no. Estas consultas son necesarias para disminuir los tiempos de obtención de posibles denuncias laborales, puesto que estrechan el abanico de resultados.

Método de evaluación del diccionario

Tras terminar el diccionario buscamos un método de evaluación para el algoritmo. Partimos con la idea de ponderar las palabras usando un *ranking inverso*; las palabras en una posición mayor serían las que más ponderaran. Decidimos operar sobre el porcentaje de texto que estaba recogido en el diccionario para que no penalizar a los textos más cortos. Sin embargo, el método planteaba un problema. Si el texto era lo suficientemente corto se podrían dar muchos falsos positivos.

WORD	FREQUENCY
pagar	603
contrato	557
empresa	549
tienda	522
encargado	381
horario	285
cobrar	281
jefe	279
turno	219
compañero	212
sueldo	189
compañera	189
extra	183
jornada	172
dinero	171
descanso	163
empleado	158
euro	154
contratar	149
entrevista	129
semanal	127
laboral	123
puesto	121
vacaciones	121
formación	113
despedir	97
jefa	96
bronca	91
nómina	81
gerente	69
manager	67
convenio	65
dueño	64
festivo	63
finiquito	63
despido	60
camarero	59
fijo	59
baja	54
oficina	52
seguridad	51
salario	50
denunciar	50
pagado	49
librar	48
amenazar	43
renovar	39
ilegal	38
supervisor	36
superior	35
acoso	32
legal	32
explotar	30
explotador	28

(a) 2ª iteración, parte 1

mensual	27
amenaza	27
queja	24
sindicato	24
indefinido	24
machista	21
explotación	19
reclamar	19
paro	18
precariedad	16
remunerado	15
abuso	15
cobro	14
nomina	14
maltrato	14
fichar	13
despedido	13
indemnizaci	13
precario	12
improcedent	10
cláusula	8

(b) 2ª iteración, parte 2

WORD	FREQUENCY
contrato	557
jefe	279
jornada	172
contratar	149
despedir	97
nómina	81
convenio	65
finiquito	63
despido	60
denunciar	50
amenazar	43
renovar	39
ilegal	38
acoso	32
explotar	30
explotador	28
amenaza	27
sindicato	24
explotación	19
precariedad	16
abuso	15
despedido	13
indemnización	13
precario	12
improcedente	10

(c) 3ª Iteración

Figura 7.1: Algunos de los diccionarios obtenidos

Decidimos, por lo tanto, no tener en cuenta el porcentaje del texto que está en el diccionario, sino el porcentaje del diccionario que estaba contenido en el texto, lo que nos llevó a trabajar con un número de palabras absoluto. Ciertamente esta solución sí que penalizaría el tamaño de los textos, pero sólo hasta un punto lógico siempre que el número de palabras fuera alcanzable (los textos muy cortos se verían más penalizados, pero en parte sería de manera

lógica, un texto con sólo 4 o 5 palabras difícilmente es una denuncia bien redactada).

7.1.3. Desarrollo

API de Twitter

Desarrollamos dos formas de recolección de tweets. La primera, mediante el uso de *streaming*, que nos permite recoger tweets en tiempo real (lo que serían un par de minutos de antigüedad). La segunda, haciendo uso del módulo *snscrape*, un scraper de redes sociales con el que se puede recoger tweets de una mayor antigüedad (pudiendo ser de diferentes años). Para ambos métodos, la consulta con la cuál limitar resultados y reducir tiempos, se compuso de las palabras del diccionario de la tercera iteración.

Ambos códigos funcionan de una manera similar. Los dos recogen tweets, en base a una consulta compuesta por el operador 'OR' sobre ese conjunto reducido del diccionario, y los van almacenando en una base de datos. Mientras tanto, otro hilo obtiene los datos de la base de datos y determina mediante el algoritmo anteriormente mencionado si se tratan o no de denuncias laborales, en cuyo caso se recogen en un archivo *JSON*. Finalmente, este hilo elimina los datos de la base de datos. A partir de un cierto número de denuncias, acorde a la tasa de actualización de los datos de la *blockchain* en la web y al coste de las subidas a *IPFS*, se suben mediante el API que conecta con todo el sistema *blockchain*.

En el método basado en *streaming* se utilizan dos hilos. El primero, un hilo demonio, se encarga de recoger todos los tweets que correspondan con la consulta que le pasamos. Esta consulta consiste en un primer filtrado de tweets, donde pedimos que contengan alguna palabra de la tercera iteración del diccionario. En cuánto este hilo escucha algún tweet lo almacena en la base de datos. El otro hilo, mientras tanto, se encarga de consultar de forma constante la base de datos y de eliminar los tweets que se encuentran en ella, pero antes, por cada publicación que contenga, aplica el algoritmo para ver si se trata verdaderamente de una denuncia. En caso de serlo, la almacenará en formato *JSON*. El *scrape* funciona de forma similar al *streaming*, pero en vez de realizar la consulta a la API de Twitter mediante un *listener* lo hace a través de un *scraper web*.

Base de datos

Como se menciona en el apartado anterior, hacemos uso *MongoDB Atlas* para almacenar todos los tweets que recibimos de la API de Twitter. La razón de que utilicemos una base de datos es el problema de concurrencia al que podría someterse el sistema si los almacenáramos en un fichero simple. Dicho archivo sería escrito y borrado a la vez, todo en un mismo bloque de datos, lo que podría dar resultados inesperados. Por esto elegimos una base de datos, al ser sistemas preparados para problemas de concurrencia.

Estructura de los datos

En la figura 7.2 se observan los datos recogidos de cada tweet. Datos como el número de *likes, retweets y replies* son altamente susceptibles de cambiar con el tiempo. Por esta razón, se guarda el identificador del tweet.

```
{
  "link": "https://twitter.com/Milenio/status/1524192069341884416",
  "id": "1524192069341884416",
  "text": "Controladores denuncian jornadas de trabajo extenuantes que comprometen la seguridad de los vuelos y la salud de los empleados; en caso de quejarse sufren acoso laboral  Algunos usuarios se niegan a viajar mediante el AIFA  #Milenio19h con @pedrogamboamr https://t.co/QZbaCIXp2k",
  "user": "Milenio",
  "date": 1652230682,
  "likes": 0,
  "retweets": 0,
  "replies": 0,
  "hashtags": [
    "Milenio19h"
  ]
}
```

Figura 7.2: Archivo JSON con los datos de un tweet

- **link**: Enlace al *tweet* para poder consultar en cualquier momento los datos del *tweet*, además de poder visualizar otros datos que no recogemos como las respuestas de los *tweets*.
- **id**: Identificador del *tweet* que permite acceder a su contenido utilizando *APIs* de *Twitter*.
- **text**: El texto del tweet levemente modificado, debido a problemas de formato con el *JSON*. Entre estas modificaciones se encuentran la sustitución de saltos de línea por espacios y la agregación a las comillas dobles inglesas de barras invertidas.
- **user**: El usuario que escribió el tweet.
- **date**: La fecha en la que se publicó el tweet.
- **likes**: El número de likes del tweet.
- **retweets**: El número de retweets del tweet.
- **replies**: El número de respuestas del tweet.
- **hashtags**: Los hashtags contenidos en el texto del tweet.

Despliegue

Una vez el código funcionaba en entorno local, buscamos un servicio de *cloud computing* en el que desplegarlo. Inicialmente nos planteamos usar *Heroku*. Sin embargo, cuando tratamos de instalar todas las dependencias de *Python* para que pudiera subirse el código, tuvimos problemas de *slug size*. El compilador *slug* de *Heroku* lleva a cabo un proceso de compresión para poder subirlo a la nube, un problema bastante recurrente es superar el tamaño máximo limitado a 500MB. La librería que ocupaba tanto tamaño era *stanza*, por lo que cambiarla no era una opción tras la exhaustiva búsqueda que supuso encontrar una librería en español de NLP.

Recurrimos a *Amazon Web Services*, en dónde la cuenta de *Amazon Student* nos proporcionaba acceso a varios GB de almacenamiento y a instancias EC2 de tipo t2.micro. Sin embargo, este tipo de instancias no eran suficientes para nuestros código, ya que cuentan únicamente con 1 hilo por CPU y una única CPU. Por lo que decidimos utilizar las instancias t3.micro con 2 hilos por CPU para ejecutar nuestro código. La conexión con la misma se hizo usando el protocolo *SSH* junto con unas claves para entrar en su entorno.

Para el correcto funcionamiento de toda esta sección fue necesario manejar los distintos permisos de seguridad de todos los servidores, como sería permitir ciertos tipos de tráfico red (*SSH, TCP, HTTP y HTTPS*), así como crear un único usuario con máximos permisos en la base de datos.

Subida de datos a la Blockchain

El equipo encargado del bloque 2, nos facilitó un código en *javascript* que, mediante un método POST, permite subir archivos a *IPFS* y guardar los *IDs* de los tweets en la *blockchain*. Para ejecutarlo en nuestro programa *Python*, inicialmente, utilizamos la librería *Js2Py* [44], *Javascript to Python*. Ésta permite traducir código de *Javascript* a *Python* o incluso ejecutar el código *Js* en una máquina virtual desde *Python*. Sin embargo, no conseguimos que funcionara correctamente, por lo que terminamos usando el módulo *subprocess*, que permite crear procesos hijo de lenguajes de programación distintos del proceso padre. Gracias a este módulo, llamamos al intérprete de comandos de *Node*, al que se le pasa como parámetro el código *Javascript* para que sea ejecutado.

7.2. Bloque 2: Guardar datos

7.2.1. Fase inicial

Comenzamos con una investigación de librerías y tecnologías para manejar contratos inteligentes y almacenar información. Para ello, también estudiamos otros proyectos relacionados, como uno que hace uso de la blockchain aplicada a cuestiones de agricultura [34]. Gracias a ello, pudimos familiarizarnos con tecnologías como *Remix*, *Node.js* y el sistema de archivos

descentralizado *IPFS*. La elección de éste último fue una decisión de diseño, con el objetivo de evitar la censura mediante el almacenaje descentralizado de los datos.

Comenzamos implementando un contrato inteligente simple para familiarizarnos con su funcionamiento. Desarrollado en *Solidity* [24], tenía dos métodos; un *set* para almacenar un entero en una variable, y un *get* para devolverlo. Inicialmente, intentamos realizar el despliegue del contrato usando *Truffle*, pero finalmente nos decantamos por *Remix*, debido a su sencillez. La diferencia de utilizar Truffle y Remix es que con éste último una vez compilado el programa, teníamos que almacenar el código binario del mismo; el ABI (Application Binary Interface). El ABI lo guardábamos en un archivo independiente de la aplicación y lo llamábamos cuando creamos la instancia del contrato en *Node.js*. Además nos permitía usar *React* para comprobar su correcto funcionamiento a través de una interfaz.

Haciendo uso de la extensión de navegador *metamask*, creamos una billetera con ethers de prueba para poder ejecutar las transacciones. Éstos los conseguimos de páginas web en las que bastaba con introducir la dirección de la billetera en la que quieres recibir los tokens. Encontramos varios problemas. Para la red de prueba de *Ropsten* era complicado obtener estos ethers. Además, tuvimos que cambiar a la red de prueba *Rinkeby*, ya que *Ropsten* incrementó el coste de las transacciones en gas.

Una vez estuvo el contrato desplegado y funcionando, comenzamos a instalar las dependencias del proyecto. Instalamos la librería *Web3*, para interactuar con la blockchain y la cuenta, así como *Ipfs-http-client* para interactuar con *IPFS*.

Utilizamos *javascript* para empezar a interactuar con el contrato inteligente. Hicimos uso de *Infura* como infraestructura para las llamadas al contrato y puerta de enlace a *IPFS*. Tras crear la instancia del contrato, gracias a la dirección del mismo, realizamos llamadas de prueba a sus funciones. Para ello llamábamos a *set*, pasando un número cualquiera como parámetro. A continuación, invocábamos *get* para devolverlo por consola. Tras varios errores, como no definir correctamente el gas utilizado en las transacciones, conseguimos realizar un buen uso de las funciones del contrato.

Tras tener el código en *javascript* enlazado al contrato, comenzamos a realizar pruebas con *IPFS*. Comenzamos con pruebas sencillas, como subir "hola mundo" gracias al método *add*, guardar el *hash* que devolvía la función o declarar el método *cat* para descargar la información de *IPFS* usando como parámetro el *hash* anterior. Observamos que la información descargada de IPFS es necesario decodificarla, ya que viene distribuida en *chunks*.

7.2.2. Desarrollo

Contrato inteligente de la aplicación

Comenzamos a desarrollar el contrato inteligente que íbamos a usar en la aplicación. En la figura 7.3 se aprecian los tipos de datos que utilizamos; una estructura y un array de estruc-

turas. Cada estructura tiene el *hash* de *IPFS* y un array con los identificadores de los tweets que almacena ese fichero de *IPFS*. El otro atributo que tiene es un mapa que tiene como clave el identificador del tweet y como valor la posición de éste en el array de estructuras anterior.

```
1 struct TweetsFile {  // represent a file with tweets stored on IPFS
2     string IPFShash; // the hash of the file
3     string[] tweetID; // array with ID's of tweets on this IPFS file
4 }
5 TweetsFile[] tweetsFiles; // array with all of our files stored on
      IPFS

7 mapping(string => uint) storedTweets; //a map with all the tweetsID
      and the file in which they are
```

Figura 7.3: Estructura de datos del contrato

Al tener guardados el conjunto de *hashes* devueltos por *IPFS* al subir información, junto a los identificadores de los tweets, podíamos subir varios archivos a *IPFS* con denuncias en lugar de uno sólo. Añadimos las funciones *getFile* y *setFile* para obtener y alterar el array.

Desarrollo de la API

Una vez enlazamos el código *javascript* que se encargaba de hacer las llamadas al contrato, el contrato de la blockchain e *IPFS*, buscamos una plataforma que nos permitiese tanto obtener las denuncias proporcionadas por el bloque anterior, como mandárselas a la página web. Desarrollamos una API *REST* usando *express*. La API cuenta con dos métodos; un método *post* que obtiene la información de las denuncias para poder almacenarla, y un método *get* que envía un archivo *JSON* limpio a la aplicación web.

El primer cambio que tuvimos que realizar es dejar de utilizar *React* para visualizar las acciones, ya que el objetivo era automatizar el proceso de envío y recepción de información y no se necesitaba en este bloque una interfaz gráfica. Esto supuso volver a crear el proyecto, esta vez en *javascript* puro, reutilizando código.

Por otro lado, enfrentamos un problema cuando la versión de *Node.js* que estábamos usando se actualizó y dejó de ser compatible la unión realizada entre *express* e *IPFS*. Además, tuvimos que solucionar problemas de dependencias entre módulos, como las incompatibilidades entre "import" y "require", lo cuál conseguimos cambiando el fichero de configuración de *npm*.

En este momento, la API comenzó a funcionar correctamente.

- El método *get* de la API llama a *getFiles* del contrato, y obtiene así los *hashes* de *IPFS* junto a los identificadores de los tweets. A continuación, los descarga de *IPFS*, decodifica

y les da un formato de lectura correcto. Por último, se devuelve la información en un archivo *JSON* para que pueda ser utilizada por la página web.

- El método *post* funciona de forma que almacena en *IPFS* un archivo *JSON*. *IPFS* devuelve el *hash* y mediante el método *setFile* del contrato almacena el *hash* junto con los identificadores de los tweets. De esta forma, nos pueden devolver la información de las denuncias para almacenarla en la blockchain correctamente.

Despliegue

La API realizada funcionaba en nuestra máquina local. Para que los bloques de recoger datos y mostrar datos pudiesen interaccionar con nosotros, necesitábamos que fuese accesible desde internet a través de una IP pública. Decidimos utilizar *Heroku*, una plataforma de servicios en la nube. Una vez desplegado el servidor, ambas partes podían hacer uso de la API con los métodos *get* y *post*. Observamos que para enviar datos a la API, creamos un pequeño código en *javascript* que es capaz de enviar peticiones *post* con el fichero que se tenga que subir.

Actualización de los datos

Una vez tuvimos la aplicación funcionando, nos planteamos cómo mantener la información de las denuncias actualizada. Algunos metadatos, como el número de likes y retweets, cambian con el tiempo, por lo que teníamos que garantizar que estuviesen actualizados. Añadimos la función *updateTweets* al contrato inteligente, que se encargaba de actualizar la información de los tweets para, posteriormente, subirla a *IPFS*. No se eliminan los *hashes* anteriores en la estructura de datos para garantizar inmutabilidad y que se pueda consultar el estado de los tweets en todas sus fases. Desde el código en *javascript*, haciendo uso de la API de Twitter, obtenemos información de los tweets usando su identificador; de esta manera comprobamos qué tweets han cambiado sus metadatos y los actualizamos. El texto original del tweet nunca será modificado. Finalmente, se modificó el código *get* de nuestra API para que en la respuesta no devuelva los tweets y así no tener los mismos tweets con diferentes métricas dejando únicamente las más recientes.

El principal problema de este apartado deriva de las limitaciones que tiene la API de Twitter, ya que tiene un límite de tweets para su verisión 2 y la cuenta que utiliza un bearer token para conectarse tiene permitida la consulta de 300 tweets cada 15 minutos. Para solventar esta situación, el método desarrollado para actualizar los tweets los módifica en caso de ser necesario en tandas, de esta forma, realizando sucesivas llamadas al método de actualizar acabará por no tener más datos incorrectos. Por esta razón tuvimos que crear variables globales, para saber en todo momento en qué punto de la actualización de los datos se encuentra, esto implica el fichero se está actualizando, el último tweet actualizado y el número de tweets consultados a la API para no sobrepasar el límite. Como ayuda para realizar esta tarea, creamos un *cron*. Este *cron*, se encarga de realizar llamadas a nuestra API de forma indefinida hasta que calcula que todos los tweets han sido actualizados, esto lo sabe comparando el número

de llamadas que se hacen junto con el número de tweets que hay almacenados y sabiendo el número de tweets que se actualizan por llamada. Una vez todos los tweets están actualizados se para, y deja pasar un tiempo establecido antes de volver a funcionar. Observamos que un inconveniente que plantea esta decisión; la información se va a subir a IPFS varias veces para tenerlo todo bien actualizado, porque dicho contenido no puede ser eliminado y reemplazado.

Últimos contratiempos

El proyecto ya estaba funcionando, pero surgieron algunos contratiempos a los que tuvimos que hacer frente:

- Tuvimos que corregir el formato y tipo de los datos que se enviaban a la API.
- Hubo un agotamiento en la cantidad de ethers de prueba que teníamos para desplegar contratos o ejecutar sus funciones. Tuvimos que buscar nuevas páginas web que nos enviasen tokens.
- Las cuentas que conseguimos para utilizar la API de Twitter están muy limitadas,lo que dificultó la actualización de los datos

7.3. Bloque 3: Mostrar datos

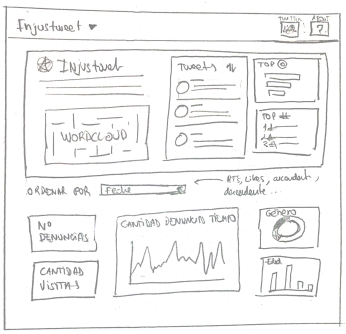
7.3.1. Diseño

Las distintas iteraciones de la fase de diseño, produjeron una serie de bocetos que fueron aumentando en su grado de fidelidad progresivamente.

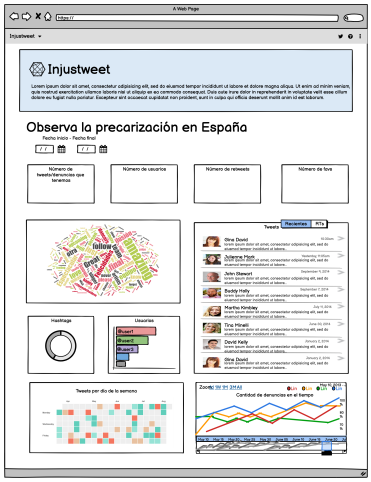
Comenzamos elaborando una gran cantidad de bocetos de baja fidelidad a papel, teniendo en cuenta los requisitos que habíamos obtenido en la fase anterior. A continuación, realizamos una fusión de las características que más nos gustaban de cada uno, obteniendo así el boceto inicial. En la Figura 7.4a podemos verlo.

Posteriormente, gracias a los avances realizados en la fase de implementación, y a una reunión de evaluación que tuvimos con el resto del equipo, decidimos realizar cambios y mejoras. Para ello, diseñamos un boceto de mayor fidelidad con *Balsamiq*. Esta herramienta, a través de la creación de *wireframes*, nos permitió entender mejor cómo trasladar nuestra idea a la aplicación. Se puede ver en las Figuras 7.4b y 7.4c.

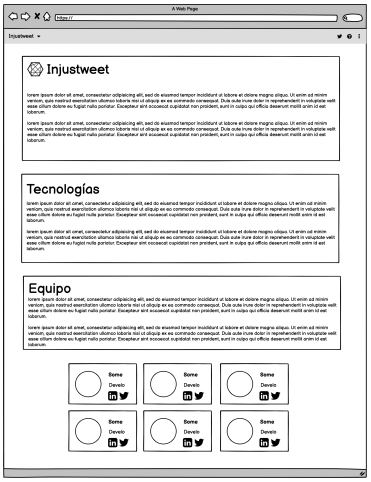
A partir de este punto, el resto de fases de diseño consistieron en ir realizando cambios progresivos sobre la aplicación. En las iteraciones que involucraron mayor nivel de cambios, como introducción de nuevos componentes, utilizamos *GIMP* para probar distintas reorganizaciones y características. En la Figura 7.5 podemos ver algunos de los cambios que fue sufriendo la aplicación.



(a) Boceto inicial en papel



(b) Boceto del dashboard en Balsamiq



(c) Boceto de la página de información en Balsamiq

Figura 7.4: Evolución de los bocetos

(a) Prototipo sobre la aplicación

(b) Redistribución de elementos en GIMP

Figura 7.5: Evolución del dashboard

El diseño final obtenido para nuestra página web se encuentra en las figuras 7.6 y 7.7. Observamos que durante todas las iteraciones, que finalmente nos han conducido a él, se han tenido en cuenta los 10 principios de diseño de Jakob Nielsen [62]. Comentamos algunos de ellos a continuación:

- Visibilidad del estado del sistema: La aplicación intenta mostrar al usuario en todo momento qué está pasando y en qué punto de la navegación se encuentra. Por ejemplo, se ha incluido un esqueleto de página web para que el usuario sepa que está cargando.
- Adecuación entre el sistema y el mundo real: El sistema intenta hablar con el mismo lenguaje que los usuarios. Por ejemplo, se utiliza el símbolo de calendario para indicar el

filtro por fechas.

- Libertad y control por el usuario: Se permite que los usuarios puedan volver fácilmente a un estado anterior. Por ejemplo, al realizar una búsqueda se pueden eliminar fácilmente los filtros introducidos.
- Consistencia y estándares: Repetimos patrones para no confundir a los usuarios. Por ejemplo, al ser datos de Twitter, utilizamos una temática similar, asi como su lenguaje (likes, hashtags...).
- Estética y diseño minimalista: Se ha intentado simplificar y eliminar contenido irrelevante para que el usuario sólo se fije en lo realmente importante.

7.3.2. Implementación

Tras el desarrollo de los primeros bocetos iniciales, nos planteamos cómo implementar dichas ideas. La primera decisión que se tomó fue hacer uso de *React*. Las principales referencias que hemos utilizado para utilizar *React* fueron la documentación oficial de su web [79] y el libro *Learning React* [7].

7.3.2.1. Fase inicial

En primer lugar, dedicamos un periodo de tiempo a familiarizarnos y buscar tecnologías que nos permitiesen incorporar en nuestra página web las ideas determinadas en la fase de diseño. Recordamos que nuestro objetivo era mostrar datos en forma de dashboard.

Por un lado, investigamos qué librerías existían compatibles con *React* que nos permitiesen implementar gráficas y diagramas para visualizar datos. Encontramos una gran cantidad de ellas (como *recharts* [82], *charts.js* [13], *visx* [2] o *nivo* [63]) con las que realizamos un proyecto de prueba.

En paralelo, buscamos cómo realizar un dashboard con dichas gráficas en *React*. Tuvimos un primer acercamiento a *React-Bootstrap* [11], una biblioteca de código libre con herramientas *HTML*, *CSS* y *javascript* para crear páginas y aplicaciones web. Esta librería nos aportaba bastante facilidad a la hora de diseñar la web, porque nos permitía no tener que crear los componentes básicos desde cero. Creamos otro proyecto basándonos en una plantilla [101] para entender su funcionamiento y que nos permitió aprender sobre enrutamiento en *React*, y sobre el lenguaje de estilos *sass*. Sin embargo, investigando y haciendo pruebas con *Bootstrap*, nos encontramos con *Material UI*, un framework para *React* basado en *Material Design*. Nos decantamos por este último por ser mucho más personalizable y poseer una documentación muy extensa.

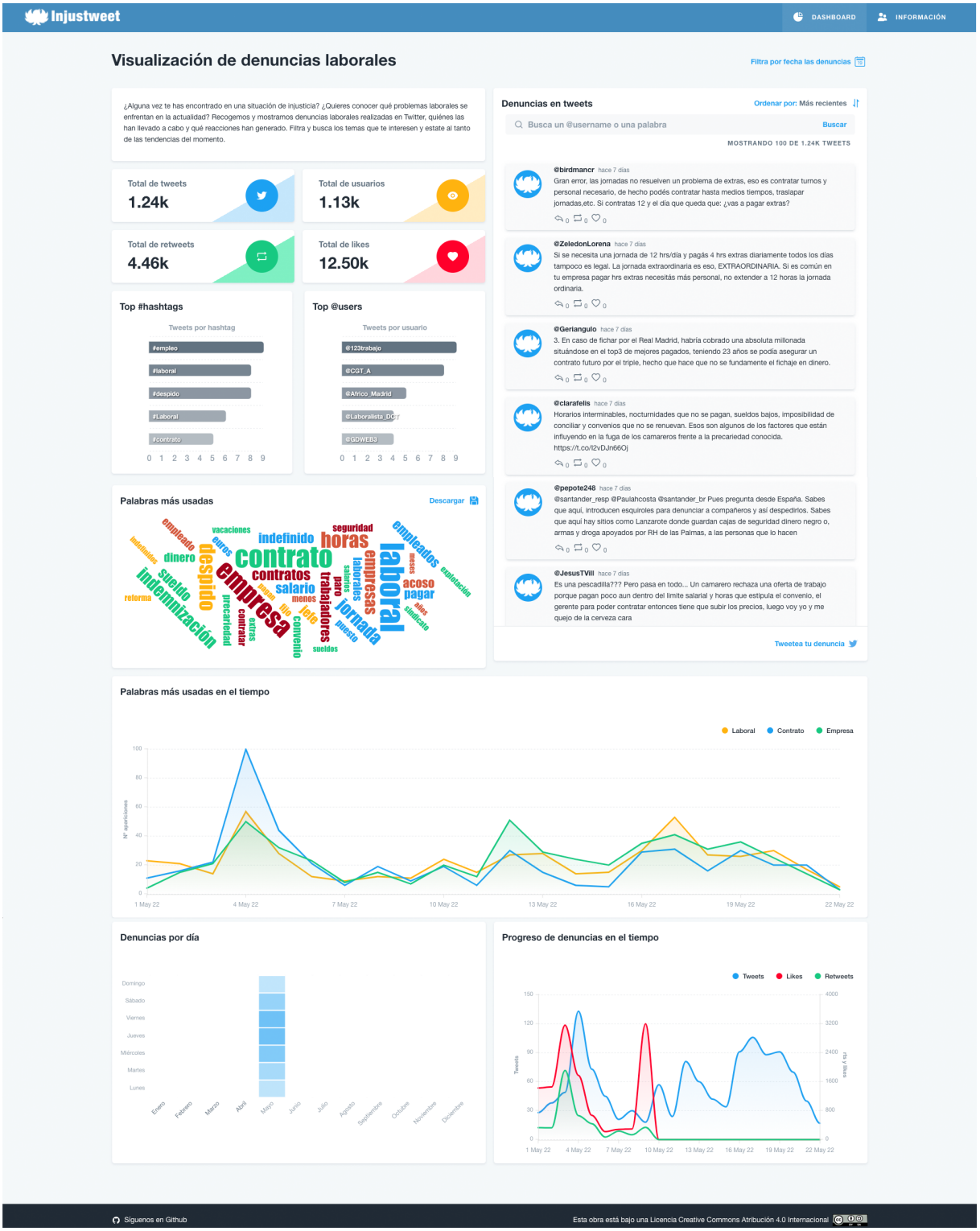


Figura 7.6: Dashboard final

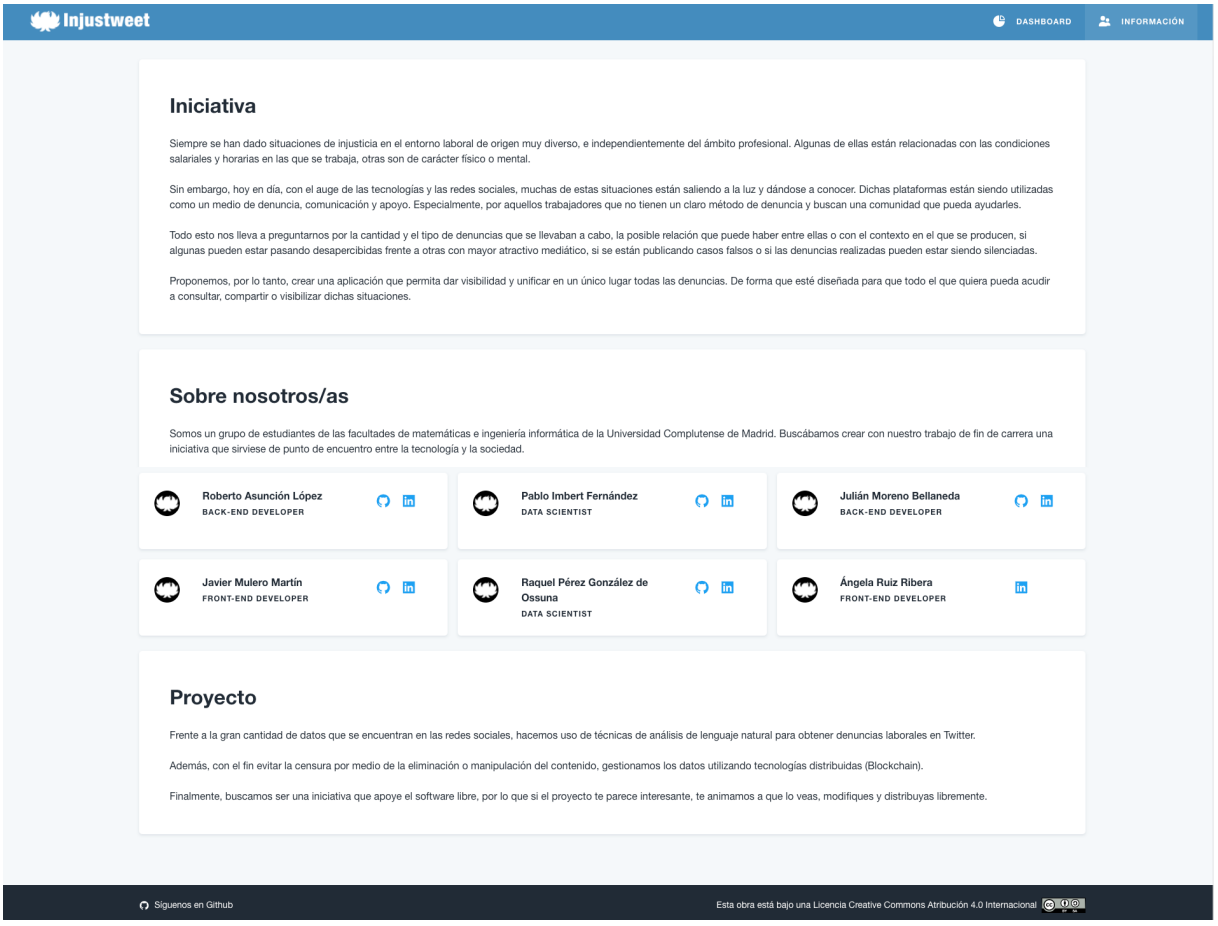


Figura 7.7: Página de información final

7.3.2.2. Desarrollo

Creación del dashboard

El resultado de la fase anterior fue un pequeño proyecto que recogía algunos de los requisitos expuestos. Sin embargo, con el fin de cumplir todos los objetivos propuestos, decidimos aumentar la velocidad de aprendizaje e implementación. Es por ello que tomamos como referencia una plantilla que limpiamos y modificamos para que se ajustase a nuestros requisitos. Escogimos una basada en *Material UI*, la plantilla de Minimal Free [56], con licencia de uso MIT. Observamos que, gracias a ella, descubrimos una nueva librería para visualizar datos en forma de gráficas, *ApexCharts*, que resultó ser la que terminamos utilizando.

Creamos un mínimo producto viable que consistía en una página web *single page* con dos ventanas enrutadas dinámicamente. Una encargada de mostrar el dashboard y otra de mostrar información sobre la aplicación. Se implementaron los siguientes componentes para mostrar datos;

- 4 tarjetas para mostrar totales (cantidad de usuarios, tweets, likes y retweets).
- 1 wordcloud con las palabras más utilizadas.
- 2 tarjetas para mostrar rankings (por usuarios y hashtags).
- 1 visualizador de tweets para mostrar las denuncias.
- 1 gráfica temporal para representar las tendencias anuales en la cantidad de tweets.
- 1 línea de tiempo de doble eje para representar la progresión en la cantidad de tweets, likes y retweets a lo largo del tiempo.

Base de datos

La idea inicial era alimentar a la página web con los datos de IPFS. Sin embargo, los accesos para recoger datos de la blockchain pueden llegar a ser muy costosos. Además, en *IPFS* no pueden crearse índices para mejorar la eficiencia y tampoco cuenta con gran variedad de consultas. Es por todo ello que se tomó la decisión de crear una "memoria caché" que sirviese para alimentar a la página web. La idea es que esta BBDD fuese actualizada cada dos días con los datos de la blockchain. Decidimos utilizar *MongoDB Atlas*, para lo cuál resultaron de especial utilidad sus artículos y manuales [58, 57].

A partir de este punto, comenzamos a desarrollar tres proyectos en paralelo; uno encargado de crear la página web (*dashboard-twitter*), otro de gestionar las consultas con la BBDD (*cache-twitter*) y un tercero que se encargase de actualizarla cada dos días (*update-cache-twitter*).

Desarrollo en paralelo

Durante las sucesivas iteraciones de la fase de implementación, se fueron añadiendo cambios en los tres proyectos. Los mencionamos de forma superficial, porque se explicarán en detalle en la solución final (capítulo 8).

- **dashboard-twitter**: Se añadieron funcionalidades y cambios de estilo con el objetivo de cumplir todos los requisitos que se habían establecido. Los componentes básicos que tenía el producto mínimo viable se hicieron más complejos, aumentando las opciones que ofrecían (permitir descargas, hacer zoom en las gráficas, elegir qué datos ver...). Además, se añadieron nuevos componentes y funcionalidades (búsquedas, filtros, gráfica de correlación entre palabras...). Finalmente, tuvo lugar una fase en la que se hicieron los estilos más amigables y se añadieron detalles de diseño que mejoraban la experiencia de usuario (esqueleto de página web, pie de página...).

- **cache-twitter** y **BBDD**: Se aumentó la variedad de las consultas que se pueden realizar y se mejoró la eficiencia de la BBDD a la hora de resolverlas.

  Inicialmente, las consultas sólo podían filtrarse por fecha. A medida que aumentó la complejidad de los componentes, también lo hicieron las consultas. Permitimos filtrar por usuario, realizar búsquedas en el texto de las denuncias o devolver las consultas ordenadas. Para poder realizar estas operaciones de manera eficiente, recurrimos al uso de índices en la BBDD. A continuación, mostramos los que se crearon: *date*, *user*, *retweets* y *likes*.

- **update-cache-twitter**: Se creó un script que actualizase la BBDD de la forma más eficiente posible y creando un nivel de protección que evitase corromper la BBDD. En un primer lugar, también iba a encargarse de mantener actualizados los metadatos de los tweets (información sobre retweets, likes y respuestas) en esta parte de la aplicación. Sin embargo, decidimos en equipo que la actualización se hiciera desde la parte del backend de la aplicación, actualizando y guardando esta información también en IPFS.

Despliegue

Para que los tres proyectos pudiesen interactuar entre sí, y no estuviesen en local, se desplegaron en un servidor *cloud*. Decidimos utilizar *Heroku*, lo que facilitó el intercambio de información a través de llamadas *HTTP*.

Capítulo 8

Arquitectura de la solución

Utilizamos esta sección para explicar, en detalle, la arquitectura final de Injustweet, la aplicación que hemos creado. Podemos ver en la figura 8.1 el esquema general de la misma, dividido en los tres grandes bloques; recogida de datos, almacenaje de datos y visibilización de datos. A continuación, explicamos cada uno de ellos en detalle.

8.1. Bloque 1: Recoger datos

Comenzamos estudiando la parte de la arquitectura destinada a la recolección de datos (tweets). Es decir, el primer bloque que se observa en la figura 8.1. Lo primero que observamos es que se obtienen de dos fuentes distintas.

- **API Twitter Developer**: A través de su servicio *streaming*, los obtiene en tiempo real.
- **Scraper de redes sociales**: Permite acceder al histórico de tweets de Twitter.

8.1.1. Stream.py

Comenzamos viendo la arquitectura del programa **stream.py**, en la figura 8.2, que obtiene *tweets* en tiempo real. Como se puede ver en la figura 8.3, el código parte de un hilo principal desde el que se crea un hilo demonio (gracias al parámetro *threaded*) que ejecuta un *listener*. Para ello es necesario proporcionar los *tokens* de la cuenta de *Twitter Developer*. Este *listener* emplea el método *stream* proporcionado por la librería *tweepy*, que a su vez utiliza el protocolo *Streaming HTTP* para dar datos a través de una conexión abierto en streaming con la API de Twitter. Sobre este *listener*, se usa una función *filter*, permitiendo filtrar tanto el lenguaje del tweet, como las palabras clave que debe contener el texto (siendo éstas las que conforman la tercera iteración del diccionario).

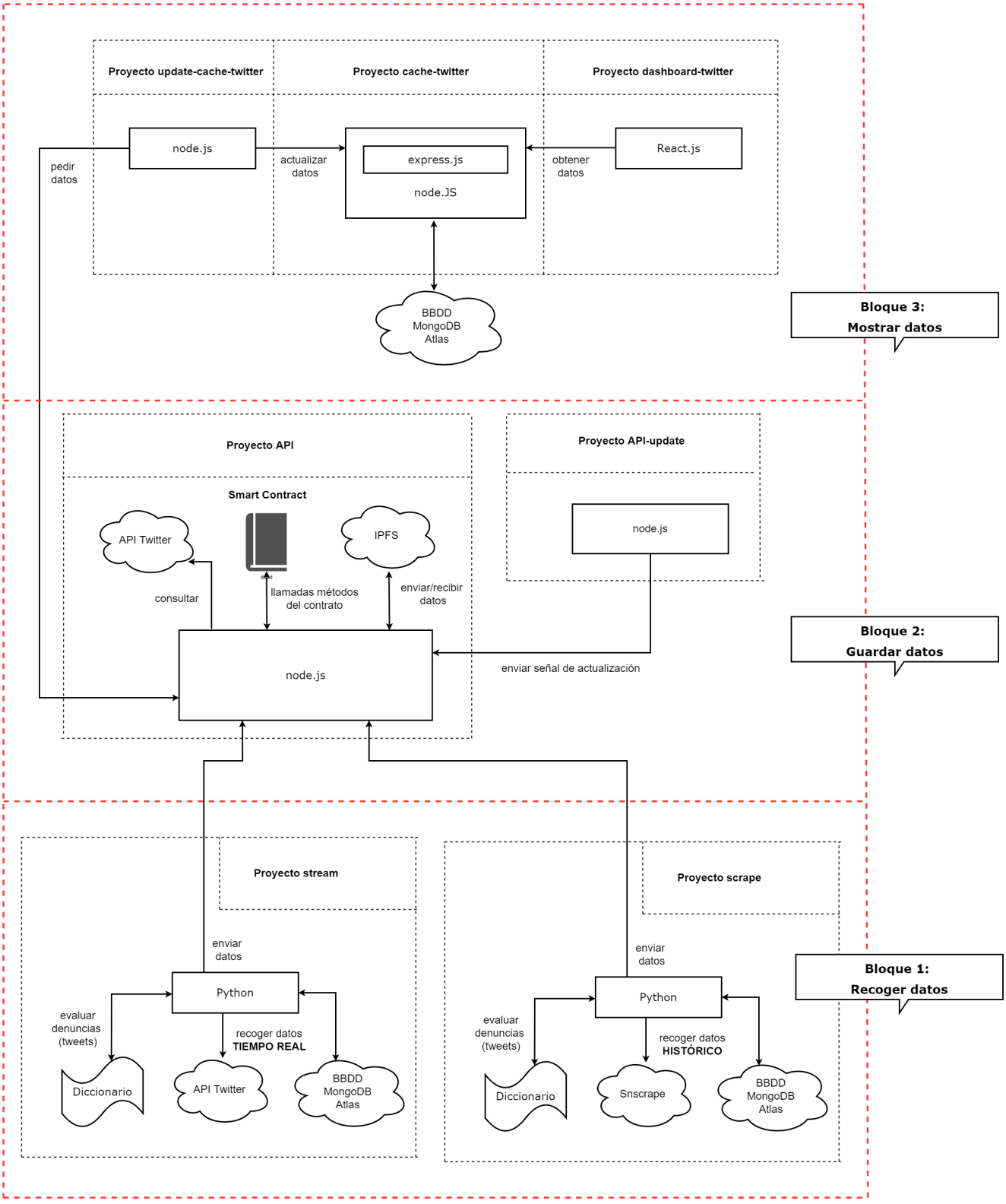


Figura 8.1: Arquitectura de la aplicación

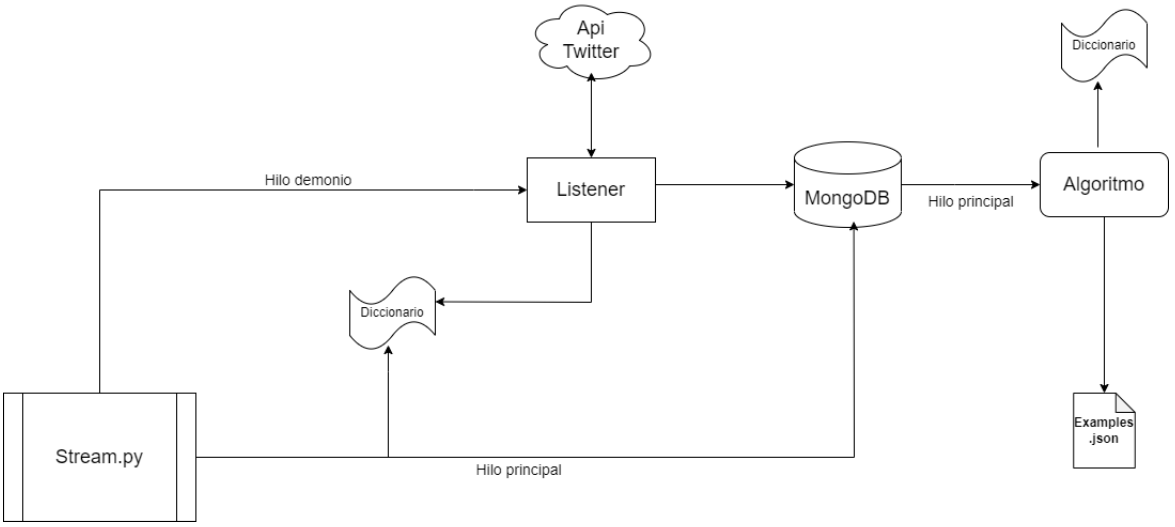


Figura 8.2: Obtención de tweets en tiempo real

```
query = pd.read_csv("../../../dict/query_dic.csv")
freq_dict = pd.read_csv("../../../dict/FREQUENCIES_DIC.csv")
load_dotenv(find_dotenv("env/TwitterTokens.env"))

tweepy_stream = SimpleListener(
                    os.getenv('API_KEY'),
                    os.getenv('API_KEY_SECRET'),
                    os.getenv('ACCESS_TOKEN'),
                    os.getenv('ACCESS_TOKEN_SECRET'), daemon=True)

tweepy_stream.filter(languages=['es'], threaded=True,
track=[
    query["WORD"][0], query["WORD"][1], query["WORD"][2],
    query["WORD"][3], query["WORD"][4],query["WORD"][5],
    query["WORD"][6],query["WORD"][7],query["WORD"][8],
    query["WORD"][9],query["WORD"][10],query["WORD"][11],
    query["WORD"][12],query["WORD"][13],query["WORD"][14],
    query["WORD"][15],query["WORD"][16],query["WORD"][17],
    query["WORD"][18],query["WORD"][19],query["WORD"][20],
    query["WORD"][21],query["WORD"][22],query["WORD"][23],
    query["WORD"][24]])
```

Figura 8.3: Creación del listener

Cuando el *listener* reciba un tweet la función *on_status* será invocada. En esta función recogemos los datos previamente mencionados en el formato del *JSON* para subirlos a la base de datos, tal y como se ve en la figura 8.4.

```
def on_status(self, status):
    post = {'link': link, 'id': tweet_id, 'text': text, 'user': user,
        'date': date, 'likes': n_likes, 'retweets': n_retweets, '
       replies': n_replies, 'hashtags': l_hashtags}
```

Figura 8.4: Recogida de datos para formar el JSON

Una vez recogidos los datos, se llama desde la función *on_status* a la función *insert_one*, que inserta en la colección *tweet_stream* de la base de datos *collected_tweets* un *documento*, como se ve en la figura 8.5. La dirección contenida en la creación de un objeto de la clase *MongoClient* es la dirección para conectarse al *cluster* de *MongoDB*, aunque algunos parámetros (las claves) han sido sustituidos por seguridad.

```
class SimpleListener(tweepy.Stream):
    client = MongoClient("mongodb+srv://XXX:XXXX@cluster0.nf86w.
       mongodb.net/Twitter-dbs?retryWrites=true&w=majority",
       tlsCAFile=certifi.where())
    db = client['collected_tweets']
    collection = db['tweet_stream']

    self.collection.insert_one(post)
```

Figura 8.5: Conexión del Listener con MongoDB

Mientras tanto, el hilo principal accede a la BBDD y consulta, de forma indefinida, todos los *documentos* que haya. Para cada uno, limpia el texto, eliminando *links*, emoticonos, caracteres especiales y normalizando el texto pasándolo a minúsculas. Para ello, usamos el método *clean_text* que se observa en la figura 8.6. Es necesario aclarar que éste no es el formato con el que se suben los tweets a *IPFS*, ya que hemos tratado de mantener el texto lo más fiel posible al original. Estas modificaciones surgen por el formato *JSON*, que no acepta ciertos caracteres como los saltos de línea o las comas dobles (estos cambios se pueden visualizar en la figura 8.7). Por ello, esta función solamente es necesaria a la hora de hacer el análisis del texto.

```
def clean_text(text):
    c_t = re.sub(emoji.get_emoji_regexp(), " ", text)
    c_t = re.sub("(@.+)|(#.+)   ", "", c_t)
    c_t = re.sub(r"https\S+", "", c_t)
    c_t = re.sub(r'[^\w]', ' ', c_t)
    c_t = c_t.lower()
    return " ".join(c_t.split())
```

Figura 8.6: Código para limpiar textos de tweets antes de su análisis

```
def clean_text_final_format(text):
    c_t = re.sub(r'\n', ' ', text)
    c_t = re.sub(r'"', '\\"', c_t)

    return " ".join(c_t.split())
```

Figura 8.7: Código para limpiar textos de tweets para su subida a IPFS

Una vez hecho esto, el algoritmo procede a evaluar si se trata de una denuncia o no. Para ello, se envía el texto al módulo precargado de *spacy*. En nuestro caso, llamado *nlp*, que nos devuelve un *Doc* procesado. A continuación, almacenamos en un array todos los *tokens*, quedándonos con aquellos que no sean ni *stopwords* ni signos de puntuación. Además, nos quedamos con aquellos que sólo tengan caracteres alfanuméricos y descartamos todos los *tokens* de menos de cuatro caracteres. Esto se debe a que, por lo que pudimos observar, a veces fallaba a la hora de identificar algunas palabras vacías de corta longitud. Observamos lo anterior en la figura 8.8 .

```
text = clean_text(post['text'])
obj = nlp(text)
tokens = [tk.orth_ for tk in obj if not tk.is_punct | tk.is_stop]
normalized = [tk.lower() for tk in tokens if len(tk) > 3 and tk.
    isalpha()]
```

Figura 8.8: Código para el procesamiento mediante lenguaje natural, parte 1

El siguiente paso, es almacenar en un *string* cada uno de los *tokens* separados por un espacio. Este *string* lo procesaremos con *stanza*, lo que nos da un *document* procesado. Almacenamos en un *array* cada uno de los lemas de las palabras. Se puede observar en la figura 8.9.

```
for n in normalized:
        stringed = stringed + n + " "

doc = nlp_s(stringed)

for sent in doc.sentences:
    for word in sent.words:
        lemmatized.append(word.lemma)
```

Figura 8.9: Código para el procesamiento mediante lenguaje natural, parte 2

Una vez tenemos la lista de lemas, evaluamos si se trata de una denuncia o no. Para ello,

iteramos la segunda versión del diccionario. Si hay una coincidencia de alguna palabra en el texto, aumentamos el valor. Consideramos también que, si una palabra del diccionario aparece más de una vez en el texto, sólo se contabilizará una vez. Una vez tenemos el número de coincidencias en el texto, consideramos que es una denuncia si el número de palabras del diccionario encontradas en el texto supone un porcentaje del mismo de más del *5,34 %* (decisión que se expondrá más adelante en el apartado de evaluación del capítulo 9), lo que sería un valor mayor o igual a *5 palabras*. Se puede ver la evaluación en la figura 8.10.

```
1 def is_a_complain(text, freq_dict):
2     value = 0
3     repeated_words = []

5     for i in range(len(freq_dict)):
6         if freq_dict["WORD"][i] in text and freq_dict["WORD"][i] not
              in repeated_words:
7             value += 1
8             repeated_words.append(freq_dict["WORD"][i])

10    return ((value / len(freq_dict)) >= 0.0534)
```

Figura 8.10: Código para la evaluación de identificar una denuncia laboral

En caso de que sí se haya clasificado como denuncia, se almacenarán los campos especificados anteriormente en formato *JSON*, y serán almacenados en un fichero dedicado a guardar las denuncias antes de ser enviadas. Véase en la figura 8.11. Sean denuncias o no, eliminamos el tweet de la BBDD.

A partir de un umbral determinado en 80 posibles denuncias laborales, recogidas en el documento *JSON*, es cuando llamamos al *API* desarrollado en lenguaje *JavaScript*. El umbral ha sido determinado teniendo en cuenta la tasa de actualización de la caché de la página web; sería contraproducente hacer subidas más rápido de lo que se recogen, puesto que las subidas a *Blockchain* suponen un coste que debe ser minimizado. La función *erase_lastjson* (figura 8.12) elimina todos los caracteres, comenzando desde el final del archivo, hasta el último *JSON* (desde el final del archivo éste se encuentra desde el carácter '}'). Esto nos permite asegurar que el formato del archivo sea el correcto, tanto en una ejecución sin errores (en la que es necesaria eliminar la última coma y los saltos de línea), como en una con errores (si los datos estuvieran a medio escribir también se eliminarían). Antes de subirlo, es necesario cerrar el *JSONArray* del archivo *.json* con el carácter ']' y cerrar el archivo, esto último se lleva a cabo para que el archivo se actualice con el último carácter ']' (ya que la llamada *close* llama a *flush*, que permite actualizar el archivo en disco vaciando los datos que hay en el buffer interno del archivo). Es importante recalcar que los datos no son escritos en el disco hasta que el archivo se cierra, cosa que ocurre justo antes de ser enviado y en caso de error, o cuando se rebasa el tamaño del buffer. Podrían realizarse llamadas a *flush* para escribir en disco cada vez que se recolecta una denuncia, pero esto retrasaría la ejecución del programa de forma innecesaria,

```
if(is_a_complain(lemmatized, freq_dict)):
    aux_json += "{\"link\":\"" + post['link'] + "\", \"id\":\"" + post
        ['id'] + "\", \"text\":\"" + clean_text_final_format(post["
        text"]) + "\", \"user\":\"" + post['user'] + "\", \"date\":"
                + str(int(post['date'].timestamp())) +", \"likes\":" +
                    str(post['likes']) + ", \"retweets\":" + str(post[
                    'retweets']) + ", \"replies\":" + str(post['replies
                    ']) + ", \"hashtags\":"
    aux_hashtags = "["
    for h in post['hashtags']:
        aux_hashtags+= ("\"" + h['text'] + "\", ")

    if (len(aux_hashtags) > 1):
        aux_hashtags = aux_hashtags[:-2]
    aux_hashtags += "]"

    aux_json += (aux_hashtags + "}, \n")
    f.write(aux_json)
    return True
return False
```

Figura 8.11: Código para almacenar las denuncias en formato JSON

ya que si hubiera un error el bloque *finally* se encargaría de cerrarlo.

Tras tener preparado el archivo se utiliza la clase *Popen* del módulo *subprocess*, con el cual se ejecuta un *proceso hijo* con el código especificado en **CodigoAPICliente.js** (figura 8.13) que permite hacer la subida a través de la API. Este código importa el archivo *.json* que almacena los tweets y lo carga en una variable. Después, se configuran los parámetros de la llamada a la API de forma que se haga un método *POST* cuyo contenido será un *JSON* y el archivo importado se mete el campo *body* de la petición. Por último se realiza la llamada al método *set* del API que se encarga de subir el contenido a *IPFS* usando la función *fetch*. Una vez creado el proceso en *Javascript*, con la función *wait* esperamos a que dicho proceso hijo termine su ejecución para después borrar el archivo subido y dejarlo como antes de que recogiera denuncias añadiendo el carácter '[' con el que comienza el *JSONArray*. Este truncado del archivo se realiza para evitar subir el mismo archivo repetido a *IPFS*.

En la figura 8.14 se muestra el código del hilo principal que se ha descrito en esta sección.

```
def erase_lastjson(f):
    n_c = 0
    f.seek(0, os.SEEK_END)
    file_size = f.tell()
    while (file_size - n_c) > 0:
        f.seek(file_size - n_c)
        aux = f.read(file_size - n_c)
        if aux != '':
                if aux[0] == '}':
                        break
        n_c += 1

    f.seek(-n_c+1, os.SEEK_END)
    f.truncate()
```

Figura 8.12: Código para manejo de errores, parte 1

```
import fetch from "node-fetch";
import { createRequire } from "module";
const require = createRequire(import.meta.url);
const postBody = require('../../../json/XXXXX');

const options = {
    method: 'POST',
    body: JSON.stringify(postBody),
    headers: {
        'Content-Type': 'application/json'
    }
}
await fetch('https://precariedappv2.herokuapp.com/set',options);
```

Figura 8.13: Código para la subida de archivos a IPFS

```
try:
    while (1):
        for post in collection.find():
            if(index > 80):
                erase_lastjson(f)
                f.write("]")
                f.close()
                p = subprocess.Popen(["node", "CodigoAPICliente.js"])
                p.wait()
                f = codecs.open("../../../json/examples.json", 'a+',
                    encoding='utf-8', errors='ignore')
                f.seek(0, os.SEEK_SET)
                f.truncate()
                index = 0
                f.write("[")

            if text_analysis(post, nlp, nlp_s, freq_dict, f):
                index+=1
            collection.delete_one({"_id": post['_id']})
```

Figura 8.14: Código fundamental del hilo principal

Con el objetivo de reducir los posibles errores de formato que puedan surgir al subir el archivo se han tomado varias medidas. Por una parte se emplea un bloque *finally* que captura cualquier error que pueda surgir al tratar de subir el archivo, determinar si un texto es una denuncia o eliminar de la base de datos. Este fragmento de código puede visualizarse en la figura 8.15. El comportamiento es bastante similar a una subida normal de los datos, a excepción de la subida de los datos. Esto se debe a que los fallos que pueden surgir durante la ejecución son los originados con el *API*, en ocasiones la sentencia *wait* genera excepciones de *Timeout*. Por lo tanto el propósito de esta sección del código es el de asegurar la conservación de los datos obtenidos hasta ese momento, con lo que, si se ejecutara el bloque *finally*, los datos se conservarían en el formato adecuado para subirlos manualmente a posteriori.

```
finally:
    erase_lastjson(f)
    f.write("]")
    f.close()
```

Figura 8.15: Código para manejo de errores, parte 2

8.1.2. Scrape.py

Por otro lado tenemos el **scrape**, cuyo funcionamiento es muy similar al *stream*, por lo que solamente resaltaremos la diferencias más sustanciales. La arquitectura es la misma que

el anterior código, figura 8.2, con la salvedad de utilizar un *scraper web* en vez de un *listener* y el API de Twitter.

La estructura del programa emplea igualmente 2 hilos, uno con el que obtiene los datos mediante *snscrape* y prácticamente igual al del *stream* que realiza el análisis y los sube a la *blockchain*. Tal y como se ve en la figura 8.16, se crea desde el hilo principal un nuevo hilo que ejecuta la función que se encargará de recoger los tweets con *snscrape*. El hilo principal es exactamente igual al descrito anteriormente, recoge textos de la base de datos, llama a las funciones para clasificar el texto y, a partir de un cierto número de denuncias, sube el archivo a la *blockchain*. También cuenta con las mismas medidas de seguridad del formato del archivo.

```
def main():
    new_thread1 = Thread(target=thread_function)
    new_thread1.start()
```

Figura 8.16: Creación del hilo secundario en Scrape

Centrándonos en la función que obtiene los tweets, representada en la figura 8.17, el *scraper* no necesita utilizar los *tokens* de la cuenta *developer* de Twitter, ya que es independiente al *API* de Twitter. En nuestro caso empleamos la versión de librería de *Python*, pero podríamos usar la versión *CLI* utilizando el módulo *os* de Python para llamar al comando desde *Python* como si fuera una terminal. La consulta es muy parecida a la usada en el código anterior, con la salvedad de poder especificar el rango de fechas de los tweets con los campos *since* y *until*. Más allá de estos sutiles pero importantes cambios, no se diferencia del anterior código.

```
def thread_function():
    for i, tweet in enumerate(sntwitter.TwitterSearchScraper(
        query["WORD"][0] + " OR " + query["WORD"][0] + " OR " + query
            ["WORD"][0] + " OR " + query["WORD"][3] + " OR " + query["
            WORD"][4] + " OR " + query["WORD"][5] + " OR " + query["
            WORD"][6] + " OR " + query["WORD"][7] + " OR " + query["
            WORD"][8] + " OR " + query["WORD"][9] + " OR " + query["
            WORD"][10] + " OR " + query["WORD"][11] + " OR " + query["
            WORD"][12] + " OR " + query["WORD"][13] + " OR " + query["
            WORD"][14] + " OR " + query["WORD"][15] + " OR " + query["
            WORD"][16] + " OR " + query["WORD"][17] + " OR " + query["
            WORD"][18] + " OR " + query["WORD"][19] + " OR " + query["
            WORD"][20] + " OR " + query["WORD"][21] + " OR " + query["
            WORD"][22] + " OR " + query["WORD"][23] + " OR " + query["
            WORD"][24] + " lang:es -is:retweet since:2022-05-01 until
            :2022-05-09" ).get_items()):
```

Figura 8.17: Obtención de datos con snscrape

8.2. Bloque 2: Guardar datos

Nos centramos ahora en la parte de la arquitectura destinada al almacenaje de datos. Es decir, el segundo bloque que se observa en la figura 8.1. El objetivo de este bloque es almacenar los datos obtenidos en un sistema descentralizado, para evitar que sean censurados. En particular, vamos a hacer uso de *IPFS*, al cuál accederemos gracias a un contrato en la blockchain.

La API, al desplegarse, configura con *Web3* la conexión con *Ethereum* y con la cuenta de criptomonedas, utilizando el módulo *truffle-hdwallet-provider*. Para ello se utilizan dos claves; la de la cuenta de la que se sacan los fondos para ejecutar los métodos del contrato y la del proveedor de *Infura* que nos conecta con la red de prueba de *Rinkeby*, en la que está desplegado el contrato. El manejo de estas claves se hace con *dotenv.config()* y el archivo de configuración *.env*.

Tras configurar estas conexiones con la blockchain, se configura el código de la API usando *express*. A continuación, se crea la conexión con *IPFS* usando como puerta de enlace la API de *Infura*. Finalmente, para terminar la configuración, se le proporciona a *Web3* tanto la dirección del contrato como el ABI, lo que devuelve una variable que permite interactuar con los métodos de éste.

Observamos que para modificar el estado de la blockchain se necesita dinero (*ether*). Por lo tanto, llamar a los métodos de un contrato inteligente puede ser caro. Aunque en nuestro caso trabajamos con la red de prueba de *Rinkeby*, el contrato está desarrollado y pensado de forma que tenga un consumo de gas reducido para que sea eficiente. A continuación, mostramos los métodos que proporciona la API, desarrollada en *express*, y su funcionamiento.

- **Recogida de datos**: Recibe los tweets obtenidos mediante los procesos de *scrape* y *stream* del Bloque 1. Hace uso del método *POST* de la API que se puede observar en la figura 8.18.

  El método recibe un archivo *JSON* con un array que contiene la información del tweet que se desea almacenar. Tras *parsear* la petición, por cada objeto se obtiene el identificador del tweet, que se almacena en un array. Se guarda el cuerpo del *post* en *IPFS* y obtenemos el hash asociado al fichero subido, para poder acceder a él más adelante. Este hash, junto con el array de identificadores, se manda al contrato usando la función *setFile()*. Se puede observar en la figura 8.19.

  La función crea una estructura con estos datos y la amacena en el array principal del contrato. También se recorre los tweets guardando los IDs en un mapa como clave y asignándoles como valor la posición en la que se encuentra el hash del fichero que contiene esos tweets del array principal. Por último, se cierra la conexión con el cliente. En la figura 8.20 podemos comprobar el funcionamiento de esta parte mediante un diagrama de secuencia.

  Para recibir en la API la información con los tweets, se hace uso de un código en *Javascript* que se encarga de almacenar en la variable *postBody* un JSONArray con los

```
app.post('/set', async (req,res) => {
  let idArray = [];
  req.body.forEach(obj => { // Collects the tweet ID's
    idArray.push(obj["id"].toString());
  });
  //Store the tweets in IPFS and gets the IPFS hash
  const added = await client.add(JSON.stringify(req.body));
  //Sends the file IPFS hash with the tweet ID's it contains to de
      smart contract
  await contract.methods.setFile(added.path, idArray).send({
    from:'0x0fe2273f4754f494d3fe0C6D8cB5aE83f2b64bF9'});
  res.json({}); // Close the connection
});
```

Figura 8.18: Código del método POST de la API

```
    function setFile(string memory hash, string[] memory tweetsID)
        external {
        TweetsFile memory tf = TweetsFile(hash, tweetsID);
        tweetsFiles.push(tf);
        for(uint i =0; i < tweetsID.length; i++){
            storedTweets[tweetsID[i]] = tweetsFiles.length - 1;
        }
    }
```

Figura 8.19: Función *setFile()* del smart contract

datos de los tweets, y con la función fetch se conecta a la API enviando el archivo en el campo "body" de la petición. Este código viene más detallado en la figura 8.13 en el apartado anterior, junto con su explicación.

- **Envío de datos**: Proporciona los tweets almacenados en *IPFS* al frontend de la aplicación. Hace uso de un método *GET* de la API.

  En este caso, primero se accede desde el código en *Javascript* a la función del contrato en *Solidity getFiles* que devuelve el array principal del contrato en la blockchain. Ver la figura 8.21. Esta función es de tipo *view* dado que no modifica el estado de la blockchain, por lo que no consume nada al ser llamada.

```
function getFiles() external view returns (TweetsFile[] memory){
    return tweetsFiles;
}
```

Figura 8.21: Función *getFiles()* del smart contract

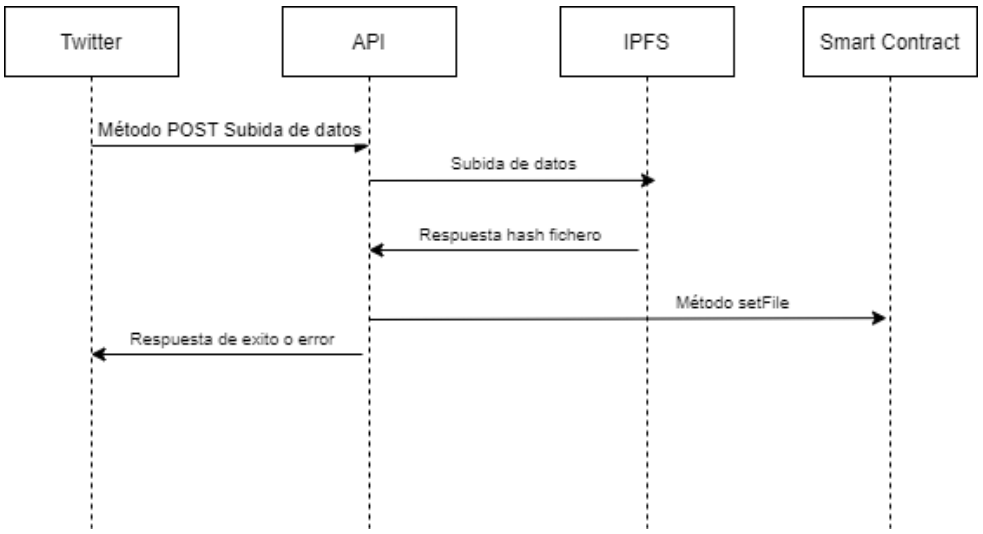


Figura 8.20: Diagrama de secuencia del método POST de la API

Usando el array con los hashes y los IDs asociados a cada hash, se recorren todos los elementos, descargando el contenido de *IPFS* con los hashes. La descarga se realiza por trozos o *chunks*, y una vez descargados se les da un formato *JSON*, el que tenían antes de ser subidos, y se recorren los tweets comprobando que su ID se encuentra en la lista de IDs asociados al fichero. En caso de no encontrarse, significa que ese tweet ha sido actualizado y que su información más reciente está en otro fichero, por lo que se elimina de la respuesta.

Cuando un fichero no contenga ningún tweet válido, se elimina su "hueco vacío"que deja en la respuesta y se pasa a comprobar el siguiente hash.

A continuación, se limpia la respuesta de corchetes, y se añaden comas entre los distintos JSONArray para almacenarlos de forma conjunta y continua y que la respuesta sea un único JSONArray con la información de cada tweet. Por último, se envía la respuesta como tipo *string*. En la figura 8.22 podemos comprobar el funcionamiento de esta parte mediante un diagrama de secuencia.

- **Actualización de datos**: Actualiza los metadatos de los tweets almacenados en *IPFS*. Hace uso de un método *GET* de la API. Para realizar la actualización automática de los datos, se llama a este método para indicar que ha de comenzar sin tener que enviar ningún otro dato.

  Al igual que en el envío de datos, primero se accede desde el código en *Javascript* a la función del contrato en *Solidity getFiles*. También se recorren todos los elementos descargando el contenido de *IPFS* y se le da formato al contenido descargado, pero en este caso, si se alcanza la cantidad de tweets que se pueden mirar, para de hacer comprobaciones. Una vez se ha recompuesto la información de IPFS, y se ha comprobado que el ID del tweet está en la lista de IDs asociados, se llama a la API de Twitter para *Javascript* a la que se le pasa el ID del tweet. Este método nos devuelve la información que tiene Twitter sobre ese tweet en concreto y comprobamos que las métricas que

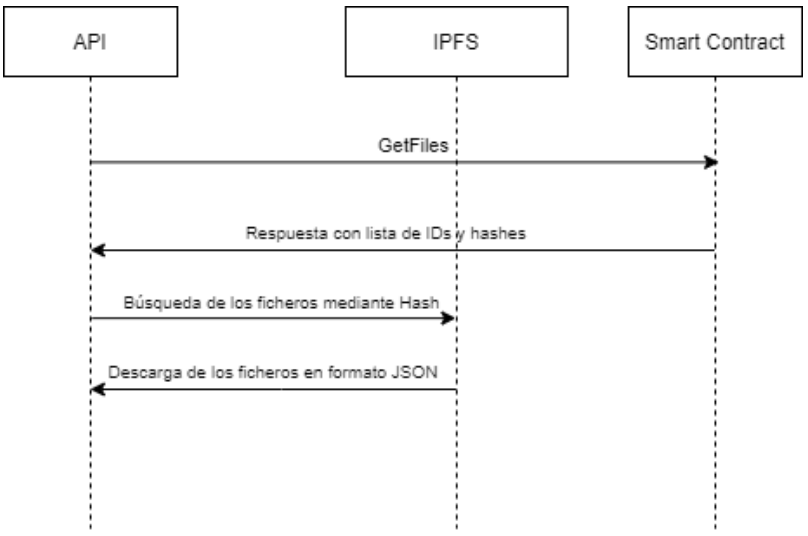


Figura 8.22: Diagrama de secuencia del método GET de la API

pueden haber variado con el tiempo y nos interesa mostrar en la web, es decir, los retweets, los likes y las respuestas, no sean diferentes a las que tenemos almacenadas. En caso de ser diferentes, éstas se modifican y se almacena la información del tweet en un array con todos los que se van a actualizar y el ID en otro array. Una vez se han probado todos los tweets, se sube a IPFS un nuevo fichero con todos los actualizados y el hash de este archivo se le pasa al contrato junto con la lista de IDs que hemos subido a IPFS llamando a la función *updateTweets*. Esta función se encarga de añadir el nuevo fichero al array principal como si de un archivo con tweets nuevos se tratase y modificar el mapa de clave ID, valor posición del array del hash del fichero en el que se almacena sustituyendo la posición por la del nuevo fichero, también elimina el ID de la lista de asociados a su hash anterior. Para ello recorre la lista de IDs asociados al fichero y comprobando que dos identificadores son iguales, al ser tipo *string* no se puede hacer la comparación con "==", sino que se debe usar *keccack* para calcular el hash y comprobar que los bytes sean iguales, una vez encontrado se quita del array usando la función *pop()*. En caso de que el array quede vacío se elimina, pero no el hash del fichero por motivos de inmutabilidad e integridad.

La actualización se realiza gracias a un código que se encarga de llamar periódicamente a este método. Este código que está aparte es un *cron* que utiliza los módulos **node-fetch** para enviar peticiones a la API y **node-schedule** para actuar de *cron*. También almacena el número de tweets que lleva actualizados y una variable booleana que se encarga de limitar la llamada a la actualización de los datos para que no actualizar más veces de las que se pillan los datos y evitar consumo de memoria del contrato innecesaria. Este código primero obtiene el número de tweets actualmente válidos que hay con una llamada al método *get* de la API. Con la longitud de la respuesta, sabe el número de tweets que tiene y lo almacena. Después hay dos funciones que se llaman de forma periódica. La primera, como se puede ver en la figura 8.24 es un trabajo que se llama cada 2 minutos, tiempo suficiente para que la API haya podido actualizar una tanda de

```
1     function updateTweets(string memory hash, string[] memory
          tweetsID) external {
2         for(uint j = 0; j < tweetsID.length; j++){
3             uint pos = storedTweets[tweetsID[j]];
4             for(uint i = 0; i < tweetsFiles[pos].tweetID.length; i++){
5                 if(keccak256(abi.encodePacked(tweetsFiles[pos].
                      tweetID[i])) == keccak256(abi.encodePacked(
                      tweetsID[j]))){
6                     tweetsFiles[pos].tweetID[i] = tweetsFiles[pos].
                          tweetID[tweetsFiles[pos].tweetID.length-1];
7                     tweetsFiles[pos].tweetID.pop();
8                 }
9             }
10            if(tweetsFiles[pos].tweetID.length == 0){
11                delete tweetsFiles[pos].tweetID;
12            }
13        }
14        TweetsFile memory tf = TweetsFile(hash, tweetsID);
15        tweetsFiles.push(tf);
16        for(uint i =0; i < tweetsID.length; i++){
17            storedTweets[tweetsID[i]] = tweetsFiles.length - 1;
18        }
19    }
```

Figura 8.23: Función *updateTweets* del smart contract

tweets. Esta función, si el booleano de control es verdadero, llama al método *update* de la API e incrementa la variable que almacena el número de tweets actualizados, si se han terminado de actualizar todos, pone el booleano a falso para que no se puedan seguir haciendo llamadas. La segunda función se ejecuta cada más tiempo, en este caso días. Se encarga de que una vez ha pasado suficiente tiempo, ponga la variable de control a verdadero para que se pueda volver a ejecutar la actualización de los datos y obtiene el número de tweets con otra llamada al *get* de la API, porque la cantidad de tweets a actualizar puede haber cambiado.

8.3. Bloque 3: Mostrar datos

Finalmente, estudiamos la parte de la arquitectura destinada a mostrar y representar datos. Es decir, el tercer bloque que se observa en la figura 8.1. Como hemos mencionado con anterioridad, está constituido por tres proyectos que se ejecutan de forma independiente.

- Proyecto **cache-twitter**: desarrollado en *node.js*, y haciendo uso de *express*, es el servidor que se encarga de recibir y gestionar peticiones sobre la base de datos de *MongoDB*. Esta base de datos actúa como una "memoria caché". Como mencionamos en desarrollo, se tomó esta decisión porque los accesos a *IPFS* son costosos en tiempo. Por lo tanto,

```
schedule.scheduleJob('*/2 * * * *', async function(){
    if(ok){
      await fetch('https://precariedappv2.herokuapp.com/update', {
          method:"GET"});
      numUpdated+=20;
      if(numUpdated>=numTweets){
        ok=false;
        numUpdated=0;
      }
    }
});
```

Figura 8.24: Código de ayuda para actualizar la API

reducimos la cantidad de ellos a cada dos días. Esos accesos sirven para actualizar la BBDD que se encarga de proporcionar datos a la aplicación web.

- Proyecto **dashboard-twitter**: desarrollado en *React*, representa el frontend de la aplicación. Muestra los datos recogidos en la BBDD que hemos construido. Observamos que los datos mostrados no están actualizados en tiempo real.
- Proyecto **update-cache-twitter**: desarrollado en *node.js*, recoge datos de *IPFS* cada dos días para actualizar la "memoria caché".

Observamos que se ha seguido la arquitectura full-stack conocida como *MERN* (*MongoDB*, *express.js*, *React.js* y *Node.js*). Estudiemos en detalle el funcionamiento de cada uno por separado.

8.3.1. cache-twitter

Creamos un servidor que se queda a la escucha de peticiones realizadas tanto por la página web (dashboard-twitter), como por el script encargado de actualizar la "memoria caché" (update-cache-twitter). Una vez aceptada la petición *HTTP*, el servidor utiliza un cliente de la base de datos que hemos creado para conectarse con ella y realizar la acción que se haya solicitado. Contamos con un fichero *.env* que hace de capa de seguridad al acceder a la BBDD de *MongoDB Atlas*. Podemos ver en el diagrama de secuencia 8.25 las interacciones que se llevan a cabo.

Observamos el tipo de peticiones que realizan la página web y el script a la BBDD:

- **Página web**: Únicamente hace las consultas necesarias para mostrar datos en el dashboard, no puede modificar el contenido de la BBDD. Estas consultas se filtran y ordenan según los criterios que estipule la página web, para lo cuál se ha heho uso de las funciones *db.collection.find()* y *db.collection.sort()*. Observamos que, con el fin de aumentar la eficiencia de la BBDD para gestionar estas peticiones, se han implementado índices

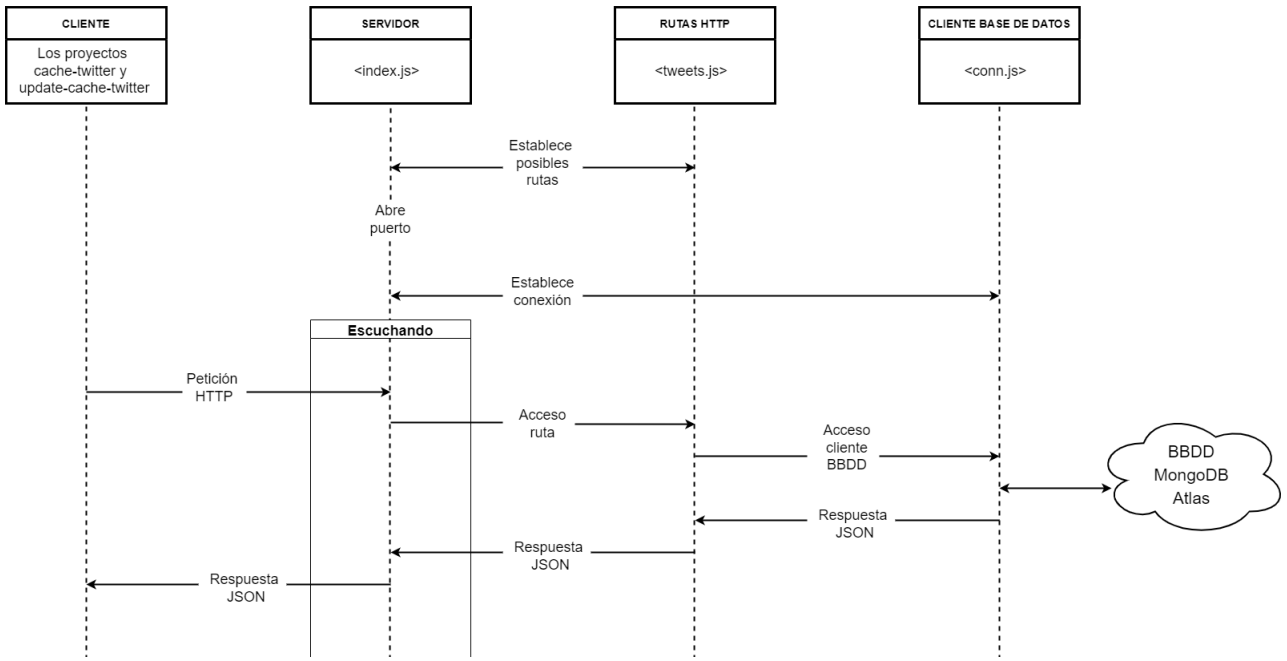


Figura 8.25: Diagrama de secuencia de cache-twitter

en los campos *date*, *user*, *retweets* y *likes*.

Por un lado, existe una consulta que obtiene todos los datos filtrados por una fecha de inicio y final (figura 8.26). Por otro lado, existe otra consulta que además de lo anterior, devuelve los tweets ordenados de 4 posibles maneras (fecha ascendente/descendente, cantidad de likes y cantidad de retweets) y tiene la opción de filtrar o no los tweets de acuerdo al nombre de usuario o por palabras en el texto del tweet (figura 8.27).

```
router.route("/").get(function (req, res) {
  let db_connect = dbo.getDb("twitter");
  let date_start = parseInt(req.query.dateStart);
  let date_end = parseInt(req.query.dateEnd);
  db_connect
    .collection("tweets_ipfs")
    .find({ date: { $gt: date_start, $lt: date_end } })
    .toArray(function (err, result) {
      if (err) throw err;
      res.json(result);
    });
});
```

Figura 8.26: Llamada realizada por la página web: datos filtrados por fecha

```
router.route("/words").get(function (req, res) {
  let db_connect = dbo.getDb("twitter");

  let word = req.query.word;
  let date_start = parseInt(req.query.dateStart);
  let date_end = parseInt(req.query.dateEnd);
  let order = parseInt(req.query.order);

  let textuser = word[0] == "@" ? "user" : "text"
  let nameVariable = (order == 0 || order == 1) ? "date" : ((order ==
     2) ? "retweets" : "likes");
  let ascdesc = (order == 1) ? 1 : -1;
  value = word[0] == "@" ? word.slice(1) : { $regex: word };

  var query = {}
  query['date'] = { $gt: date_start, $lt: date_end };
  query[textuser] = value;

  var query_sort = {};
  query_sort[nameVariable] = ascdesc;

  db_connect
    .collection("tweets_ipfs")
    .find(query)
    .sort(query_sort)
    .toArray(function (err, result) {
      if (err) throw err;
      res.json(result);
    });
});
```

Figura 8.27: Llamada realizada por la página web: datos filtrados por fecha y/o nombre de usuario y/o palabra en el texto, y ordenados (fecha ascendente/descendente o cantidad de likes/retweets)

- **Script**: Encargado de gestionar la BBDD, realiza operaciones para eliminar todos los elementos de la BBDD y añadir elementos dado un array (figura 8.28). Para ello, se ha hecho uso de las funciones *db.collection.deleteMany()* y *db.collection.insertMany()*.

```
// To add a list of tweets to the data base
router.route("/add").post(function (req, response) {
  let db_connect = dbo.getDb("twitter");
  db_connect.collection("tweets_ipfs").insertMany(req.body, function
     (err, res) {
    if (err) throw err;
    console.log("Tweets added successfully");
    response.json(res);
  });
});

// To delete all tweets from the data base
router.route("/delete").delete((req, response) => {
  let db_connect = dbo.getDb("twitter");
  db_connect.collection("tweets_ipfs").deleteMany({}, function (err,
     obj) {
    if (err) throw err;
    console.log("All document deleted");
    response.json(obj);
  });
});
```

Figura 8.28: Llamadas realizadas por el script

8.3.2. update-cache-twitter

Está constituido por un script que, de forma automática, accede cada dos días a los datos almacenados en *IPFS*. Para ello, utiliza el método *get* proporcionado por la API del Bloque 2. Posteriormente, elimina el contenido de la BBDD y vuelca los nuevos datos actualizados sobre ella, utilizando las funciones *delete* y *add* explicadas en el proyecto anterior. Mostramos en la figura 8.29 dichas llamadas.

Observamos que se ha hecho uso del paquete *node-schedule* de *npm* para implementar la periodicidad del script (una vez cada dos días) y de *node-fetch* para gestionar las llamadas *HTTP*. Además, destacamos que se ha implementado una estructura de control de errores de forma que si no se obtienen bien los datos de *IPFS*, la BBDD permanece inalterada. De la misma manera, no se añadirán tweets si la eliminación previa ha fallado.

```
await fetch('https://precariedapp.herokuapp.com/get', {
    method: "GET" });
await fetch("https://cache-twitter.herokuapp.com/delete", {
    method: "DELETE"
    });
await fetch("https://cache-twitter.herokuapp.com/add", {
    method: "POST",
    headers: { "Content-Type": "application/json",},
    body: res,
    });
```

Figura 8.29: Llamadas realizadas para recoger datos de la blockchain y eliminar/añadir datos a la "memoria caché"

Para justificar la implementación que hemos llevado a cabo, observamos que en la base de datos de *IPFS* puede haber tres tipos de datos a la hora de actualizar la "memoria caché":

1. Datos que sí es estaban en la "memoria caché" y no han sido actualizados en *IPFS*.
2. Datos que sí estaban en la "memoria caché" y sí han sido actualizados en *IPFS*. Este caso se da cuando un tweet que existía en la "memoria caché" tiene actualizado algún campo de metadatos en *IPFS*. Por ejemplo, el número de retweets o likes asociado a él. Esto se debe a que los datos se toman de Twitter en tiempo real y los campos se van actualizando a medida que pasa el tiempo.
3. Datos nuevos que no estaban en la "memoria caché".

Nos planteamos si sería más eficiente actualizar los datos existentes en la "memoria caché" y añadir aquellos que sean nuevos, en vez de vaciar y volcar todo el contenido. Sin embargo, descartamos esa opción por los siguientes motivos:

- Las dos bases de datos cuentan con la misma estructura, y no añadimos ningún dato nuevo, por lo que no es necesario recorrerlas elemento a elemento cuando vaciamos y volcamos los datos en la "memoria caché".
- No podemos conservar los datos que ya están en la "memoria caché" y añadir los nuevos debido al tipo de datos número 2. Esto nos forzaría a recorrer y comprobar todos los elementos cada vez.
- Al realizar la operación cada dos días, no es un proceso que afecte negativamente al rendimiento de la aplicación.

8.3.3. dashboard-twitter

Implementa el frontend de la aplicación representado, por componentes, en la figura 8.30. Los próximos apartados los dedicamos a exponer la estructura general de la aplicación, los componentes del dashboard y los estilos utilizados en su diseño.

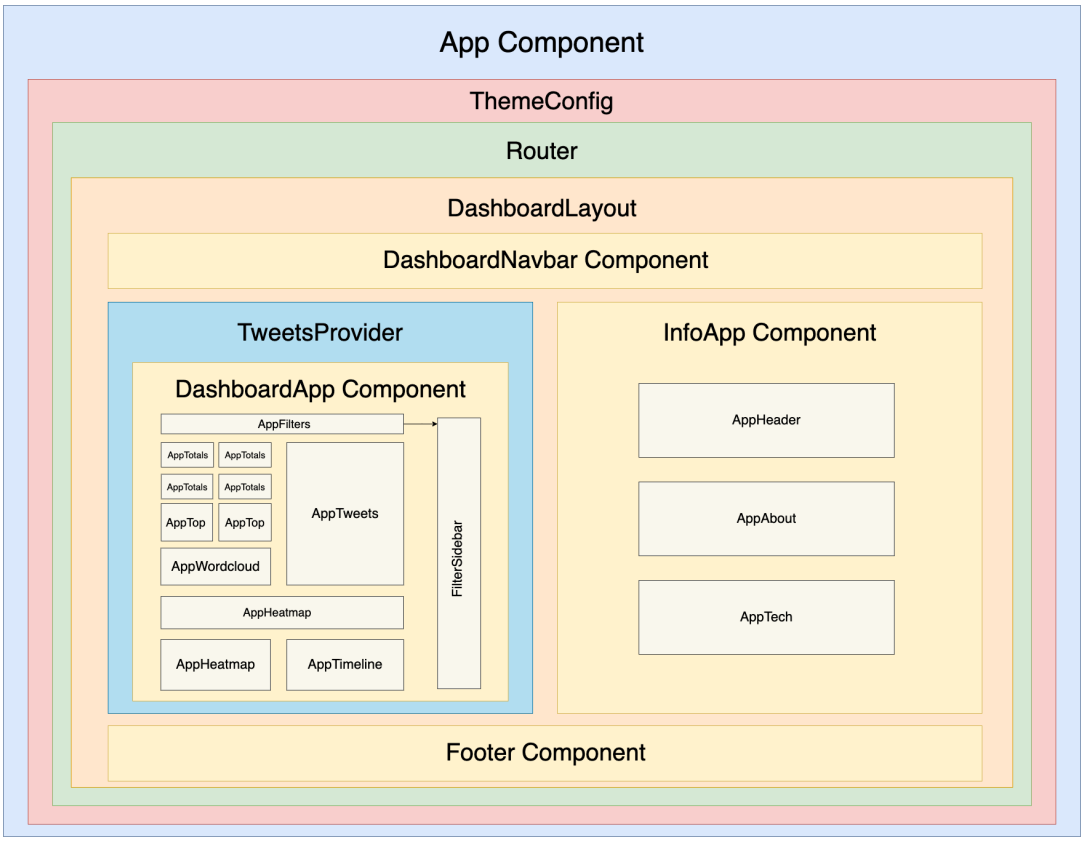


Figura 8.30: Diagrama de la estructura de dashboard-twitter

8.3.3.1. Estructura general de la aplicación

Para construir nuestra aplicación en *React*, definimos:

- *index.js*: Encargado de invocar la función *ReactDom.render* que renderiza nuestra aplicación en el elemento raíz del *DOM*. Utilizamos *BrowserRouter* para mantener sincronizadas la UI con la URL, y *HelmetProvider*, para establecer meta etiquetas y mejorar el posicionamiento en buscadores web.

- *app.js*: Representa nuestra aplicación y los estilos, de los cuáles hablaremos más adelante, que se aplican sobre ella. Contiene como componente principal el *Router*.

- *routes.js*: Encargado del enrutamiento dinámico de la página. Su elemento principal es *DashboardLayout*, que establece la estructura general de la página. Como vemos en el esquema, éste cuenta con dos hijos; *DashboardApp*, afectado por el contexto *TweetsProvider*, e *InfoApp*.

  - *InfoApp*: Cuenta con varios componentes hijos *AppHeader*, *AppTech* y *AppAbout* encargados de mostrar información sobre el proyecto y sus miembros.
  - *TweetsProvider*: *TweetsProvider*, utilizando el hook *useContext*, contiene el conjunto de funciones que calculan los valores que se mostrarán en el dashboard. Además, haciendo uso del hook personalizado *useFetch*, accede a la memoria caché

para cargar datos y actualizar los componentes. Este hook también nos proporciona un estado (*loading*) que indica si los datos se han terminado de cargar o no, que utilizamos para mostrar un esqueleto de la página web mientras no estén disponibles, haciendo uso de *Skeleton* de *Material UI*. Los otros estados de *TweetsProvider* son: *dateStart*, *dateEnd* y *data*. Indican el rango de fechas del que se obtienen datos y el conjunto de tweets, respectivamente. Todos ellos se pueden ver en la figura 8.31.

```
1 const [dateStart, setDateStart] = useState('0');
2 const [dateEnd, setDateEnd] = useState((new Date().setHours(0, 0, 0,
    0) / 1000).toString());
3 const {loading, data} = useFetch(`https://cache-twitter.herokuapp.com
    /?dateStart=${dateStart}&dateEnd=${dateEnd}`);
```

Figura 8.31: Estados de *context.js*

– *DashboardApp*: El componente *DashboardApp* define la cuadrícula y distribución de componentes en el dashboard. Los estudiamos en detalle a continuación.

8.3.3.2. Componentes del dashboard

De entre todos los componentes funcionales que posee la aplicación, dedicamos esta sección a exponer aquellos destinados a mostrar datos en el dashboard. Además, para cada uno de ellos, indicamos la función de *context.js* que le proporciona, o a la que envía, datos.

- **FilterSidebar/AppFilters**: Figura 8.32.

  El componente *AppFilters* contiene, tanto el título del dashboard, como el componente *FilterSidebar*. El objetivo de este último es dar la posibilidad al usuario de filtrar, por fecha, los datos que quiere visualizar en el dashboard. Este componente tiene particular interés porque en él vamos a hacer uso del hook *useContext* para enviar información hacia arriba en la jerarquía de componentes. Es decir, a los componentes padres en vez de a los hijos. Esta información será utilizada, posteriormente, para actualizar el resto de componentes y garantizar que todos ellos se ven afectados por los filtros.

  El componente envía datos a la función *filterTime*, que tiene dos parámetros de entrada: *start* y *end*. Indican el rango de fechas en el que se muestran los datos. Por defecto, *start* tiene valor 0, indicando el inicio de los tiempos, y *end* es el día actual. Esta función modifica los estados *dateStart* y *dateEnd*, ambos medidos en segundos desde el Epoch.

- **AppTotal (Tweets, Users, RT, FAV)** : Figura 8.33.

  Estos cuatro componentes representan, respectivamente, el total de tweets (denuncias), usuarios, retweets y likes recogidos en el rango de fechas establecido. Obtienen los datos

Figura 8.32: Componente FilterSidebar

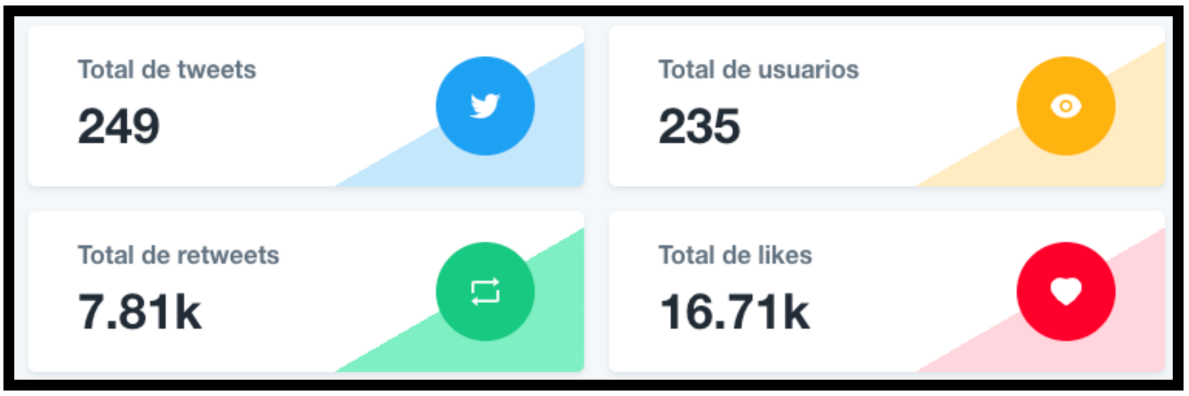


Figura 8.33: Componente AppTotals

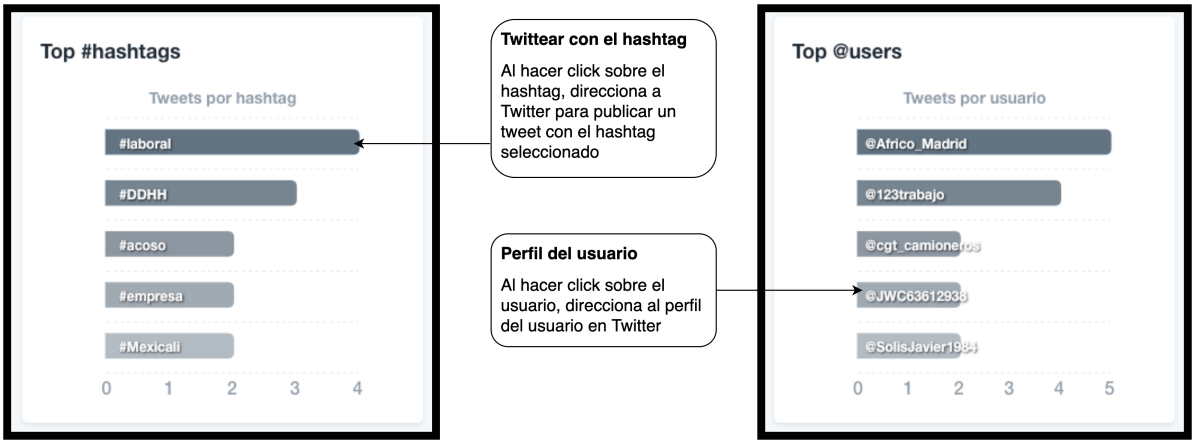


Figura 8.34: Componentes AppTopHashtag y AppTopUsers

gracias a la función *getTotals*, que toma los totales utilizando el estado *tweets*.

- **AppTopHashtags/AppTopUsers**: Figura 8.34.

  Estos dos diagramas de barras horizontales representan, en orden, los diez hashtags más utilizados y los diez usuarios que más tweets de denuncias laborales han publicado. Si se clickea sobre un hashtag, el dashboard te redirecciona a Twitter para escribir un tweet utilizándolo. En caso de clickear sobre un usuario, te redirecciona al perfil del mismo. Para ello, se ha hecho uso de la API *Web Intents* de Twitter.

  Los componentes obtienen los datos gracias a las funciones *getTopUsers* y *getTopHashtags*, que buscan los máximos utilizando el estado *tweets*.

- **AppTweets**: Figura 8.36.

  El componente *AppTweets* te permite visualizar los tweets que se han realizado de denuncias laborales. Incluye el nombre de usuario, el contenido de la denuncia y la reacción

que ha generado (cantidad de retweets, likes y respuestas).

Observamos que cuenta con un menú desplegable que el usuario puede usar para seleccionar en qué orden quiere mostrarlos (fecha creciente/decreciente u ordenados por cantidad de retweets/likes). Además, incluye una barra de búsqueda que permite buscar denuncias de usuarios concretos o de una temática en particular. Esto último lo hemos implementado creando el *Hook* personalizado *useFetch*, que permite realizar una búsqueda y filtrado sobre la BBDD. Los filtros de búsqueda pueden eliminarse. Además, se muestra el número de resultados obtenidos.

Los tweets no se muestran directamente de Twitter, sino usando el texto que se recogió en su momento. Esta decisión se tomó con el fin de evitar la censura. Es decir, si el tweet original se elimina, podremos seguir mostrándolo.

Si clickeamos sobre el nombre del usuario, gracias a la API *Web Intents* de Twitter, podemos acceder a él, en caso de que siga existiendo. Además, observamos que gracias a esa API también damos la posibilidad de tweetear una denuncia desde nuestra web utilizando el hashtag `#Injustweet`. Esto permite, no solo fomentar nuestra iniciativa, sino crear un sentimiento de comunidad entre las personas que se involucran en ella.

El componente obtiene los datos realizando la consulta a la BBDD de la figura 8.35. La llamada indica los filtros de fecha que deben aplicarse, si hay que filtrar por usuario o palabras en el tweet, y el orden que deben tener los datos devueltos.

```
1 useFetch(`https://cache-twitter.herokuapp.com/words/?dateStart=${
      dateStart}&dateEnd=${dateEnd}&word=${search}&order=${selected}`);
```

Figura 8.35: Uso del Hook useFetch en el componente AppTweets.

- **Wordcloud**: Figura 8.37.

  Representa un wordcloud con las palabras más utilizadas en los tweets de denuncias, para lo cuál se ha hecho uso del paquete *react-wordcloud* [81] de *npm*. Observamos que se ha añadido una opción de descarga utilizando el *Hook useRef*. Cuanto más grande la palabra, mayor presencia tiene.

  Para poder crearlo, se ha llevado a cabo un procesamiento del texto de los tweets. En primer lugar, se divide el texto en palabras y se convierten a minúscula. A continuación, se eliminan aquellas consideradas "vacías" (artículos, pronombres, preposiciones...), así como los hashtags. Finalmente, usando expresiones regulares, se eliminan caracteres no

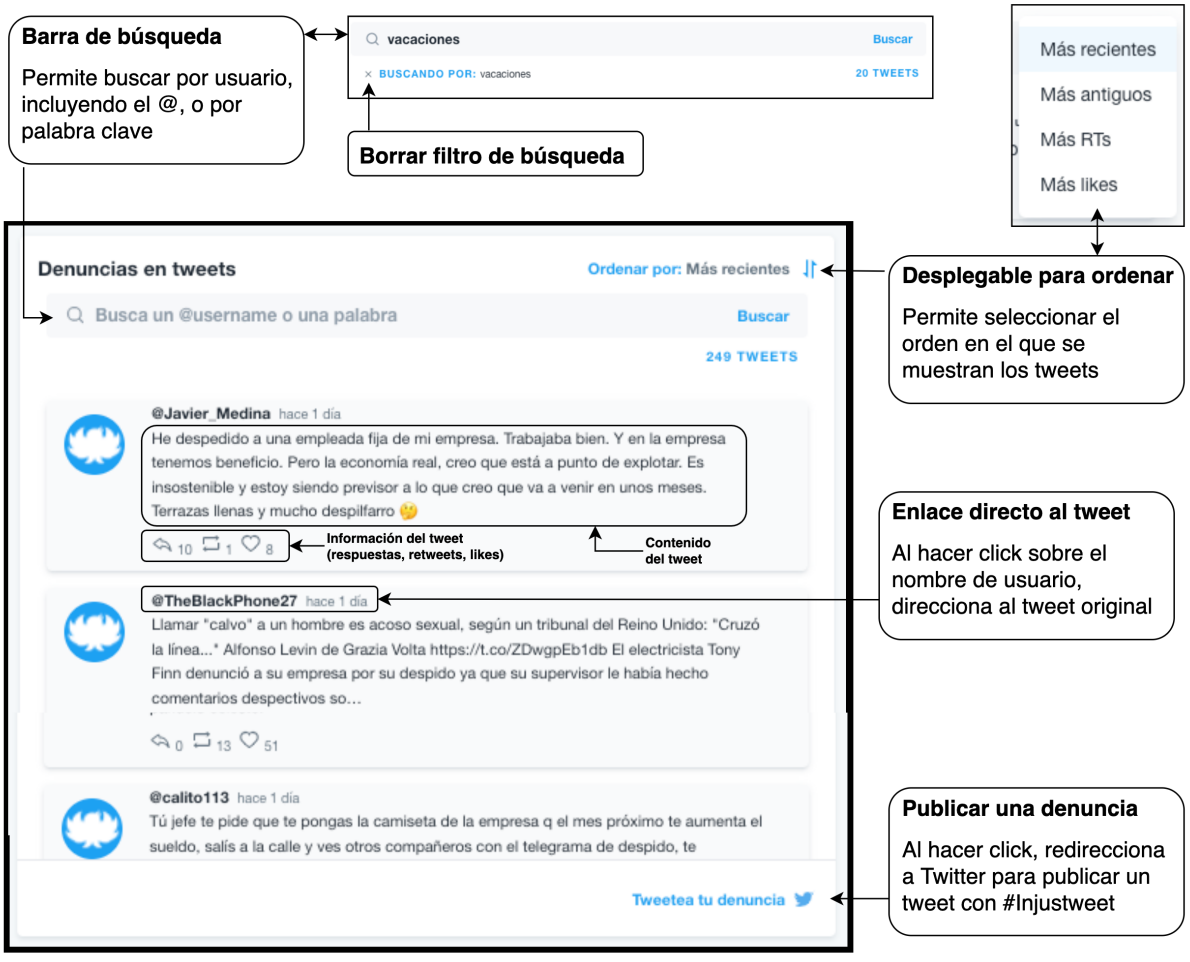


Figura 8.36: Componente AppTweets

Figura 8.37: Componente AppWordcloud

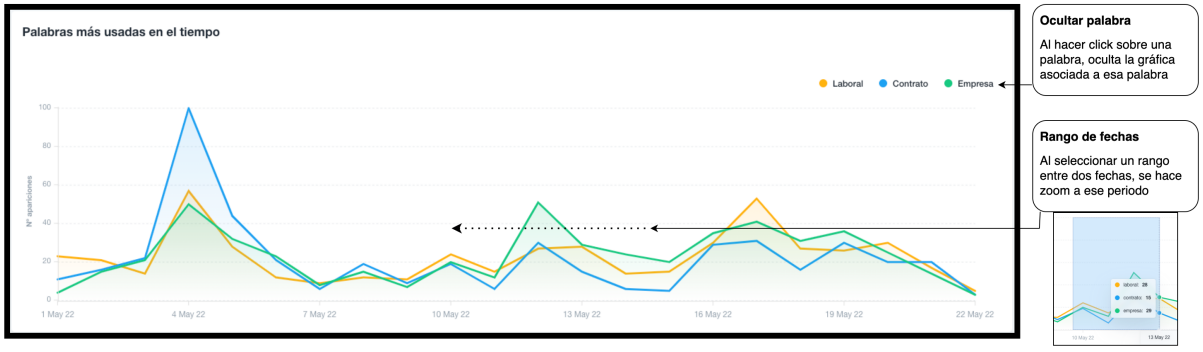


Figura 8.38: Componente AppWordsTime

alfanuméricos.

El componente obtiene los datos invocando la función *getDataWordcloud*, que devuelve las palabras junto a su cantidad de apariciones.

- **AppWordsTime**: Figura 8.38.

  Muy relacionado con el componente anterior, *AppWordsTime* toma las tres palabras más utilizadas en denuncias a día de hoy y muestra cuál ha sido su progresión y correlación a lo largo del tiempo. Para ello, calcula, utilizando expresiones regulares, el número de apariciones de dicha palabra, o derivadas de ella, en los tweets que se publicaron cada día.

  La gráfica permite elegir qué palabras visualizar, así como hacer zoom sobre un periodo de tiempo en particular.

  El componente obtiene los datos invocando la función *getDataWordsTime*, que proporciona las palabras más utilizadas y sus apariciones a lo largo del tiempo.

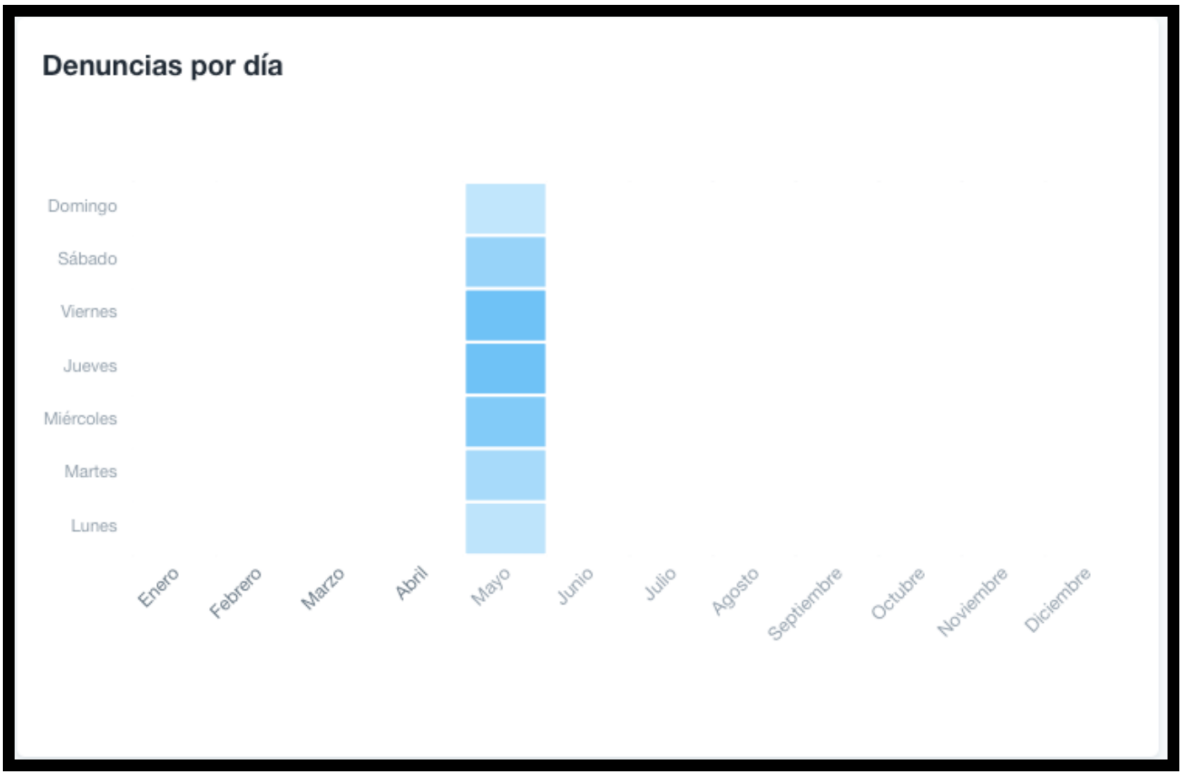


Figura 8.39: Componente AppHeatmap

- **AppHeatmap**: Figura 8.39.

  AppHeatmap desglosa la cantidad de tweets por día de la semana y mes del año, para dar una visión de cuáles son los momentos del año en los que más denuncias se han publicado. De esta forma, se pueden analizar tendencias anuales en los problemas laborales existentes.

  El componente obtiene los datos invocando la función *getDataHeatmap*, que proporciona la cantidad de tweets por día y mes.

- **AppTimeline**: Figura 8.40.

  Este componente contiene una gráfica en la que podemos visualizar la variación, a lo largo del tiempo, en la cantidad de tweets, likes y retweets. Se tomó la decisión de usar un doble eje vertical porque estas variables utilizaban escalas muy distintas. Incluimos la posibilidad de elegir cuáles de estas variables queremos visualizar y poder hacer zoom sobre un periodo de tiempo en particular.

  El componente obtiene los datos invocando la función *getDataTimeline*, que devuelve la cantidad de tweets, likes y retweets a lo largo del tiempo.

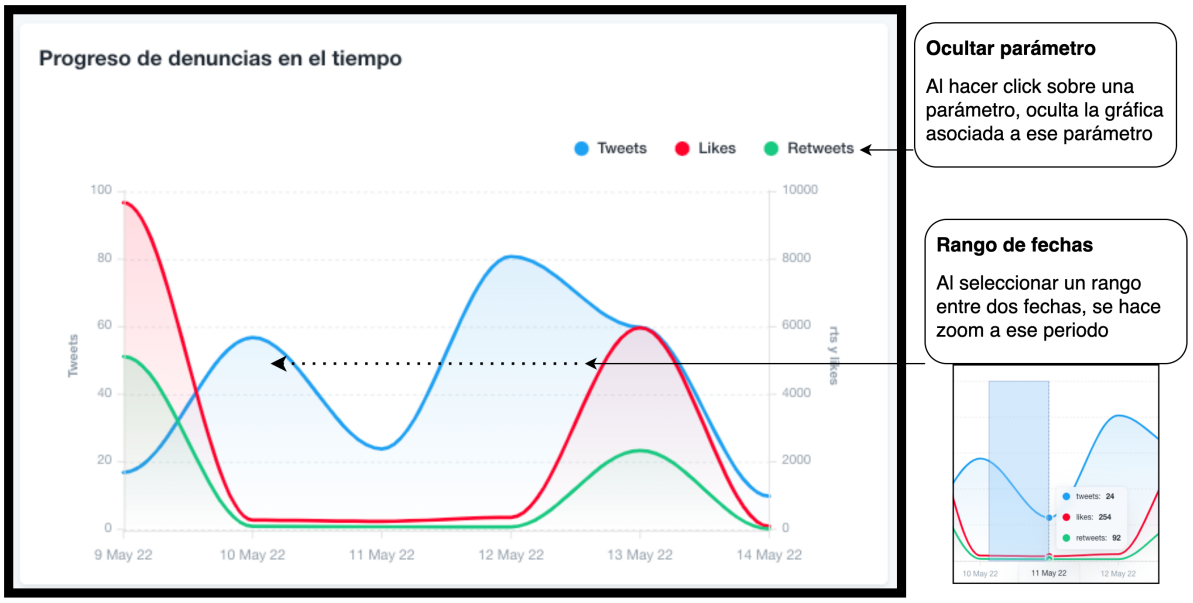


Figura 8.40: Componente AppTimeline

8.3.3.3. Estilos

A la hora de estilizar nuestra aplicación, hacer uso de la plantilla de *Minimal* nos aportó algunas facilidades para personalizar el proyecto a nuestro gusto. Ésta nos proporcionaba una estructura sobre la que trabajar y escoger los temas que queríamos usar, basada en la customización que proporciona *Material UI*.

La configuración de estilos que proporciona *Material UI* hace uso del componente *ThemeProvider*. Se basa en el uso del hook *useContext* para pasar la información relacionada con estilos a través del árbol de componentes, sin tener que enviar las propiedades manualmente en cada nivel. En nuestro caso, dentro de la carpeta *themes*, encontramos el fichero *index.js* que define el contexto mediante el componente *ThemeConfig*. Además, hace el uso del hook *useMemo* para solo calcular el valor memorizado que contiene los estilos cuando se renderiza, evitando así cálculos de más en cada renderización y optimizando la eficiencia de la aplicación.

La principal forma en la que hemos llevado a cabo la modificación de temas ha sido utilizando las variables que define *Material UI*, como *palette*, *typography* o *components*. Permiten, respectivamente, definir una paleta de colores, la tipografía utilizada y sobreescribir los componentes por defecto. Comentamos, a continuación, algunas de las modificaciones llevadas a cabo.

En referencia a la paleta de colores usada, seguimos con la línea de *Material UI* que propone escoger tonalidades distintas para diferentes situaciones:

- Color primario (*primary*): es el color más frecuente en la aplicación y se encuentra en los elementos principales de la misma.

- Color secundario (*secondary*): sirve para acentuar y diferenciar algunos elementos de la aplicación.
- Color de error (*error*): presente en los elementos que el usuario debe conocer.
- Color de aviso (*warning*): usado para representar avisos importantes.
- Color de información (*info*): se utiliza para mostrar información neutral.
- Color de éxito (*success*): indica acciones completadas de forma correcta.

Observamos que estos colores también se emplean en el uso de visualización de datos a la hora de dibujar las gráficas, manteniendo así la concordancia de colores en toda la aplicación.

Como nuestro proyecto está relacionado con Twitter, decidimos emplear una paleta de colores relacionada con la red social. Esta paleta se muestra en la Figura 8.41. Vemos que utiliza como color primario y de información el azul de "Larry", el icónico pájaro del logo de Twitter. Como color secundario y de error utiliza el rojo asociado al *like* de un tweet. El verde del *retweet* se utiliza como color de éxito. Y por último, el amarillo del antiguo botón de *fav* (sustituido por el *like*) se utiliza como color de aviso.

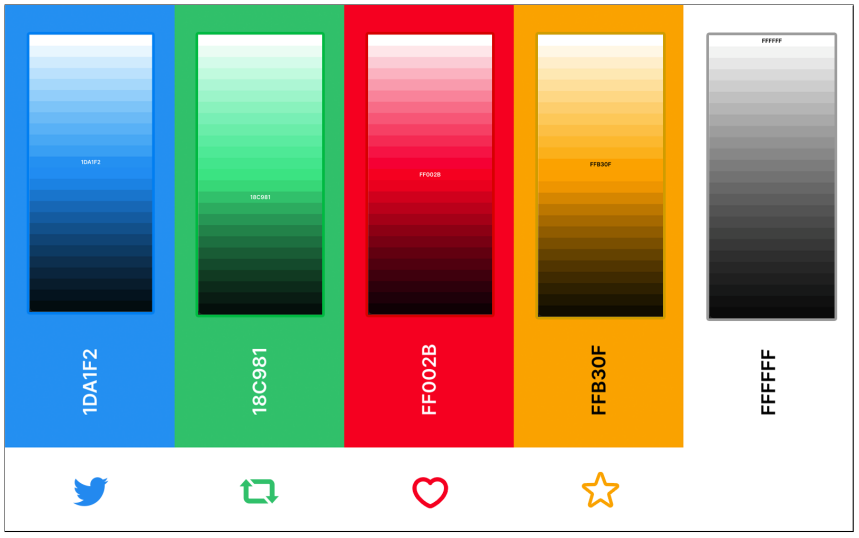


Figura 8.41: Paleta de colores. Gradientes obtenidos de la herramienta online Coolors.

La paleta se encuentra en el archivo *palette.js*, donde se declara la variable que va a ser exportada para establecerse como la paleta empleada por *Material UI* en nuestro proyecto. Observamos que en ella también se definen distintas tonalidades para las gráficas, una amplia variedad de tonos grises o los colores del texto y de los fondos, entre otros.

Por otro lado, las distintas opciones para la configuración de la tipografía se encuentran en el fichero *typography.js*, donde se puede configurar el tamaño de cada tipo de letra y la fuente empleada. En nuestro caso, hemos seleccionado *Helvética Neue*, ya que es muy utilizada en

diseño web, y en particular es la fuente que utiliza Twitter Web[105].

Otra función que permite *Material UI*, como hemos comentado, es sobreescribir componentes. En nuestro caso, utilizamos los componentes sobreescritos *Button*, *IconButton*, *Card*, *Input*, *List*, y *Typography*. Se pueden encontrar dentro de la carpeta *overrides*.

Por último, destacamos el uso de la utilidad de *styled()*, que utilizan todos los componentes de *Material UI*, y que nos ha permitido modificar algunos que están incluidos en el contexto de los temas para aplicarlo de manera distinta en cada uno de ellos.

Capítulo 9

Evaluaciones

Una vez llevada a cabo la implementación, es necesario realizar una evaluación de resultados para comprobar qué objetivos se han cumplido, cuáles no, y qué modificaciones llevar a cabo. Observamos que evaluamos el funcionamiento de la aplicación tomando dos puntos de vista; interfaz de usuario y análisis de lenguaje natural.

9.1. Evaluación: Interfaz de usuario

Durante el transcurso del proyecto, se realizaron varias sesiones de seguimiento y evaluación con los tutores del TFG que resultaron en múltiples mejoras del proyecto. Sin embargo, una vez finalizado, decidimos llevar a cabo una evaluación con usuarios finales con el objetivo de obtener información detallada y fiel a las necesidades del público final de la interfaz.

Observamos que los 10 principios de diseño de Jakob Nielsen se han tenido en cuenta en el diseño de la inferfaz. Sin embargo, al ser una interfaz simple y haber contado con el apoyo de expertos durante su desarrollo, hemos optado por no realizar una evaluación heurística y centrarnos en los usuarios finales.

9.1.1. Plan de evaluación

Identificación del propósito y los objetivos de la evaluación

El primer paso de la evaluación con usuarios es establecer el propósito y los objetivos de ésta. Buscamos identificar qué aspectos de la interfaz funcionan bien, cuáles no, y por qué motivo. Esto nos serviría para establecer el posible trabajo a futuro y las mejoras que se pueden implementar. Extraemos los siguientes objetivos de alto nivel:

- Determinar si la interfaz resulta fácil de entender y utilizar por los usuarios finales.
- Identificar aquellos elementos de la interfaz que dificultan al usuario la realización de sus tareas por resultar poco intuitivos o por propiciar que el usuario cometa errores.
- Identificar aquellos elementos que, por el contrario, el usuario echa en falta.

- Encontrar la mejor forma de solucionar cada uno de los problemas del punto anterior, sin degradar la usabilidad de otras partes de la interfaz.
- Detectar qué elementos gustan y resultan intuitivos al usuario.

Identificar los requisitos que deben cumplir los participantes

Buscamos participantes que posean las características detectadas en la fase de investigación (capítulo 6). Al tratarse de un perfil muy variado, vamos a intentar reunir a un grupo de usuarios finales que representen suficiente diversidad en cuanto a rango de edad, profesión, organismo en el cuál se desempeña dicha profesión, si han sufrido situaciones de injusticia laboral o si tienen algún interés en eliminarlas activamente.

Descripción del diseño experimental

A cada participante, se le va a explicar en qué consiste la evaluación y se le va a pedir realizar un cuestionario inicial. El objetivo de este último es conocer sus características, así como el nivel de conocimiento que poseen relacionado con nuestra aplicación. De esta manera, podemos entender y analizar mejor los resultados obtenidos, y ver si las características del usuario pueden estar afectando a su interacción con la interfaz.

Posteriormente, un moderador le pedirá al participante que interactúe con la página web, resolviendo una serie de tareas. Además, se le enviará un cuestionario anónimo para que pueda dar su opinión sin estar condicionado, lo que nos permitirá conocer su punto de vista.

El objetivo es recoger datos cualitativos (elementos que gustan/disgustan, que son fáciles/difíciles de usar, nivel de satisfacción/frustración del usuario...). No obstante, también se tendrán en cuenta algunos cuantitativos (nº de tareas bien realizadas, nº de tareas para las que se necesita ayuda, tiempo de respuesta...).

Concluimos indicando las métricas utilizadas para evaluar las tareas:

- Tiempo de respuesta: *Rápido* (menor a $5sec$), *Medio* (entre 5 y $15sec$) o *Lento* (más de $15sec$).
- Necesidad de ayuda: ✓(ninguna ayuda), ✓✗(ha necesitado guía) o ✗(no ha sabido).

9.1.2. Materiales necesarios para la evaluación

Antes de realizar la prueba de evaluación, hicimos las siguientes preparaciones:

- Guión para la orientación previa: Anexo A.1.
- Cuestionario previo: Anexo A.2.
- Tareas y escenarios

1. Dime el total de denuncias
2. Dime la cantidad de personas que han puesto denuncias
3. ¿Puedes leerme una denuncia?
4. ¿Qué denuncias ha puesto el usuario X?
5. Busca una denuncia relacionada con "vacaciones"
6. ¿Qué denuncia ha recibido la mayor cantidad de likes?
7. Si quisieses poner una denuncia desde el dashboard, ¿cómo lo harías?
8. Filtra por fechas para ver datos correspondientes únicamente al último mes
9. Indícame el perfil del usuario que más denuncias ha puesto
10. Visita su perfil
11. ¿Cuál es el hashtag más utilizado?
12. ¿Cuál es la palabra más utilizada en denuncias?
13. ¿Cómo evoluciona dicha palabra a lo largo del tiempo?
14. ¿Cómo varía la cantidad de denuncias con el tiempo?
15. Haz que en el gráfico solo aparezca la cantidad de retweets en el tiempo
16. ¿En qué momento del año hay mayor número de denuncias?
17. Si quiero saber más sobre la iniciativa, ¿dónde me meto para averiguarlo?

- Cuestionario final: Anexo A.3.

9.1.3. Resultados de la evaluación

Dedicamos esta sección a mostrar los resultados obtenidos durante el proceso de evaluación.

En primer lugar, el cuadro 9.1 muestra las métricas resultantes y la acumulación de las observaciones realizadas durante las evaluaciones. Para calcular las métricas, hemos seguido las siguientes reglas:

– *Resolución*: Si más de un tercio de los usuarios necesitan ayuda, entonces consideramos que se necesita ayuda. Si más de un cuarto no saben realizar una tarea, entonces no se sabe realizar la tarea. El motivo de estas decisiones es que, para que la aplicación sea intuitiva, las personas que necesiten ayuda o no sepan realizar tareas deben ser una minoría.

– *Tiempo de respuesta*: Al tratarse de un dato cuantitativo, hemos optado por realizar la media de los tiempos.

Si se quiere ver en detalle, de forma individualizada, la información, acceder a:

- **Sesión 1**:

  - Cuestionario previo: Anexo A.4.
  - Tareas y observaciones: Anexo A.1.
- **Sesión 2**:
  - Cuestionario previo: Anexo A.4.
  - Tareas y observaciones: Anexo A.2.
- **Sesión 3**:
  - Cuestionario previo: Anexo A.4.
  - Tareas y observaciones: Anexo A.3.
- **Sesión 4**:
  - Cuestionario previo: Anexo A.4.
  - Tareas y observaciones: Anexo A.4.
- **Sesión 5**:
  - Cuestionario previo: Anexo A.4.
  - Tareas y observaciones: Anexo A.5.

Por otro lado, la información recogida durante los cuestionarios finales anónimos fue tanto cuantitativa como cualitativa. En el cuadro 9.2 se observa la primera y, a continuación, la segunda:

- La iniciativa ha tenido muy buena acogida y ha parecido original. Los entrevistados han expresado su interés en tener un lugar donde se pueda dar visibilidad a situaciones de injusticia laboral, especialmente con la importancia que está cobrando a nivel social. Además, hacen hincapié en que se alimente desde un lugar de libre acceso por los trabajadores, como es la red social Twitter.
- La preocupación más repetida ha sido la relacionada con la fiabilidad de los datos. Consideran que al ser extraídos de Twitter, debería existir algún medio por el cuál se verifiquen.
- Muestran su interés en que la iniciativa pueda generalizarse a otro tipo de denuncias.
- Lo que más ha llamado la atención del dashboard ha sido su claridad y atractivo visual. Consideran que es intuitivo.
- Consideran que ellos mismos le darían uso a la aplicación en caso de estar buscando trabajo o vivir una situación de injusticia. También destacan el uso que pueden darle organismos sociales y laborales.
- Algunas funcionalidades que proponen añadir son: diferenciar sectores laborales, género, edad o localización.

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Rápido	
2	✓	Rápido	
3	✓	Rápido	
4	✓	Medio	Respetar @ y mayúsculas a veces da problema.
5	✓	Rápido	Pulsa al botón *enter* para enviar la búsqueda
6	✓	Lento	Tardaban en que se les ocurriese ordenar por likes.
7	✓	Lento	Poco intuitivo posición y el formato del botón
8	✓	Medio	Los filtros están fuera del plano de visión. No resalta.
9	✓	Medio	
10	✓	Medio	
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Rápido	No está acostumbrado a varias escalas.
15	✓	Rápido	Piensan en clickear para mostrar, pero en la interfaz eso sirve para ocultar.
16	✓	Rápido	
17	✓	Rápido	

Cuadro 9.1: Recopilación de datos de las sesiones de evaluación

- Algunas observaciones que realizan:
  - Cambiar la localización del acceso a Twitter para denunciar.
  - Implementar la interacción con Twitter dentro del dashboard.
  - El gráfico de doble escala puede no ser comprendido por todo el mundo.
  - Permitir que las búsquedas no necesiten el @ en los usuarios o que no hagan distinción entre mayúscula y minúscula. Resaltar la palabra buscada en el texto.
  - Hacer el acceso a los filtros más visible.

9.1.4. Conclusiones de la evaluación

A la vista de los resultados obtenidos, las conclusiones más importantes que extraemos para el futuro desarrollo del proyecto son las siguientes:

Sentencia	**Grado de conformidad**
La aplicación web me ha resultado fácil de usar	4.6
La interfaz me resulta agradable a la vista	4.6
Me resulta útil la información que muestran las gráficas	4
La información que muestra la aplicación me parece fiable	3.2
Es sencillo acceder a Twitter	4.6
Es útil acceder a Twitter	4.4
Es sencillo filtrar los datos	4.8
Es útil filtrar los datos	4.6
Es sencillo realizar búsquedas	4.8
Es útil realizar búsquedas	4.6
Es sencillo ver las denuncias	4.8
Es útil ver las denuncias	4.8
Es sencillo entender las gráficas	4
Es útil entender las gráficas	4

Cuadro 9.2: Datos cuantitativos obtenidos de los cuestionarios finales. El grado de conformidad se mide de 1 a 5 (1: Totalmente en desacuerdo, 5: Totalmente de acuerdo)

- La aplicación y la iniciativa han resultado interesantes y parece que serían usadas por un gran número de personas. Además, han aparecido varias posibles líneas de trabajo que podrían seguirse (integrar Twitter en el dashboard, generalizar a otro tipo de denuncias, tratar más datos...). Por lo tanto, el proyecto parece tener continuidad.
- El dashboard resulta estéticamente atractivo e intuitivo. No obstante, hemos detectado algunas mejoras que podrían implementarse. Algunas relevantes son:
  - Hacer más visibles algunos botones como el de realizar denuncias, ordenar o filtrar.
  - Permitir en búsquedas utilizar el botón *enter*. Además, no exigir que las búsquedas por usuario necesiten @ y respetar mayúsculas y minúsculas.
  - Mantener las gráficas simples para un usuario general. El doble eje podría ser confuso.
- Un punto débil que ha sido remarcado por varios usuarios es el de la fiabilidad del contenido de las denuncias. Consideran que algunos datos recogidos pueden estar afectados por información falsa o creada por automatizaciones. Es por ello que creemos necesario tener esto en cuenta a la hora de recoger los datos desde Twitter, atravesando algún filtro donde las denuncias sospechosas sean marcadas como tal. Además, sería positivo incluir en el dashboard algún aviso sobre la procedencia de los datos.

9.2. Evaluación: Análisis del lenguaje natural

Tras la creación del diccionario y el desarrollo del algoritmo de clasificación, resultaba esencial medir y analizar la eficacia que tenían a la hora de detectar denuncias laborales. Es por ello que se llevó a cabo una fase de evaluación en la que se analizaron distintas métricas. Para poder desarrollarla, creamos un corpus formado por denuncias laborales y otro con una temática totalmente distinta.

- Corpus con 150 publicaciones que no habíamos usado de la cuenta de @trabajosruineros.
- Corpus con el mismo número de publicaciones y similar longitud, con *reviews* de películas.

Optamos por analizar las métricas de *exactitud* (*accuracy*), *exhaustividad* (*recall*), *ROC-AUC*, *precisión* y la media *F1*, con el objetivo de quedarnos con la métrica que diese mejores resultados para nuestro caso concreto. Inicialmente, usamos el 4 % del diccionario (3 palabras), cuyas estadísticas se pueden apreciar en la columna "Iniciales" en el cuadro 9.3. Sin embargo, una vez comenzó a funcionar el programa decidimos aumentar el porcentaje a un 5 % (4 palabras) y, finalmente, a un 5,34 % (5 palabras). Los dos resultados anteriores se observan en las dos últimas columnas del cuadro 9.3.

Métricas	**Iniciales** 3 palabras 4 % del diccionario	**Previas** 4 palabras 5 % del diccionario	**Iniciales** 4 palabras 5.34 % del diccionario
Confusion matrix	$\begin{bmatrix} 147 & 5 \\ 13 & 139 \end{bmatrix}$	$\begin{bmatrix} 149 & 3 \\ 28 & 124 \end{bmatrix}$	$\begin{bmatrix} 151 & 1 \\ 44 & 108 \end{bmatrix}$
Accuracy	0.94	0.9	0.85
Recall	0.91	0.82	0.71
ROC-AUC	0.94	0.9	0.85
Precision	0.97	0.98	0.99
F1	0.94	0.89	0.83

Cuadro 9.3: Métricas obtenidas

La primera observación que realizamos es que con 3 palabras salían muchos textos relacionados con situaciones laborales, pero no siempre eran denuncias. En nuestro caso, nos interesaba primar la precisión antes que la exhaustividad, ya que una vez los tweets se almacenen en la blockchain, van a permanecer siempre guardados sin posibilidad de modificarse o eliminarse. Por lo tanto, hemos considerado preferible aumentar el número de verdaderos positivos, a pesar de que ello implique directamente la existencia de un mayor número de falsos negativos. Es por ello que, como se aprecia en las imágenes, el aumento del porcentaje del diccionario nos ha proporcionado mejores resultados.

Una vez obtenidos unos resultados tan buenos, decidimos probar si de verdad funcionaban tan bien en el entorno final, es decir, en Twitter. Realizamos, por lo tanto, una segunda evaluación. Esta evaluación consistió en tomar 100 tweets que clasificador por el algoritmo como

negativos y otros 100 que ha clasificado como positivos. Posteriormente, manualmente, calculamos el porcentaje de verdaderos positivos y negativos, y de falsos positivos y falsos negativos. Representamos los resultados en la matriz de confusión de la figura. Estos tweets se pueden encontrar dentro del repositorio de *GitHub* en *"Previous-work-related-to-data-recollection"* en la carpeta JSON. 9.1.

- De los 100 negativos que se observan en el fichero *classified_as_negative.json* solamente dos son falsos negativos. Se encuentran en las líneas 34 y 48 del fichero *false_negatives.json*.
- De los 100 positivos que hemos recogido en *classified_as_positive.json*, hay 28 falsos. Se encuentran en las líneas: 1, 4, 9, 17, 23, 26, 29, 30, 39, 42, 43, 46, 47, 50, 53, 62, 65, 66, 67, 68, 74, 76, 77, 78, 84, 86, 94 y 95. También podemos verlos más claramente en el fichero *false_positives.json*

		Predicción	
		0	1
Realidad	0	98	28
	1	2	72

Figura 9.1: Matriz de confusión de la segunda evaluación

Las métricas obtenidas de este segundo análisis, utilizando los datos de la matriz de confusión, son: Accuracy (0.85), Recall (0.97), Precision (0.72) y F1 (0.83). Cómo se puede observar, los resultados son distintos a los que se previeron en la evaluación inicial. La mayor diferencia se ve en la precisión, que ha disminuido considerablemente. Esto se debe a que los textos con los que creamos el diccionario eran considerablemente más largos que el tweet promedio. Además, el texto del corpus que usamos como negativos era muy distinto a las denuncias, por lo que la existencia de falsos positivos era más complicada que en esta última, en dónde la consulta que realizábamos para recoger tweets devolvía datos relacionados con la temática de denuncia laboral. Aún así, seguimos obteniendo un valor de precisión muy bueno.

Cabe destacar también que en esta segunda vuelta al realizarse la clasificación de forma manual hay un factor perceptual, es decir, existe la posibilidad de que otra persona considere como denuncias tweets que han sido considerado como falsos positivos o viceversa, ya que existen muchas publicaciones cuya finalidad es visibilizar condiciones y situaciones laborales precarias sin tener como finalidad realizar una denuncia pública.

Capítulo 10

Conclusiones y trabajo a futuro

10.1. Conclusiones

El trabajo desarrollado y presentado hasta el momento, nos ha permitido obtener una aplicación que cumple los principales **objetivos** definidos al inicio del proyecto. Se encuentran en el capítulo 1. En primer lugar, **Injustweet** proporciona una plataforma en la que se da **visibilidad**, gracias al uso de datos y estadísticas, a denuncias laborales llevadas a cabo en Twitter. Como se ha visto en los resultados de la evaluación con usuarios, se ha conseguido implementar una **interfaz intuitiva** y **atractiva**, además de **funcional**, que cumple la mayoría de los requisitos propuestos. Por otro lado, gracias a las técnicas de **análisis de lenguaje natural** empleadas, se ha conseguido obtener, de entre el conjunto de datos de Twitter, **denuncias laborales** de personas **hispanohablantes**. Como se ha visto en el apartado de evaluación, con muy buenos resultados. Finalmente, observamos que se ha hecho frente, en gran medida, al problema de la **censura** de datos en Twitter. Esto ha sido posible gracias al **almacenaje** de datos **distribuido**, lo que garantiza que aunque se elimine el contenido original, los datos sigan estando guardados.

Sin embargo, y a pesar de haber obtenido un resultado final del que estamos muy orgullosos, debido a lo ambicioso que era el proyecto tuvimos que prescindir de algunas funcionalidades. Además, por la propia naturaleza de las tecnologías usadas, la aplicación presenta **limitaciones**. Proporcionamos los siguientes ejemplos:

- Como vimos en el capítulo 6.2 de requisitos, algunos fueron determinados no prioritarios y no se llevaron a cabo. El principal está relacionado con mostrar información más diversa (tipo de denuncia, localización, sector laboral...). Requería de un desarrollo y análisis mucho más elaborado para poder extraer dicha información, así como una mayor cantidad de datos.
- Utilizar tecnologías distribuidas ha sido muy útil para garantizar que, aunque la fuente original de la denuncia sea eliminada, el contenido permanezca. Sin embargo, esto puede plantear un problema. Si se insertan datos falsos en la blockchain, permanecerán en ella para siempre.

Destacamos que no sólo ha sido gratificante el resultado final que acabamos de exponer, sino también el aprendizaje que hemos obtenido en el proceso. El desarrollo de este proyecto ha resultado ser muy enriquecedor desde distintos puntos de vista. Por un lado, nos hemos enfrentado a **tecnologías** con las que la mayoría de nosotros **no** estábamos **familiarizados**, como han sido el uso de *React* en el *frontend* o de tecnologías distribuidas en el *backend*. Es por ello que, aunque tuvimos un periodo de trabajo previo específicamente destinado a su aprendizaje, al final gran parte del TFG ha sido un constante aumento de conocimientos.

Por otro lado, **poner en práctica** algunas de las **metodologías** que estudiamos en la carrera, como *scrum* o *dcu*, nos ha proporcionado una perspectiva más clara sobre cómo se organizan y llevan a cabo proyectos. Especialmente, teniendo en cuenta que el equipo estaba formado por seis personas, y que nosotros teníamos que ser los que nos **planificásemos** y **coordinásemos**. La mayor dificultad que encontramos, por ejemplo, fue cuándo tuvimos que crear el primer producto viable uniendo el trabajo que habíamos realizado en paralelo y cómo, a partir de ese momento, la comunicación y organización fue esencial para implementar cambios.

Finalmente, queremos destacar el aprendizaje obtenido gracias a la propia **naturaleza** de la **iniciativa**. Al tener una utilidad y objetivo **social**, ha sido un reto ver cómo aplicar nuestros conocimientos tecnológicos para solucionar un **problema real**, así como tener que **empatizar** con aquellos que van a usarlo y que carecen de dichos conocimientos.

En su conjunto, por lo tanto, hemos obtenido un producto que, a pesar de sus limitaciones, satisface los principales objetivos de la aplicación. Además, gracias a su proceso de desarrollo hemos podido adquirir competencias que nos serán muy útiles en nuestros futuros proyectos.

10.2. Trabajo a futuro

En base a los problemas que se han encontrado a lo largo del proyecto, los resultados obtenidos y las evaluaciones realizadas, planteamos unas líneas de trabajo que se podrían seguir en el futuro. Animamos a que todas ellas se realicen incentivando y usando software libre, ya que ha sido uno de los pilares del proyecto.

10.2.1. Sobre la recogida de datos de Twitter

A pesar del trabajo realizado, existen diversas formas en las cuales mejorar y ampliar la recolección de datos. Proponemos las siguientes.

- Extraer mayor cantidad y variedad de datos de los tweets (género, edad, localización, tipo de denuncia...).
- El algoritmo podría ser mejorado de múltiples maneras, el diccionario podría ser mucho más extenso o incluso emplear distintos sistemas de clasificación de textos (por ejemplo, redes neuronales).

- Emplear consultas más complejas a la hora de obtener tweets, esto aceleraría enormemente el número de denuncias obtenidas.
- Se podrían emplear mejores servidores para ejecutar el código, ya que debido a los recursos de los que disponen la recolección de datos es más lenta que en un equipo de uso convencional.
- Un sistema de detección de denuncias falsas podría ser una buena incorporación al proyecto, aunque podría presentar muchos fallos, dejando fuera denuncias que pudieran ser ciertas.

10.2.2. Sobre el uso de la blockchain

Mostramos tres vías que podrían seguirse para mejorar la integración de este proyecto con un uso más eficiente de tecnologías basadas en blockchain.

- Rediseñar el sistema de acceso a la memoria del contrato que realiza el código en *javascript* evitando usar APIs de otros sistemas. Los contratos inteligentes son públicos y la información que estos manejan también. Planteamos que sea el propio código de la API que hemos creado el que acceda de forma directa a la memoria del contrato. De esta forma, obtiene los datos minimizando las funciones en la blockchain que incrementan el gasto y reduce el código usado en *Solidity*.
- Utilizar código de bajo nivel *EVM* para reducir el número de instrucciones máquina que genera el compilador, de forma que se optimice el consumo de gas del contrato.
- Quizá hay otras formas de obtener la información necesaria para realizar la actualización de información que estén menos limitadas en cuanto a flujo de ejecución, lo que mejoraría significativamente el tiempo empleado en estas funciones y haría esta parte más escalable.

10.2.3. Sobre la interfaz de usuario

Por un lado, tras las evaluaciones realizadas con usuarios finales, proponemos realizar cambios en el diseño de la aplicación que mejoren la experiencia de usuario.

- Destacar elementos importantes como el botón para filtrar por fecha, ordenar o publicar la denuncia en Twitter.
- Mejorar la búsqueda dentro de las denuncias (marcar la palabra buscada en el texto o no exigir el uso de la @).
- Valorar la integración de la interacción con Twitter dentro de la aplicación, y permitir que el usuario realice las acciones sólo desde nuestra web.

Por otro lado, planteamos aumentar las funcionalidades actuales del dashboard haciendo uso de los requisitos no implementados y proponiendo nuevas ideas.

- Aumentar las opciones de filtro y personalización en función de distintos parámetros: género, edad, tipo de denuncia, sector laboral al que pertenece etc. Este cambio es altamente dependiente de la recolección de datos.
- A pesar de que no todos los tweets incluyen la localización, se podría valorar utilizarla para mostrar un mapa con los tweets disponibles o para realizar un filtrado sobre ellos.
- Permitir comparativas en base a los parámetros anteriores, incluyendo algún tipo de sistema de reputación.
- Incluir un sistema de recogida de denuncias por parte de aquellos usuarios que no tengan cuenta de Twitter. Podría ser anónima o incluyendo un nombre. Valorar la opción de publicarlos desde una cuenta oficial de Injustweet.

Capítulo 11

Conclusions and future work

11.1. Conclusions

It can be concluded at this stage that we have been able to develop an application that satisfies the **main goals** established at the beginning of the project. They can be found at chapter 2. First of all, **Injustweet** is a platform that gives **visibility** to workplace denouncements thanks to the use of graphs and statistics. According to the results obtained in the evaluation stage with users, we have been able to implement an **interface** wich is not only **intuitive** and **atractive**, but also **functional**. It satisfies most of the requirements proposed. Besides that, even though there is a gigantic amount of data in Twitter, we have been able to obtain only those related to **workplace denouncements** from **Spanish-speaking** users. That has been possible thanks to the use of **natural language processing** techniques (NLP). As shown in the stage of evaluation, the results obtained from the analysis are very accurate. Finally, on another note, we have also been able to mitigate the problem of **censorship** in Twitter. It has been possible thanks to the use of **distributed storage**, which guarantees that even if data is deleted from Twitter, we will still have a copy of it.

However, in spite of obtaining such good results and a final product that makes us feel proud, we have to take into account that it has its **limitations**. The project was very ambitious, that is why we had to drop some requirements at early stages. Moreover, we faced technological impediments. Below you can view some examples.

- As shown in chapter 6.2, some requirements were considered non-priority and dropped. The most important one is that we had to limit the number of characteristics obtained from the data (type of complaint, localization, laboral sector...). To be able to implement it, we would have needed more data and analytical capability.
- As we mentioned before, the use of distributed storage has enable us to guarantee that even if data is deleted from Twitter, there will always be a copy of it. However, that could raise an issue. If fake data is added to the blockchain, it will be kept forever.

It is at this point that we would like to highlight that we are not only proud of the results just shown, but also of the process that led as to achieve them. The development of this

project has turned out to be very enriching and has helped us **learn** in different ways. On one hand, we have used **technologies** we had never worked with before. For example, we built the frontend with *React* and the backend with blockchain. That is way, in spite of having a period of time dedicated to learning, we spent most of the project researching.

On the other hand, we have **put into practice methodologies** that we only knew from a theoretical approach. Therefore, we have better understood how projects are managed and carried out. Especially taking into account that we were a team of six, and that we had to **plan** and **coordinate** the project from start to finish. To give an example, one of the biggest challenges we faced was the time where we had to put all the project together. From that point on, communication was a key part of it. Every change made had to be discussed and planned beforehand.

Finally, we want to emphasize that due to the nature of the initiative we have learned how to solve a **real-world** problem and **empathize** with the people who are involed in it. We were able to use our technological knowledge to develop an application by bearing in mind that users do not usually have it.

To conclude, we have created a product that even with it's limitations, it is able to satisfy the main goals established. Moreover, thanks to the process followed, we have acquired skills that will be very useful in future projects.

11.2. Future work

While developing our project, and thanks to the results obtained in the stage of evaluation, we have been able to detect some problems and changes that could be made. We propose them as future work and encourage anyone interested to develop them by using free software.

11.2.1. Data recollection

We propose the following ways of improving the way in which data is recollected and the kind of data obtained.

- To obtain more quantity and diversity of data (gender, age, localization, type of complaint...).
- To improve the algorithm used. For example, by incrementing the size of the dictionary or using other methods of classification such as neural networks.
- To increase the complexity of the queries made so that tweets could be obtained at a higher speed.
- To consider using faster servers. Right know we face some limitations.
- To include a system which could help detect fake complaints.

11.2.2. Blockchain

The efficiency of this project could be improved by making some changes in the backend. We propose the following ideas related to the use of the blockchain.

- To change the system in which we access the data of the smart contract so that we avoid using APIs from other systems. We propose to implement those functions in our *javascript* code to decrease the costs of the transactions.
- To use code *EVM* to reduce the number of machine instructions. This could further decrease the costs of the transactions.
- To consider obtaining the information required to update the data from other sources. Right now the API from Twitter is very limited, so this change would make the proyect more scalable.

11.2.3. User interface

On one hand, after having interviewed many users, we have found some changes that could be made in order to improve the user experience.

- To give more visibility to important features such as the button used to filter, sort and publish a complaint in Twitter.
- To improve the way in which tweets are searched for (to highlight the word or to not require the use of `@`).
- To consider the possibility of including all the interaction with Twitter inside the dashboard, so that users can make every possible action without leaving the web page.

On the other hand, we propose other functionalities that could be added.

- To increase the filter options and the capacity for giving a personalized experience by including more information (gender, age, type of complaint, laboral sector...).
- To include maps and information related to localization.
- To include diagrams that compare tweets according to the topics mentioned before. It could be a good idea to also include a reputation system.
- To implement a way of making complaints. It could be anonymous. Consider the possibility of publishing them from an official Injustweet account.

Anexos

Apéndice A

Entrevistas

A.1. Guión inicial

"La sesión de hoy va a consistir en una evaluación de nuestra aplicación, que es un dashboard en el que se muestran gráficos e información sobre denuncias laborales. Yo voy a moderar la sesión, y tú serás el usuario de la aplicación.

Vamos a plantearte una serie de tareas que tendrás que intentar resolver en la página web. Durante estas tareas, me gustaría que dijeras en voz alta todo lo que se te pase por la cabeza. Por ejemplo, los problemas que encuentras, las cosas que te confunden o lo que más te guste.

Yo no voy a intervenir mientras tú respondes. Es probable que haya momentos de silencio o que te anime a que sigas resolviéndolo sin mi ayuda. Esto es porque queremos saber cómo un usuario cualquiera utilizaría la aplicación sin ayuda externa.

Cuando hayamos terminado, te haremos una pequeña entrevista para recoger conclusiones. Si tienes alguna duda sobre el proceso o sobre lo que tienes que hacer, puedes hacerla ahora."

A.2. Cuestionario previo

- ¿Cuántos años tienes? (Respuesta corta)
- ¿Has sufrido alguna situación de injusticia laboral? (Sí/No)
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti? (Sí/No)
- ¿Alguna vez has denunciado situaciones de injusticia laboral? (Sí/No)
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...) (Respuesta corta)
- ¿Trabajas en el sector público o privado? (Público/Privado/Otro)
- ¿Has utilizado alguna vez un dashboard? (Utilizo a diario/He usado varias veces/Me suena haber visto alguno/Nunca)

- ¿Tienes experiencia analizando gráficas? (Sí, todo tipo de gráficas/Sí, salvo que sean muy complejas/Sí, sólo gráficas simples/Ninguna experiencia)
- ¿Cómo de cómodo te sientes entendiendo el lenguaje de Twitter (hashtags, likes, retweets, usuarios...)? (Soy experto/a/Algunos conceptos me fallan/No sé nada)

A.3. Cuestionario final

- ¿Qué te ha parecido la aplicación y la iniciativa? (Respuesta larga)
- ¿Qué es lo que más te ha llamado la atención? ¿Por qué? (Respuesta larga)
- ¿Crees que darías uso a la aplicación? ¿Por qué sí/no? (Respuesta larga)
- ¿Hay funciones que te hayan gustado especialmente? ¿Alguna que no? (Respuesta larga)
- ¿Hay algo que no te haya quedado claro con ella? (Respuesta larga)

- La aplicación web me ha resultado fácil de usar. En caso contrario, ¿por qué? (1 total desacuerdo - 5 total acuerdo)
- La interfaz me resulta agradable a la vista. En caso contrario, ¿por qué? (1-5)
- Me resulta útil la información que muestran las gráficas. Indica si hay algo que no. (1-5)
- La información que muestra la aplicación me parece fiable. Indica posibles preocupaciones. (1-5)

- Es sencillo acceder a Twitter. (1-5)
- Es útil acceder a Twitter. (1-5)
- (Comentarios sobre la interacción con Twitter)

- Es sencillo filtrar los datos. (1-5)
- Es útil filtrar los datos. (1-5)
- (Comentarios sobre los filtros)

- Es sencillo realizar búsquedas. (1-5)
- Es útil realizar búsquedas. (1-5)
- (Comentarios sobre las búsquedas)

- Es sencillo ver denuncias. (1-5)
- Es útil ver denuncias. (1-5)
- (Comentarios sobre las denuncias)

- Es sencillo entender las gráficas. (1-5)
- Son útiles las gráficas. (1-5)
- (Comentarios sobre las gráficas)

- ¿Has echado en falta algo que consideras importante a la hora de obtener información sobre precariedad laboral? En caso afirmativo, ¿el qué? (Sí/No)
- ¿Recomendarías la aplicación a ...? (Respuesta larga)
- Si quieres añadir recomendaciones, opiniones, críticas o cualquier cosa que no haya quedado reflejada antes, puedes hacerlo. (Respuesta larga)

A.4. Desarrollo entrevistas

Cuestionarios previos

Usuario 1

- ¿Cuántos años tienes?: 51
- ¿Has sufrido alguna situación de injusticia laboral?: Sí
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti?: Sí
- ¿Alguna vez has denunciado situaciones de injusticia laboral?: Sí
- Si quieres desarrollar dichas situaciones de injusticia laboral, puedes hacerlo aquí: CONTRATACIÓN EN CATEGORÍA INFERIOR, "CONTRATACIÓN" SIN FORMALIZACIÓN DE CONTRATO, CONTRATACIÓN MEDIA JORNADA TRABAJANDO MÁS TIEMPO, BECARIO SIN COBRAR/FORMALIZACIÓN DE BECA/REALIZANDO TRABAJO SIN FORMACIÓN, ACOSO LABORAL, DESPIDO IMPROCEDENTE...
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...): FUNCIONARIO
- ¿Trabajas en el sector público o privado?: Público
- ¿Has utilizado alguna vez un dashboard?: He usado varias veces
- ¿Tienes experiencia analizando gráficas?: Sí, salvo que sean muy complejas

- ¿Cómo de cómodo/a te sientes entendiendo el lenguaje de Twitter? (Hashtags, likes, retweets, usuarios...): Algunos conceptos me fallan

Usuario 2

- ¿Cuántos años tienes?: 22
- ¿Has sufrido alguna situación de injusticia laboral?: Sí
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti?: Sí
- ¿Alguna vez has denunciado situaciones de injusticia laboral?: No
- Si quieres desarrollar dichas situaciones de injusticia laboral, puedes hacerlo aquí: prácticas sin remunerar, explotación laboral
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...): estudiante de máster de la rama biosanitaria
- ¿Trabajas en el sector público o privado?: Público
- ¿Has utilizado alguna vez un dashboard?: He usado varias veces
- ¿Tienes experiencia analizando gráficas?: Sí, todo tipo de gráficas
- ¿Cómo de cómodo/a te sientes entendiendo el lenguaje de Twitter? (Hashtags, likes, retweets, usuarios...): Soy experto/a

Usuario 3

- ¿Cuántos años tienes? : 58
- ¿Has sufrido alguna situación de injusticia laboral?: Sí
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti?: Sí
- ¿Alguna vez has denunciado situaciones de injusticia laboral?: No
- Si quieres desarrollar dichas situaciones de injusticia laboral, puedes hacerlo aquí: En mi caso fueron hace años con una situación personal difícil y los tuve que asumir, hoy no se dan y si se dieran no lo aceptaría. He presenciado situaciones injustas relacionadas con el maltrato verbal y menosprecio de las personas que no denuncié en su momento al presenciarlas y me arrepiento de no haberlo hecho.
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...): Soy profesional en una organización de IT y me dedico a la Gestión y seguimiento de los servicios que prestamos a terceros.
- ¿Trabajas en el sector público o privado?: Privado

- ¿Has utilizado alguna vez un dashboard?: Utilizo a diario
- ¿Tienes experiencia analizando gráficas?: Sí, salvo que sean muy complejas
- ¿Cómo de cómodo/a te sientes entendiendo el lenguaje de Twitter? (Hashtags, likes, retweets, usuarios...): Algunos conceptos me fallan

Usuario 4

- ¿Cuántos años tienes?: 24
- ¿Has sufrido alguna situación de injusticia laboral?: Sí
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti?: Sí
- ¿Alguna vez has denunciado situaciones de injusticia laboral?: Sí
- Si quieres desarrollar dichas situaciones de injusticia laboral, puedes hacerlo aquí: Me plantearon hacerme un contrato laboral con unas condiciones laborales específicas que luego cambiaron a la semana porque consideraron que a ellos no les beneficiaba.
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...): Pluriempleada
- ¿Trabajas en el sector público o privado?: Privado
- ¿Has utilizado alguna vez un dashboard?: Me suena haber visto alguno
- ¿Tienes experiencia analizando gráficas?: Ninguna experiencia
- ¿Cómo de cómodo/a te sientes entendiendo el lenguaje de Twitter? (Hashtags, likes, retweets, usuarios...): Soy experto/a

Usuario 5

- ¿Cuántos años tienes?: 22
- ¿Has sufrido alguna situación de injusticia laboral?:Sí
- ¿Has presenciado alguna situación de injusticia laboral que no te haya pasado a ti?:Sí
- ¿Alguna vez has denunciado situaciones de injusticia laboral?:No
- Si quieres desarrollar dichas situaciones de injusticia laboral, puedes hacerlo aquí:A mi hermano, que trabajaba en una empresa de vtc, le engañaron con las vacaciones y con los días de libranza y con las horas que tenía que hacer
- Indica tu situación profesional (estudiante, tipo de trabajo, paro...):Estudiante
- ¿Trabajas en el sector público o privado? Estudio en universidad privada
- ¿Has utilizado alguna vez un dashboard?:Nunca
- ¿Tienes experiencia analizando gráficas?:Ninguna experiencia
- ¿Cómo de cómodo/a te sientes entendiendo el lenguaje de Twitter? (Hashtags, likes, retweets, usuarios...):Soy experto/a

Tareas y escenarios

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Rápido	
2	✓	Rápido	
3	✓✗	Medio	Algunos de los datos mostrados no eran denuncias. Eso generó confusión.
4	✓✗	Rápido	El primer instinto fue hacer click en el usuario para ir a Twitter. Se le indicó que lo hiciese en el dashboard.
5	✓	Rápido	
6	✓	Lento	
7	✓	Lento	Poco intuitivo posición botón
8	✓	Rápido	
9	✓	Medio	
10	✓	Rápido	
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Rápido	
15	✓	Rápido	Su primer instinto fue clickear para mostrar, pero en la interfaz eso sirve para ocultar.
16	✓	Rápido	
17	✓	Rápido	

Cuadro A.1: Evaluación: Sesión 1

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Rápido	
2	✓	Rápido	
3	✓	Rápido	
4	✓	Medio	Tener que poner la @ y no poder poner todo en minúscula fue problemático. Eliminar el filtro bien.
5	✓	Rápido	
6	✓	Medio	
7	✓	Medio	Poco intuitivo posición botón
8	✓	Medio	Los filtros están fuera del plano de visión. No gusta que estén a la altura del título.
9	✓	Medio	
10	✓	Rápido	
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Rápido	No está acostumbrado a varias escalas.
15	✓	Rápido	Su primer instinto fue clickear para mostrar, pero en la interfaz eso sirve para ocultar.
16	✓	Rápido	
17	✓	Rápido	

Cuadro A.2: Evaluación: Sesión 2

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Lento	Asocia el color rojo de la tarjeta de likes con negativo o alerta
2	✓	Rápido	
3	✓	Rápido	
4	✓	Rápido	Se ha dado cuenta de la @, y borra el filtrado correctamente.
5	✓	Rápido	Pulsa al botón *enter* para enviar la búsqueda
6	✓	Lento	
7	✓✗	Lento	Se va a información. Cuando lo encuentra, comenta que debería estar más arriba y más resaltado.
8	✓	Medio	
9	✓	Medio	
10	✓	Rápido	
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Medio	Primera respuesta relacionada con el gráfico de denuncias por día, pero comenta la respuesta seguidamente. No está acostumbrado a varias escalas.
15	✓	Rápido	
16	✓	Rápido	
17	✓	Rápido	Añadiría un botón aquí también para escribir denuncia

Cuadro A.3: Evaluación: Sesión 3

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Rápido	
2	✓	Rápido	
3	✓	Rápido	
4	✓	Rápido	
5	✓	Rápido	Comenta que le gustaría ver la palabra buscada de forma resaltada en los tweets
6	✓✗	Medio	Hace click en la tarjeta de likes
7	✓✗	Lento	Tarda en encontrar el botón tras subir y bajar varias veces
8	✓	Rápido	
9	✓	Rápido	
10	✓	Medio	Primero busca el usuario en la búsqueda, pero se le aclara que se quiere acceder a su perfil de Twitter
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Rápido	
15	✓	Rápido	
16	✓	Rápido	
17	✓	Rápido	

Cuadro A.4: Evaluación: Sesión 4

Tarea	Resolución	Tiempo respuesta	Observaciones
1	✓	Rápido	
2	✓	Rápido	
3	✓	Rápido	
4	✓✗	Medio	Le falta la @
5	✓	Rápido	Pulsa enter para buscar
6	✓✗	Lento	Se va hacia la última gráfica
7	✓	Rápido	
8	✓	Rápido	
9	✓	Rápido	
10	✓	Medio	
11	✓	Rápido	
12	✓	Rápido	
13	✓	Rápido	
14	✓	Rápido	
15	✓	Rápido	Espera que al pulsar sobre rts se mantenga solo esa gráfica. Comenta que es un poco liosa, no entiende por qué se muestran rts y likes también
16	✓	Rápido	
17	✓	Rápido	

Cuadro A.5: Evaluación: Sesión 5

Bibliografía

[1] Afinn, *Afinn*. URL: https://github.com/darenr/afinn.

[2] Airbnb, *Visx*. URL: https://airbnb.io/visx/.

[3] Amazon, *Amazon Web Services*. URL: https://aws.amazon.com/es/.

[4] T. Analytics, *Twitter analytics*. URL: https://analytics.twitter.com/about.

[5] D. G. W. Andreas M. Antonopoulos, *Mastering Ethereum: Building Smart Contracts and DApps*, O'Reilly Media, Inc., 1st ed., 2018.

[6] ApexCharts, *Apexcharts*. URL: https://apexcharts.com/.

[7] A. Banks and E. Porcello, *Learning React: Functional Web Development with React and Redux*, O'Reilly Media, Inc., 1st ed., 2017.

[8] O. W. Bertelsen and S. Bødker, *Activity theory*, HCI models, theories, and frameworks: Toward a multidisciplinary science, (2003), pp. 291–324.

[9] Black Lives Matter, *Black Lives Matter*. URL: https://blacklivesmatter.com/.

[10] body parser, *body parser*. URL: https://github.com/expressjs/body-parser.

[11] Bootstrap, *Bootstrap*. URL: https://getbootstrap.com/.

[12] Certifi, *Certifi*. URL: https://certifiio.readthedocs.io/en/latest/.

[13] Charts.js, *Charts.js*. URL: https://www.chartjs.org/docs/latest/.

[14] V. S. Code, *Visual studio code*. URL: https://code.visualstudio.com/.

[15] codecs, *codecs*. URL: https://docs.python.org/3/library/codecs.html.

[16] E. Commission, J. R. Centre, P. De Filippi, J. Brekke, E. Martínez Vicente, G. Lopéz Morales, C. Orgaz Alonso, B. Bodó, S. Meiklejohn, S. Hassan, K. Beecroft, A. Figueras Aguilar, D. Rozas, and M. Atzori, *Scanning the European ecosystem of distributed ledger technologies for social and public good : what, why, where, how, and ways to move forward*, Publications Office, 2020.

[17] M. Cooley, *Human-centered design*, Information design, (2000), pp. 59–81.

[18] M. Dalili and M. Dastani, *The role of twitter during the covid-19 crisis: A systematic literature review*, Acta Informatica Pragensia, 9 (2020).

[19] M. Developer, *Javascript*. URL: https://developer.mozilla.org/es/docs/Web/JavaScript.

[20] diagrams.net, *diagrams.net*. URL: https://www.diagrams.net/.

[21] M. Dorash, *Machine learning vs. rule based systems in nlp*. URL: https://medium.com/friendly-data/machine-learning-vs-rule-based-systems-in-nlp-5476de53c3b8.

[22] EasyOCR, *Easyocr*. URL: https://github.com/JaidedAI/EasyOCR.

[23] emoji, *emoji*. URL: https://github.com/carpedm20/emoji/.

[24] Ethereum, *Solidity github*. URL: https://github.com/ethereum/solidity.

[25] Ethereum.org, *Ethereum*. URL: https://ethereum.org/en/developers/docs/apis/javascript/.

[26] ——, *Ethereum Whitepaper*. URL: https://ethereum.org/en/whitepaper/.

[27] Etherscan, *Etherscan*. URL: https://etherscan.io/.

[28] express, *express*. URL: https://expressjs.com/es/.

[29] GIMP, *GIMP - image manipulation*. URL: https://www.gimp.org/.

[30] Github, *Github*. URL: https://github.com/.

[31] D. Glezos, *Everything you need to know about machine translation*. URL: https://es.transifex.com/blog/2015/machine-translation-101/.

[32] Googletrans, *Google*. URL: https://py-googletrans.readthedocs.io/en/latest/.

[33] A. Gruzd and P. . Mai, *Covid-19 misinformation portal – a rapid response project from the ryerson university social media lab.*

[34] J. Hao, Y. Sun, and H. Luo, *A safe and efficient storage scheme based on blockchain and ipfs for agricultural products tracking*, Journal of Computers, 29 (2018), pp. 158–167.

[35] T. hdwallet provider, *Truffle hdwallet-provider*. URL: https://github.com/trufflesuite/truffle/tree/master/packages/hdwallet-provider.

[36] Heroku, *Heroku*. URL: https://id.heroku.com/.

[37] HTTP, *HTTP docs*. URL: https://httpwg.org/specs/rfc7231.html.

[38] INE, *INE gráficos*. URL: https://www.ine.es/ss/Satellite?L=es_ES&c=INESeccion_C&cid=1259925463134&p=1254735110672&pagename=ProductosYServicios%2FPYSLayout.

[39] INFURA, *Infura*. URL: https://docs.infura.io/infura.

[40] INSTAGRAM @TRABAJOSRUINEROS, *Instagram @trabajosruineros*. URL: https://www.instagram.com/trabajosruineros/?hl=es.

[41] INSTAGRAPI, *instagrapi*. URL: https://adw0rd.github.io/instagrapi/.

[42] IPFS, *IPFS*. URL: https://ipfs.io/.

[43] JOBSTICE, *Jobstice*. URL: https://twitter.com/jobsticeoficial.

[44] JS2PY, *Js2Py*. URL: https://github.com/PiotrDabkowski/Js2Py.

[45] D. JURAFSKY AND J. H. MARTIN, *Speech and Language Processing*.

[46] KEYRING, *Keyring*. URL: https://github.com/jaraco/keyring.

[47] R. KUMARI, *4 types of machine translation in nlp*. URL: https://www.analyticssteps.com/blogs/4-types-machine-translation-nlp.

[48] LATEX PROJECT TEAM , *The Latex Project*. URL: https://www.latex-project.org/.

[49] M. LEARN, *Text classification: What it is and why it matters*. URL: https://monkeylearn.com/text-classification/.

[50] LICENCIA CREATIVE COMMONS, *Creative commons*. URL: https://creativecommons.org/licenses/by/4.0/.

[51] LICENCIA GNU, *GNU*. URL: https://www.gnu.org/licenses/gpl-3.0.html.

[52] D. K. MAHTO AND L. SINGH, *A dive into web scraper world*, in 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), IEEE, 2016, pp. 689–693.

[53] A. MATAMOROS-FERNÁNDEZ AND J. FARKAS, *Racism, hate speech, and social media: A systematic review and critique*, Television & New Media, 22 (2021), pp. 205–224.

[54] ME TOO MOVEMENT, *Me Too Movement*. URL: https://metoomvmt.org/.

[55] METAMASK. URL: https://metamask.io/.

[56] MINIMAL FREE, *Minimal Free Dashboard*. URL: https://github.com/minimal-ui-kit/material-kit-react.

[57] MongoDB, *MongoDB Atlas*. URL: https://www.mongodb.com/atlas/database.

[58] ——, *MongoDB Manual*. URL: https://www.mongodb.com/docs/manual/.

[59] A. Z. Mustafeez, *Text classification in nlp*. URL: https://www.educative.io/edpresso/text-classification-in-nlp.

[60] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009.

[61] R. T. Nandita Krishnan, Jiayan Gu and L. C. Abroms, *Research note: Examining how various social media platforms have responded to covid-19 misinformation*, Harvard Kennedy School (HKS) Misinformation Review, (2021).

[62] J. Nielsen, *Ten usability heuristics*, 2005.

[63] Nivo, *Nivo*. URL: https://nivo.rocks/.

[64] node fetch, *node-fetch*. URL: https://github.com/node-fetch/node-fetch.

[65] node schedule, *node-schedule*. URL: https://github.com/node-schedule/node-schedule.

[66] Node.js, *Node.js*. URL: https://nodejs.org/es/.

[67] npm, *About npm*. URL: https://docs.npmjs.com/about-npm.

[68] ——, *Paquetes npm*. URL: https://www.npmjs.com/.

[69] C. C. (Organization), *Copyleft: manual de uso*, Traficantes de Sueños, 2006.

[70] Overleaf, *Overleaf*. URL: https://es.overleaf.com/.

[71] pandas, *pandas*. URL: https://github.com/pandas-dev/pandas/.

[72] Portwiture, *Portwiture*. URL: http://portwiture.com/.

[73] PyCharm, *PyCharm*. URL: https://www.jetbrains.com/es-es/pycharm/.

[74] PyMongo, *PyMongo*. URL: https://github.com/mongodb/mongo-python-driver.

[75] Pytesseract, *Pytesseract*. URL: https://github.com/madmaze/pytesseract.

[76] Python, *Python*. URL: https://www.python.org/.

[77] python dotenv, *python-dotenv*. URL: https://github.com/theskumar/python-dotenv.

[78] D. Qiu and R. Srikant, *Modeling and performance analysis of bittorrent-like peer-to-peer networks*, SIGCOMM Comput. Commun. Rev., 34 (2004), p. 367–378.

[79] React, *React*. URL: https://es.reactjs.org/.

[80] ——, *React Developer Tools*. URL: https://es.reactjs.org/blog/2019/08/15/new-react-devtools.html.

[81] REACT WORDCLOUD, *react-wordcloud*. URL: https://react-wordcloud.netlify.app/.

[82] RECHARTS.JS, *Recharts.js*. URL: https://recharts.org/en-US/api.

[83] REMIX, *remix*. URL: https://remix-ide.readthedocs.io/en/latest/.

[84] RINKEBY, *Rinkeby*. URL: https://rinkeby.etherscan.io/.

[85] R. ROGERS, *Debanalizing twitter: The transformation of an object of study*, in Proceedings of the 5th Annual ACM Web Science Conference, WebSci '13, New York, NY, USA, 2013, Association for Computing Machinery, p. 356–365.

[86] ROPSTEN, *Ropsten*. URL: https://ropsten.etherscan.io/.

[87] S. S, *Statistical vs rule based machine translation; a case study on indian language perspective*, 2017.

[88] E. SARAVIA, *Fundamentals of nlp - chapter 1 - tokenization, lemmatization, stemming, and sentence segmentation*. URL: https://dair.ai/notebooks/nlp/2020/03/19/nlp_basics_tokenization_segmentation.html.

[89] K. SHARMA, S. SEO, C. MENG, S. RAMBHATLA, AND Y. LIU, *Covid-19 on social media: Analyzing misinformation in twitter conversations*, (2020).

[90] SMART, *Iniciativa Smart*. URL: https://smart-ib.coop/.

[91] SNSCRAPE, *snscrape*. URL: https://github.com/JustAnotherArchivist/snscrape.

[92] SOLIDITY, *Solidity*. URL: https://docs.soliditylang.org/.

[93] SPACY, *spaCy*. URL: https://github.com/explosion/spaCy.

[94] STANZA, *Stanza*. URL: https://github.com/stanfordnlp/stanza.

[95] STATISTICS, *Statistics*. URL: https://github.com/stanfordnlp/stanza.

[96] D. STOJANOVSKI, I. DIMITROVSKI, AND G. MADJAROV, *Tweetviz: Twitter data visualization*, (2014). URL: https://www.csc2.ncsu.edu/faculty/healey/tweet_viz/tweet_app/.

[97] SUBPROCESS, *Subprocess*. URL: https://docs.python.org/3/library/subprocess.html.

[98] TEAMGANTT, *teamgantt*. URL: https://app.teamgantt.com/.

[99] TEXTBLOB, *Textblob*. URL: https://textblob.readthedocs.io/en/dev/.

[100] Threading, *Threading*. URL: https://docs.python.org/3/library/threading.html.

[101] C. Tim, *Plantilla paper dashboard react*. URL: https://www.creative-tim.com/product/paper-dashboard-react.

[102] Time, *Time*. URL: https://docs.python.org/3/library/time.html.

[103] TrackMyHashtag, *Trackmyhashtag*. URL:https://www.trackmyhashtag.com/.

[104] M. Trigás Gallego, *Metodologia scrum*, (2012).

[105] Twitter, *Brand Twitter guidelines*. URL: https://about.twitter.com/en/who-we-are/brand-toolkit.

[106] ——, *Twitter API (docs)*. URL: https://developer.twitter.com/en/docs.

[107] ——, *Twitter for websites (docs)*. URL: https://developer.twitter.com/en/docs/twitter-for-websites.

[108] twitter-api v2, *twitter-api-v2*. URL: https://github.com/plhery/node-twitter-api-v2.

[109] Twitter hashtag #FightFor15, *Twitter hashtag #FightFor15*. URL: https://twitter.com/search?q=%23FightFor15&src=typed_query.

[110] Twitter hashtag #Malpasopagaya, *Twitter hashtag #Malpasopagaya*. URL: https://twitter.com/hashtag/Malpasopagaya?src=hashtag_click.

[111] Twitter @jobsmierda, *Twitter @jobsmierda*. URL: https://twitter.com/jobsmierda?lang=es.

[112] M. UI, *Material UI*. URL: https://mui.com/.

[113] H. Ullah, Z. Ullah, S. Maqsood, and A. Hafeez, *Web scraper revealing trends of target products and new insights in online shopping websites*, International Journal of Advanced Computer Science and Applications, 9 (2018), pp. 427–432.

[114] S. Vieweg, *Microblogged contributions to the emergency arena: Discovery, interpretation and implications*, (2010).

[115] Vincent Pasquier, Alex J. Wood, *The power of social media as a labour campaigning tool: lessons from our walmart and the fight for 15 — ETUI, the european trade union institute*, 2020.

[116] Y. Wang, M. McKee, A. Torbica, and D. Stuckler, *Systematic literature review on the spread of health-related misinformation on social media*, Social Science Medicine, 240 (2019), p. 112552.

[117] Web3, *Web3*. URL: https://web3js.readthedocs.io/en/v1.7.0/.

[118] Z. West, *Stop words: Removing words without meaning*. URL: https://www.alpharithms.com/stop-words-lists-removal-195521/.

[119] A. Williams, *User-centered design, activity-centered design, and goal-directed design: a review of three methods for designing web applications*, in Proceedings of the 27th ACM international conference on Design of communication, 2009, pp. 1–8.

[120] B. Wireframes, *Balsamiq wireframes*. URL: https://balsamiq.com/wireframes/.

[121] A. Wood, *Networks of injustice and worker mobilisation at walmart*, Industrial Relations Journal, 46 (2015).

[122] B. Zhao, *Web scraping*, Encyclopedia of big data, (2017), pp. 1–3.