Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Aplicación web de predicción de rutas con modelos de aprendizaje profundo

Estudante: Gabriel Sánchez Campello
Dirección: Laura M. Castro Souto
Dirección: Carlos Fernández Lozano

A Coruña, setembro de 2021.

Dedicado a todas las personas que creyeron en mi y me apoyaron siempre.

Agradecimientos

Agradezco este trabajo a mi familia que me ha apoyado durante toda mi vida.

A mi pareja que me ha acompañado durante toda mi carrera universitaria y siempre ha estado ahí para mí.

Y por último, a mis profesores por acompañarme y guiarme a lo largo de este proyecto a pesar de mi inconstancia.

Resumen

En la actualidad, un gran porcentaje de la población tiene coche propio y lo usa en su día a día. Este uso continuo y en grandes cantidades provocan varios problemas entre los que destacan la contaminación atmosférica; los atascos, los cuales solo incrementan la contaminación; y la expulsión de gases nocivos para la salud, entre los que destaca el óxido nitroso. Desde las administraciones públicas se han tomado medidas para combatir este aumento de la contaminación como por ejemplo: fomentar el uso de servicios de transporte público con diferentes incentivos, limitaciones de velocidad, reformas de aceras y mejoras en los carriles bici... Pero siempre puede hacerse más.

Este trabajo presenta una aplicación web en la que una persona podrá buscar una ruta entre dos puntos en una fecha y hora futura y obtener las posibles rutas y la contaminación media de cada una. De esta manera, si el usuario realiza el viaje andando o en bici, evitará las zonas con mayor riesgo para su salud; y si realiza el viaje en coche podrá saber qué rutas tienen más atascos ya que es previsible que cuánto mayor atasco, mayor contaminación. Al mismo tiempo, se estará reduciendo la contaminación total, ya que se circulará de forma más fluida evitando los atascos.

Abstract

Nowadays, a large percentage of the population owns a car and uses it on a daily basis. This continuous and large-scale use of cars causes several problems, including air pollution, traffic jams, which only increase pollution, and the release of gases that are harmful to health, including nitrous oxide. Public administrations have taken measures to combat this increase in pollution, such as encouraging the use of public transport services with different incentives, speed limits, pavement reforms and improvements to bicycle lanes... But there is always more that can be done.

This paper presents a web application in which a person can search for a route between two points at a future date and time and obtain the possible routes and the average pollution of each one. In this way, if the user makes the journey on foot or by bicycle, he or she will avoid the areas with the greatest risk to health; and if he or she makes the journey by car, he or she will know which routes have the most traffic jams, since it is foreseeable that the more traffic jams, the greater the pollution. At the same time, you will be reducing overall pollution, as you will be driving more smoothly and avoiding traffic jams.

Palabras clave:

- Contaminación atmosférica
- Polución
- Enrutamiento
- Predicción de contaminación
- Desarrollo web
- Gestión de proyectos
- Django

Keywords:

- Air Pollution
- Pollution
- Routing
- Pollution prediction
- Web development
- Project management
- Django

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En este capítulo se explicarán las circunstancias que nos llevaron al desarrollo de este proyecto así como los objetivos que se quieren alcanzar con él.

1.1 Contexto

La idea original del proyecto surge a partir de la búsqueda de algo más que un simple desarrollo software, se buscaba que tuviera un objetivo ético y cívico que pudiera, de algún modo, ayudar a la sociedad. Por lo que para encontrar una idea a desarrollar, se han investigado diversos problemas mundiales que preocupan a la humanidad, como por ejemplo las desigualdades, las crisis financieras, etc. Entre ellas se ha escogido la contaminación atmosférica y los problemas de salud en grandes ciudades debido a que ambas comparten un nexo en común: la contaminación producida por los vehículos.

La contaminación en el aire tanto en interiores como en exteriores se cobra 5 millones de muertes al año a nivel mundial, un 9% del total. Esto la convierte en la cuarta causa de muerte en el mundo después de fumar, la presión arterial alta y la diabetes. En las dos últimas décadas se ha conseguido reducir la contaminación en interiores, pero la contaminación en exteriores sigue fluctuando en los mismos valores y no presenta una mejoría aceptable.

Esta falta de mejora se debe a la contaminación producida entre otras cosas por los vehículos, los cuales contribuyen en gran medida debido a los gases nocivos que expulsan por el tubo de escape. Dependiendo de la ciudad se calcula que los vehículos pueden llegar a producir el 70% de la contaminación en el aire, sin tener en cuenta consumibles tóxicos como aceite, baterías, frenos…

Entre el año 2000 y 2009 en España se produjeron 92.672 muertes provocadas por la contaminación del aire. Solamente en Madrid hubo 11.042 muertes prematuras por óxido de nitrógeno (NOx) y 5.079 por partículas en suspensión PM10 y PM2,5.

Para este proyecto, se usará la ciudad de Madrid como ciudad pionera porque es una de

las mayores ciudades de España, con una relación de 54 coches por cada 100 personas aproximadamente. Además, ofrece de forma totalmente pública el conjunto de datos necesarios para desarrollar el proyecto [1, 2, 3, 4].

Como se ha comentado anteriormente los gases expulsados por los vehículos son causantes del aumento de la contaminación atmosférica por lo que hay una posible relación directamente proporcional entre la cantidad de coches en una zona y la cantidad de contaminación en el aire, pudiendo así indicar dónde están las zonas con mayor cantidad de coches y las zonas con mayores riesgos para la salud.

Sabiendo esto se desarrollará, por un lado, un modelo de aprendizaje profundo para cada una de las estaciones de calidad de aire de Madrid, con el que predecir los niveles de contaminación que habrá en las próximas horas para cada zona de influencia de las estaciones de calidad. Por otro lado, se desarrollará una aplicación web en la que se podrá buscar diferentes rutas entre dos puntos. La aplicación intentará ofrecer por lo menos dos posibles rutas, en las que se indicará el tiempo estimado de viaje y la contaminación media que atraviesa cada una; y recomendará utilizar siempre la ruta con menor contaminación, es decir, la más saludable en caso de realizar el trayecto andando o en bici; o la ruta más rápida en caso de realizar el viaje en coche, ya que se evitarían las zonas con atascos. Para poder obtener las rutas se necesitará indicar la ubicación de origen y destino del viaje así como la fecha y hora de salida y el medio de transporte en el que se va a realizar el viaje. La fecha y hora de salida es necesaria, ya que dependiendo de la hora a la que se vaya a realizar el viaje puede haber más o menos contaminación en las diferentes zonas. Para saber la contaminación en diferentes horas, y poder calcular la contaminación en las diferentes rutas la aplicación web hará uso de la predicción realizada por el modelo de aprendizaje profundo.

1.2 Objetivos

El objetivo abstracto y a gran escala de este proyecto es ofrecer a los usuarios que quieran viajar a una determinada fecha y hora la ruta con menor contaminación posible para que, como viandantes o ciclistas, puedan evitar las rutas que pasan por zonas de mayor contaminación, reduciendo así el daño a su salud en la medida de lo posible. Del mismo modo, como conductores, también será posible evitar las zonas congestionadas con vehículos mejorando así la circulación y reduciendo la cantidad de gases nocivos que expulsa el vehículo.

Siendo un poco más concretos a continuación se relacionan los principales objetivos técnicos del proyecto:

- Introducción a las redes neuronales: tipos de aprendizaje, clasificaciones...
- Estudio teórico de redes neuronales.

- Analizar la serie temporal a varios niveles para minimizar el error en la medida de lo posible en la predicción.
- Implementar los modelos de aprendizaje en el lenguaje escogido.
- Desarrollar un proceso que nos permita comparar los resultados de diferentes modelos.
- Desarrollar la aplicación que haga uso de las predicciones del modelo desarrollado.

Estos objetivos coinciden con varias de las competencias de Ingeniería del Software y Computación. Por ejemplo:

- Identificar y analizar un problema y diseñar, desarrollar, implementar, verificar, mantener y documentar la solución software escogida.
- Conocer los fundamentos, paradigmas y técnicas de los sistemas inteligentes para analizar, diseñar y construir la solución software.
- Conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo la extracción automática de información y conocimiento a partir de los volúmenes de datos proporcionados.

El proyecto coincide con estas competencias ya que se ha identificado un problema de salud estrechamente ligado con la contaminación ambiental. Analizando el problema se ha detectado un nexo común que es la contaminación que expulsan los vehículos y para paliar el problema se ha diseñado, desarrollado e implementado una solución software. Además de que se ha documentado todo el proceso.

Para obtener una solución software que ayude con el problema se ha necesitado realizar una investigación sobre las diferentes técnicas y paradigmas de los sistemas inteligentes. También se ha demostrado saber desarrollar estas técnicas de aprendizaje además de construir la solución software que explote el resultado de las predicciones.

Capítulo 2

Herramientas y Tecnologías

En este capítulo se explicarán algunas de las muy diversas herramientas y tecnologías disponibles para la realización de este proyecto y se justificarán las decisiones tomadas al escoger entre unas y otras.

2.1 Tecnologías de mapas interactivos

En la aplicación web se hará uso de mapas para mostrar las rutas y las contaminaciones en cada franja horaria. Para ello, existen múltiples tecnologías que permiten trabajar con mapas y mostrarlos en el lado cliente.

En esta sección se comentará las ventajas y desventajas de algunas de estas tecnologías y el motivo por el que se ha escogido finalmente una de ellas.

2.1.1 *Leaflet*

Leaflet es una librería de JavaScript de código abierto para mapas interactivos aptos para dispositivos móviles y páginas web [5]. Fue lanzado en 2011 por el desarrollador Vladimir Agafonkin y ha ido creciendo hasta convertirse en uno de los líderes de su ámbito.

Presenta bastantes ventajas entre las que destacan su diseño orientado a ser una biblioteca fácil de usar y liviana. También presenta una gran comunidad que, unido a ser una herramienta OpenSource, proporciona una documentación actualizada y con actualizaciones cada medio año de media.

Pero también presenta varias desventajas. Una de las más críticas es que para conseguir mayor flexibilidad y potencia es necesario hacer uso de complementos. Esta desventaja es una consecuencia directa de su mayor ventaja, es decir, que sea liviana y fácil de usar, ya que al diseñarse de esta manera se está recortando en potencia.

2.1.2 *OpenLayers*

OpenLayers es otra de las tecnologías líderes a la hora de trabajar con mapas en lado cliente. Es una librería de JavaSricpt que hace que sea fácil poner un mapa dinámico en cualquier página web. Puede mostrar mosaicos de mapas, datos vectoriales y marcadores cargados desde cualquier fuente. El objetivo con el que fue desarrollado es promover el uso de información geográfica de todo tipo [6].

Inicialmente fue desarrollado por MetaCarta en junio de 2006 y desde el noviembre del 2007 este proyecto forma parte de los proyectos de Open Source Geospatial Foundation. En la actualidad el desarrollo y el soporte corre a cargo de la comunidad de colaboradores.

Esta librería presenta muchas ventajas, ya que es una herramienta muy potente, ya que por ejemplo presenta: una mejor integración de proyecciones con Proj4js; visualización de mapas 3D; y el uso de WebGL para mostrar rápidamente grandes conjuntos de datos vectoriales.

Como desventaja esta herramienta presenta tantas funcionalidades que la curva de aprendizaje es muy exponencial y cabe destacar que debido a estas funcionalidades la librería es pesada. Al igual que con *Leaflet* una de sus mayores desventajas es consecuencia directa de su mayor ventaja, en este caso su potencia.

2.1.3 Tecnología Escogida

Para este proyecto se ha escogido la herramienta *Leaflet* debido a que es más sencillo de aprender, la documentación es más clara y, a través de los plugins, es posible añadir prácticamente cualquier funcionalidad (calcular rutas óptimas, incluir un mapa de localización, añadir capas base de cualquier proveedor, incluir buscadores...).

Si se necesitase desarrollar una aplicación con una gran cantidad de funcionalidades complejas relacionadas con los mapas, a pesar de lo citado anteriormente, sería mejor usar *OpenLayers*. Pero cómo para este proyecto solo se necesita mostrar el mapa con las posibles rutas entre dos puntos, sumado a lo anteriormente mencionado se ha escogido *Leaflet* [7, 8].

2.2 Herramientas de optimización de rutas

Para la optimización de rutas se usará una herramienta o API externa para obtener las posibles rutas entre el origen y destino escogido por el usuario. En esta sección se comentarán los pros y contras de las diversas herramientas y cuál es la elegida.

2.2.1 GoogleMaps

Esta API es mundialmente conocida y referente en su ámbito por lo que fue un claro candidato al inicio del proyecto.

Se lanzó oficialmente en febrero de 2005 solo para Estados Unidos y con el tiempo se fue liberando en el resto de países. Al comienzo, no se tenía esperanzas en el éxito del proyecto, pero con el tiempo se ha convertido en uno de los más usados.

Tiene una gran comunidad y una documentación detallada. También se actualiza con frecuencia y es bastante precisa. Cabe destacar la fiabilidad que aporta al estar apoyada por Google. La principal y mayor desventaja de esta herramienta es que es principalmente de pago, ofreciendo un uso gratuito muy limitado.

2.2.2 MapQuest

Esta herramienta está basada en *Leaflet* y funciona de manera muy similar a GoogleMaps cuando se trata de buscar ubicaciones y encontrar direcciones. Puede reconocer nuestra ubicación actual en función de nuestra dirección IP (si usa la web) o el GPS de nuestro teléfono (si usa un dispositivo móvil). Las instrucciones de ruta son muy precisas e incluyen los niveles de tráfico actuales [9].

Entre sus ventajas se encuentra que reenrutar la ruta de viaje planificada es muy fácil y las instrucciones incluyen los costos estimados de combustible.

Como desventaja el planificador de rutas tiene menos funciones que GoogleMaps y no se actualiza con tanta frecuencia. Otra clara desventaja es que también es de pago.

2.2.3 Herramienta Escogida

Inicialmente se decide MapQuest debido a que su versión gratuita permitía más peticiones que la API de GoogleMaps, pero tras un par de usos se descubre que MapQuest no trabaja muy bien las direcciones españolas y su API devuelve las rutas en un formato difícil de trabajar. Además, se trabaja con el soporte de la herramienta por unas funcionalidades y no se consigue respuesta favorable, mientras que en la API de GoogleMaps se tiene una gran comunidad y mucha más documentación sobre como explotar la API.

Debido a esto, la herramienta seleccionada para el desarrollo del proyecto ha sido GoogleMaps.

2.3 Lenguajes de desarrollo

A día de hoy existe una gran cantidad de lenguajes de programación diferentes, cada uno con sus ventajas e inconvenientes. Existen incluso lenguajes de programación basados en otros que le añaden una capa por encima al lenguaje original creando así un nuevo lenguaje. Por lo que es difícil escoger entre ellos.

En esta sección se comentarán las características de varios de los más utilizados y los elegidos para la realización de este proyecto.

2.3.1 JavaScript

JavaScript es un lenguaje ligero e interpretado, orientado a objetos con funciones de primera clase, multi-paradigma, basado en prototipos y dinámico. Es mundialmente conocido como el lenguaje de script para páginas web, pero también se usa en muchos entornos sin navegador [10].

Fue desarrollado por Netscape durante la Guerra de Navegadores en 1995 como lenguaje web del lado cliente. Al principio no tuvo mucho éxito, pero con la llegada de AJAX se popularizó llegando a ser uno de los lenguajes de programación más usados en el ámbito web.

JavaScript no debe ser confundido con el lenguaje de programación Java. Ambos ”Java” y ”JavaScript” son marcas registradas de Oracle en Estados Unidos y otros países. Sin embargo, los dos lenguajes de programación tienen muchas diferencias en la sintaxis, semántica y usos.

Entre sus ventajas destaca su rapidez y los múltiples efectos visuales. También desataca su versatilidad para desarrollar páginas dinámicas y al ser multiplataforma puede ser ejecutado de manera híbrida en cualquier sistema operativo móvil [11].

Pero también presenta sus contras, como por ejemplo el código puede ser leído por cualquier usuario; no es compatible en todos los navegadores de manera uniforme y una de las más importantes, los usuarios tienen la opción de desactivar JavaScript desde su navegador.

2.3.2 TypeScript

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft desde mediados de 2012.

Es un superconjunto de JavaScript, que enfocándose en lo esencial añade tipado estático y objetos basados en clases.

Como ventajas de este lenguaje cabe destacar que utiliza la misma sintaxis de JavaScript y las características de EcmaScript. Además de que compila su código a código JavaScript el cual es entendido en todos los navegadores web. Al ser compilado evita errores en la ejecución. También hace que sea más sencilla la programación orientada a objetos.

Como desventaja principal se tiene el aumento en la complejidad de nuestros proyectos con todo lo que ello acarrea como por ejemplo más tiempo y esfuerzo lo cual repercute en el coste.

2.3.3 Java

El lenguaje Java fue desarrollado en sus inicios por James Gosling, en el año 1991. Inicialmente era conocido como Oak o Green y la primera versión del lenguaje fue publicada por Sun Microsystems en 1995. Es en el año 1996 con la versión del lenguaje JDK 1.0.2, cuando pasa a llamarse Java. Su popularidad fue en aumento hasta convertirse en el lenguaje de

programación más usado.

Java como lenguaje de programación de alto nivel tiene la característica de ser al mismo tiempo compilado e interpretado. Esto se consigue debido a que el compilador convierte el código fuente del programa en un código intermedio llamado bytecode, el cual es independiente de la plataforma de trabajo, y que es ejecutado por el intérprete de Java que forma parte de la Máquina Virtual de Java.

Como ventajas se tiene que es orientado a objetos, distribuido y dinámico, robusto, seguro, multitarea y portable.

Por otro lado una de las principales contras es su bajo rendimiento.

Al igual que cualquier lenguaje de alto nivel, el rendimiento se ve afectado por la compilación y abstracción de la máquina virtual, pero en Java hay que añadir la presencia del recolector de basura. A pesar de ser útil, suele producir problemas de rendimiento significativos si toma más del 20% del tiempo de CPU. Además, la configuración de almacenamiento en caché de manera incorrecta puede causar un uso excesivo de memoria y recolección de basura y aunque estos problemas se pueden prevenir con una planificación hábil, se van sumando y pueden causar diferentes volúmenes de daño [12, 13].

2.3.4 Python

Python es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo.

El creador del lenguaje es un europeo llamado Guido Van Rossum ayudado y motivado por su experiencia en la creación de otro lenguaje llamado ABC. Su objetivo al crear Python era cubrir la necesidad de un lenguaje orientado a objetos de sencillo uso que sirviese para tratar diversas tareas dentro de la programación que habitualmente se hacía en Unix usando C.

Una de las ventajas que presenta es lo rápido y sencillo que es de leer su código debido a que es obligatorio las tabulaciones para programar en Python. Otras de las ventajas es que admite múltiples sistemas y plataformas, presenta una gran cantidad de frameworks que proporcionan mucha flexibilidad, permite escalar con facilidad y es multiparadigma.

Por otro lado sus desventajas son que al ser interpretado se pierde velocidad, no tiene mucha presencia en las plataformas móviles y no es bueno para los proyectos que requieran multiprocesadores [14].

2.3.5 Lenguajes de Programación elegidos

Para el lado cliente se ha escogido JavaScript debido a que en este proyecto no se espera tener que realizar acciones complejas en el lado del cliente, solo se realizarán peticiones sencillas al lado del servidor por lo que usar otro tipo de lenguaje como TypeScript solo se conseguiría incrementar la complejidad del proyecto innecesariamente incumpliendo el principio de diseño KISS. Esto, sumado a las ventajas mencionadas anteriormente de JavaScript, como por ejemplo el ser multiplataforma, hace que sea el lenguaje perfecto para nuestro proyecto.

Para el lado servidor se ha escogido Python por diversos motivos. Uno de los principales es que es uno de los mejores lenguajes para desarrollar proyectos con inteligencia artificial debido a que dispone de librerías cómo *NumPy*, *scikit-learn*, *Keras* y *Matplotlib*. Otro de los motivos es la existencia de un framework geográfico de clase mundial llamado *GeoDjango*, el cual tiene como objetivo facilitar en la medida de lo posible la creación de aplicaciones web SIG y aprovechar el poder de los datos habilitados espacialmente. Es decir, *GeoDjango* se centra en simplificar lo máximo posible la creación de aplicaciones web geográficas, así como servicios basados en la ubicación.

Por estos motivos se ha decidido escoger Python como el lenguaje para el lado servidor.

2.4 Librerías especializadas

Como se comentaba en el apartado 2.3, al escoger Python como herramienta de desarrollo para el backend y el modelo de aprendizaje se dispone de varias librerías muy útiles especializadas en los temas en cuestión. Por lo que en esta sección se comentará brevemente varias de las librerías más importantes de las que se harán uso para desarrollar este proyecto.

2.4.1 *Tensorflow*

Actualmente es la librería de *Deep Learning* más utilizada. Esta librería fue desarrollada por Google para construir y entrenar redes neuronales para sus proyectos y acabó liberándola como software libre en 2015 [15, 16].

Esta librería nos permite realizar computación matemática, ejecutando de manera rápida y eficiente gráficos de flujo. Los nodos en las gráficas representan operaciones matemáticas, mientras que las entradas y salidas de las gráficas representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos.

La arquitectura en la que está basada permite implementar el cálculo sobre una o más CPU o GPU en equipos de escritorio, servidores…

2.4.2 *Keras*

Esta librería está centrada en la construcción de redes neuronales y está escrita en Python. Consiste en una abstracción, una API a alto nivel, que permite definir y entrenar casi todo tipo de modelos de redes neuronales sin tener que entrar en detalles técnicos y que, por lo tanto, sea más fácil de usar por usuarios nuevos [17].

Destaca por ser amigable para el usuario, gracias a una sintaxis homogénea y una interfaz sencilla y, además, es modular y ampliable.

Keras por sí sola no puede construir redes neuronales, necesita apoyarse en una segunda librería que las construya a bajo nivel. Actualmente es compatible con *CNTK*, *Theano*, y la anteriormente mencionada *Tensorflow*.

En *Keras* la estructura principal se conoce como *Sequential* y permite crear redes neuronales básicas. Esta estructura es una pila lineal de capas, en la que se pueden ir añadiendo una capa tras otra.

2.4.3 *Django* y *GeoDjango*

Django es una framework de alto nivel para el desarrollo de aplicaciones web seguras y mantenibles. Actualmente es de código abierto y está escrito en Python [18, 19].

Su objetivo es facilitar la creación de aplicaciones web complejas. Para ello se centra en maximizar la reutilización de código en la medida de lo posible, la conectividad, el desarrollo rápido y el principio *Don't Repeat Yourself* (DRY).

Este framework utiliza una versión ligeramente modificada del patrón de diseño Modelo-Vista-Controlador (MVC) al que llaman como Modelo-Vista-Plantilla (MVP o MVT en inglés). En este patrón la parte del Controlador la realiza el propio *Django*, y a cambio se tiene las plantillas que son archivos HTML en el que se crearán las páginas a mostrar.

GeoDjango [20] es un módulo para *Django* que añade funcionalidades a *Django* para facilitar la creación de aplicaciones web geográficas. Alguna de las características que aporta son: campos del modelo para geometrías OGC; extensiones para consultar y manipular datos espaciales...

Capítulo 3

Metodología y Planificación

En este capítulo se describirá como se ha planificado el desarrollo de este proyecto y una breve explicación de la metodología de desarrollo escogida.

3.1 Metodología

En esta sección se comentará la metodología SCRUM seleccionada para la realización del proyecto y el motivo de esta elección. Para ello, antes se debe hablar de las metodologías ágiles, término que engloba varias metodologías diferentes entre las que se encuentra la que se ha escogido.

3.1.1 Metodologías Ágiles

Este tipo de metodologías surgen a mediados de los años 1990 como parte de una reacción contra las metodologías de ”peso pesado”, muy estructuradas y estrictas, extraídas del modelo de desarrollo en cascada [21, 22].

Las metodologías ágiles deben cumplir con el Manifiesto Ágil, que no es más que una serie de 12 principios que se pueden agrupar en 4 puntos críticos [23]:

1. **Individuos e interacciones sobre procesos y herramientas:** aunque los procesos y las herramientas ayudan a terminar con éxito un proyecto, son las personas quienes asumen, participan e implementan un proyecto y determinan cuáles procesos y herramientas utilizar. Por lo tanto, en cualquier proyecto ágil el énfasis debe estar en las personas y en sus interacciones, en vez de los complicados procesos y herramientas.

2. **Software funcionando sobre documentación extensiva:** aunque la documentación es necesaria y útil para cualquier proyecto, muchos equipos se centran en la recopilación y el registro de descripciones cualitativas y cuantitativas de los entregables, cuando el valor real que se le entrega al cliente es en forma de un software funcional. Por lo tanto,

en vez de la documentación detallada, el enfoque ágil está en la entrega de un software de buen funcionamiento en incrementos a lo largo del ciclo de vida del producto.

3. **Colaboración con el cliente sobre negociación contractual:** tradicionalmente a los clientes se les ha visto como participantes externos, involucrados principalmente al inicio y al final del ciclo de vida del producto, y cuya relación se basaba en el contrato y su cumplimiento. Las metodologías ágiles creen en un enfoque de valor compartido, en el cual los clientes se consideran colaboradores. El equipo de desarrollo y el cliente trabajan unidos para evolucionar y desarrollar el producto.

4. **Responder ante el cambio sobre seguir un plan:** en el mercado actual, donde los requerimientos del cliente, las tecnologías disponibles y los patrones empresariales cambian constantemente, es fundamental abordar el desarrollo de productos de una forma adaptativa que permita la incorporación de cambios y rápidos ciclos de vida de desarrollo de producto, en vez de enfatizar el seguimiento de planes formados probablemente con información obsoleta.

Debido a que el proyecto estará muy definido por los cambios y continuas mejoras de sus características, se consideró imprescindible adoptar una metodología ágil capaz de manejar estos requisitos, concretamente la metodología SCRUM.

3.1.2 Metodología SCRUM

SCRUM tiene sus inicios alrededor del año 1986, y fue creado por Ikujiro Nonaka e Hirotaka Takeuchi. Más tarde, en 1995, crearon un conjunto de reglas, o conjunto de buenas prácticas, enfocadas al desarrollo de software y la bautizaron con el nombre de SCRUM.

Los 3 pilares principales en los que se apoya la metodología SCRUM son:

1. **Transparencia:** todos los implicados tienen conocimiento de qué ocurre en el proyecto y cómo ocurre. Esto hace que haya un entendimiento "común" del proyecto, una visión global.

2. **Inspección:** los miembros del equipo frecuentemente inspeccionan el progreso para detectar posibles problemas. La inspección no es un examen diario, sino una forma de saber que el trabajo fluye y que el equipo funciona de manera auto-organizada.

3. **Adaptación:** cuando hay algo que cambiar, el equipo se ajusta para conseguir el objetivo del Sprint. Esta es la clave para conseguir el éxito en proyectos complejos, donde los requisitos son cambiantes o poco definidos y en donde la adaptación, la innovación, la complejidad y flexibilidad son fundamentales.

En esta metodología las iteraciones son conocidas como Sprints, y suelen durar de mínimo 1 semana y 4 semanas de máximo. Esto se debe a que se centra en ajustar sus resultados y responder a las exigencias reales y exactas del cliente, revisando cada entregable, ya que los requerimientos van variando a corto plazo.

Al final de cada Sprint o iteración, se va revisando el trabajo validado de la anterior semana. En función de esto, se priorizan y planifican las actividades en las que se invertirá nuestros recursos en el siguiente Sprint.

Otra característica de SCRUM es que los equipos serán autoorganizados entre ellos y autodirigidos, es decir, ellos mismos se van a organizar las tareas y se van a dirigir.

Esta metodología presenta una serie de eventos o etapas por Sprint:

- **Planificación de Sprint:** se realiza el primer día de cada Sprint y se divide en dos tareas principales:
  - **Selección de Requisitos:** el cliente presenta al equipo la lista de requisitos priorizada del producto o proyecto y se escogen los más prioritarios que se prevé que podrán acabar en el Sprint.
  - **Planificación de Sprint:** el equipo elabora la lista de tareas necesarias para desarrollar los requisitos seleccionados del Sprint. Se estiman las tareas de forma conjunta y se reparten.
- **Reunión diaria (daily):** reuniones cortas de máximo 15 minutos dónde el equipo se reúne para contar que ha hecho cada uno, que va a hacer y posibles problemas/impedimentos encontrados.
- **Revisión de Sprint:** el equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, replanificando el proyecto.
- **Retrospectiva de Sprint:** el equipo analiza cómo ha sido su manera de trabajar y cuáles son los problemas que podrían impedirle progresar adecuadamente, mejorando de manera continua su productividad. El Facilitador se encargará de eliminar o escalar los obstáculos identificados que estén más allá del ámbito de acción del equipo.

En SCRUM existen 3 roles diferenciados:

- *Product Owner*: es el rol central del proyecto. En algunas ocasiones es quien representa al cliente o incluso es el propio cliente.

- *SCRUM Master*: es el líder del proyecto, gestiona las acciones en cada iteración y se encarga de mantener en contacto al equipo de trabajo con el cliente.

- Equipo de desarrollo: es el equipo de trabajo que lleva a cabo las acciones propias de cada iteración: programadores, diseñadores, arquitectos, personal de servicio, entre otros.

Debido a que se está desarrollando un TFG, se ha tenido que asumir los diferentes roles y la metodología se ha adaptado todo lo posible a las condiciones extraordinarias que acarrea ser un TFG.

Al comienzo de este proyecto no se tenían claros todos los requisitos por lo que se necesitaba una metodología que se adaptase bien a los constantes cambios y SCRUM es esa metodología. Nos permite una gestión regular de las expectativas del cliente gracias a la lista de requisitos priorizada y la demostración de los resultados de cada Sprint; capacidad de flexibilidad y adaptación con la replanificación al inicio de cada Sprint; y mitigación de riesgos, ya que el desarrollo es iterativo e incremental.

Esta metodología también nos aporta productividad y calidad con la mejora continua del equipo gracias a las reuniones de retrospectiva y las reuniones diarias en las que se fomenta la comunicación y trabajo en equipo.

3.2 Planificación

En esta sección se detallarán las historias de usuario y sus tareas, las estimaciones de tiempo, el reparto de historias de usuario en cada Sprint y el coste del proyecto.

3.2.1 Sprints

A continuación se detallará lo planificado para cada Sprint llevado a cabo en el proyecto. Es decir, las historias de usuario realizadas en cada Sprint con sus Story Points asignados y el desglose de tareas para cada historia de usuario con su estimación de tiempo correspondiente.

BrainStorming

Reunión antes del primer Sprint en la que se crearán tantas historias de usuario como sean posibles, teniendo en cuenta que en la planificación de cada Sprint estas historias pueden cambiar o añadirse nuevas.

Para su creación se ha usado la plantilla "Como [rol], quiero que [objetivo] para que [motivo]" siendo opcional el motivo, ya que si el motivo va implícito en el objetivo es redundante. También se ha tenido en cuenta que las historias de usuario deben ser independientes, negociables, valorables, estimables, pequeñas y verificables.

Una vez creadas las historias de usuario se le asignaron a cada una de ellas un Story Point para indicar su grado de complejidad. Para ello existen diferentes métodos por ejemplo el método Delphi o varias de sus derivaciones, como pueden ser Delphi de banda ancha y planning póker. Para este proyecto se ha usado el método de planning póker y se ha estimado las tareas con valores de la secuencia de Fibonacci.

El resultado de esta reunión fueron 14 historias de usuario y 113 story points.

1. Como usuario de la aplicación web, quiero obtener la contaminación en un instante de tiempo concreto del pasado o futuro. [8]
2. Como usuario de la aplicación web, quiero tener un registro de mis búsquedas. [5]
3. Como usuario de la aplicación web, quiero poder guarda y/o eliminar mis sitios favoritos. [5]
4. Como usuario de la aplicación web, quiero obtener la evolución de la contaminación en un intervalo de tiempo pasado o futuro. [8]
5. Como usuario de la aplicación web, quiero obtener rutas en un instante de tiempo futuro. [13]
6. Como usuario de la aplicación web, quiero poder iniciar sesión. [2]
7. Como usuario de la aplicación web, quiero poder registrarme. [2]
8. Como usuario de la aplicación web, quiero poder cerrar sesión. [1]
9. Como usuario de la aplicación web, quiero poder entrar a la aplicación desde el ordenador y el móvil. [8]
10. Como usuario de la aplicación web, quiero que la aplicación sea estéticamente agradable e intuitiva. [8]
11. Como administrador de la aplicación web, quiero poder editar/añadir/eliminar cuentas de usuario de la aplicación. [5]
12. Como administrador de la aplicación REST, quiero que el modelo de aprendizaje profundo se reentrene todas las noches y realice la predicción de las próximas 48h. [5]
13. Como administrador de la aplicación REST, quiero un modelo de aprendizaje profundo que prediga la contaminación de las próximas 48 horas. [40]
14. Como administrador de la aplicación REST, quiero que la aplicación actualice la base de datos con los valores más recientes de las diferentes estaciones. [3]

Primer Sprint

Para este primer Sprint se decide centrarse en la historia de usuario que desarrolla el modelo de aprendizaje profundo, ya que es la que mayor riesgo presenta y al mismo tiempo es en lo que se basa el éxito del proyecto.

La historia de usuario 13 se descompone inicialmente en las siguientes tareas con su correspondiente estimación:

- Creación y población de las BBDD con la información disponible de las estaciones y los datos de contaminación.(2H)
- Pre-procesamiento de los datos para el modelo y estudio. (16H)
- Desarrollo de modelo de aprendizaje profundo con una arquitectura de red neuronal simple. (2h)
- Optimización de hiperparámetros para el modelo con arquitectura simple. (32h)
- Implementar las funciones para comprobar la precisión de la predicción. (8h)

Se planifica que cada trabajador es capaz de realizar 6 horas diarias de trabajo técnico y sabiendo que en total hay 60 horas estimadas nos da un total de 10 jornadas necesarias, es decir, 2 semanas para realizar todas las tareas. Esto es justo lo que dura el Sprint por lo que se puede entregar las tareas en tiempo y forma.

Segundo Sprint

En el segundo Sprint se seguirá en la misma línea de trabajo del anterior. Se centrará en seguir mejorando la precisión en la predicción de la contaminación probando diferentes arquitecturas e hiperparámetros. Por ello se seguirá con la historia de usuario 13.

El desglose de tareas para este Sprint de la historia de usuario 13 es el siguiente:

- Desarrollo de modelo de aprendizaje profundo con arquitectura LSTM. (2h)
- Optimización de hiperparámetros para el modelo con arquitectura LSTM. (28h)
- Desarrollo de modelo de aprendizaje profundo con arquitectura GRU. (2h)
- Optimización de hiperparámetros para el modelo con arquitectura GRU. (28h)

Para este Sprint se planifica un total de 60 horas de trabajo técnico, lo cual partiendo de la base de que cada trabajador puede dedicar unas 6 horas diarias de trabajo técnico se traduce en 10 jornadas, es decir, otras dos semanas. Si las estimaciones no fallan, se puede acabar las tareas en tiempo y forma.

Tercer Sprint

En el tercer Sprint se creará la aplicación REST que encapsulará el modelo con la arquitectura y los hiperparámetros que han dado mejor resultado en Sprints anteriores y será desplegada. Para tener lista la aplicación REST se abarcarán las historias de usuario 12 y 14. Además, se empezará la historia de usuario 1.

El desglose de tareas para la historia de usuario 12 es el siguiente:

- Encapsulamiento del modelo en una aplicación REST. (10h)
- Preprocesamiento de los datos y re-entrenamiento de los modelos. (10h)
- Guardado de los modelos. (2h)
- Predicción y guardado de la predicción. (4h)

El desglose de tareas para la historia de usuario 14 es el siguiente:

- Automatización de la descarga de la contaminación. (10h)
- Procesamiento de la información y guardado en base de datos. (10h)

El desglose de tareas para la historia de usuario 1 es el siguiente:

- Implementar una función en la aplicación REST que devuelva los datos de cada estación. (4h)
- Serialización de los datos para la devolución a través de petición REST.(4h)

Para este Sprint se planifica un total de 54 horas de trabajo técnico. Además, en este Sprint, a pesar de no ser una historia de usuario se va a realizar la tarea de desplegar la aplicación REST a la cual se le estima 6 horas por lo que el número total de horas sería de 60 horas, que vendrían a ser las 10 jornadas habituales asegurando el poder entregar en tiempo y forma lo abarcado en este Sprint.

Cuarto Sprint

En este Sprint se va a empezar a desarrollar la aplicación web. Para ello se acabará la historia de usuario 1 consiguiendo comunicación entre la aplicación web y REST. Además se abarcará la historia de usuario 5, que se considera el segundo punto crítico de todo el proyecto.

Para la historia de usuario 1 se tienen las siguientes tareas:

- Desarrollo frontend de la página web en la que se pedirá el instante de tiempo a mostrar. (8h)

- Integración de la aplicación web con la aplicación REST. (2h)
- Desarollo backend para la obtención de la contaminación. (2h)
- Escribir pruebas automáticas que comprueben que se obtiene correctamente la contaminación. (2h)

Y la historia de usuario 5 se subdividirá en las siguientes tareas:

- Desarrollo frontend de la página web en la que se pedirá la información necesaria para obtener las rutas. (12h)
- Desarrollo backend para la obtención de la contaminación. (3h)
- Desarrollo backend para la obtención de las rutas. (10h)
- Desarrollo backend para calcular la contaminación de cada ruta. (12h)
- Desarrollo frontend para mostrar las diferentes rutas y contaminación de cada una. (4h)
- Escribir pruebas automáticas que comprueben que se obtiene correctamente la contaminación. (1h)
- Escribir pruebas automáticas que comprueben que se obtiene correctamente las rutas. (2h)
- Escribir pruebas automáticas que comprueben que se calcula correctamente la contaminación de las rutas. (2h)

En este Sprint se planifica un total de 60 horas de trabajo técnico, lo cual partiendo de la base de que cada trabajador puede dedicar unas 6 horas diarias de trabajo técnico se traduce en 10 jornadas, es decir, dos semanas. Si las estimaciones son correctas y no se sufre ninguna desviación, se puede entregar en tiempo y forma.

Quinto Sprint

En este Sprint se harán las historias de usuario: 2, 3, 4, 6, 7, 8.
Las tareas para la historia de usuario 7 son:

- Crear una tabla de usuarios en la base de datos. (1h)
- Implementar un sistema de registro de usuarios. (1h)
- Desarrollo de pruebas para el registro. (1h)

Las tareas para la historia de usuario 6 son:

- Desarrollo backend de un sistema de inicio de sesión. (1h)
- Desarrollo frontend de un sistema de login. (2h)
- Permitir que solo los usuarios autenticados tengan acceso a ciertas funcionalidades. (1h)

Las tareas para la historia de usuario 8 son:

- Implementar un botón de logout. (1h)

Las tareas para la historia de usuario 2 son:

- Crear una tabla de Búsquedas en la base de datos. (1h)
- Desarrollo backend para guardar un registro de la búsqueda de cada usuario registrado. (4h)
- Desarrollo de frontends para mostrar el historial de búsquedas. (4h)
- Desarrollo de pruebas automáticas. (2h)

Las tareas para la historia de usuario 3 son:

- Crear una tabla de Favoritos en la base de datos. (1h)
- Desarrollo frontend para guardar/eliminar lugares como favoritos. (6h)
- Desarrollo backend para guardar/eliminar lugares como favoritos. (6h)
- Desarrollo de pruebas automáticas. (4h)

Las tareas para la historia de usuario 4 son:

- Desarrollo frontend para mostrar la evolución de la contaminación. (10h)
- Desarrollo backend para obtener la contaminación en un período de tiempo. (5h)
- Desarrollo backend para mostrar la evolución de la contaminación en un mapa. (6h)
- Desarrollo de pruebas automáticas. (3h)

En este Sprint se planifica un total de 60 horas de trabajo técnico, lo cual partiendo de la base de que cada trabajador puede dedicar unas 6 horas diarias de trabajo técnico se traduce en 10 jornadas, es decir, dos semanas. Si no hay desviaciones en las predicciones todo se podría entregar en tiempo y forma.

Sexto Sprint

En este último Sprint se harán las historias de usuario restantes: 9, 10, 11. Las tareas para la historia de usuario 11 son:

- Personalización de las funciones administración para poder editar/añadir/eliminar cuentas de usuario. (12h)

Las tareas para la historia de usuario 9 son:

- Adaptar todas las pantallas para que se ajuste al tamaño de los móviles. (18h)

Las tareas para la historia de usuario 10 son:

- Aplicar estilos y prácticas de buen diseño a todas las pantallas. (24h)

Para este Sprint se planifica un total de 54 horas de trabajo técnico. Además, en este Sprint, a pesar de no ser una historia de usuario se va a realizar la tarea de desplegar la aplicación web, a la cual se le estima 6 horas por lo que el número total de horas sería de 60 horas, que vendrían a ser las 10 jornadas habituales. Asegurando el poder entregar en tiempo y forma lo abarcado en este Sprint.

3.2.2 Gestión del proyecto

Para trabajar con SCRUM se utilizará la herramienta Atlassian Jira. Esta herramienta está enfocada en la gestión de proyectos y es la líder en el desarrollo software para equipos ágiles. Nos permite planificar las historias de usuario, tareas, subtareas... supervisar el trabajo con completa visibilidad, generar informes etc.

Para este proyecto se han utilizado 3 tipos de *issues*:

- *Story*: este tipo de *issue* se identifica con las historias de usuario.
- *Task*: este tipo de *issue* se identifica con esas tareas que no pertenecen a ninguna historia de usuario, pero que se tienen que realizar también. Por ejemplo el despliegue de las aplicaciones en sus correspondientes servidores.
- *Subtask*: este tipo de *issue* se corresponde al resultado de dividir las historias de usuario en diferentes tareas para ir abarcándolas de uno en uno.

Por ejemplo en la Figura 3.1 se puede observar un ejemplo de como se vería una Story, en concreto la correspondiente a la historia de usuario 13.

Todas las historias de usuario y tareas mencionadas anteriormente se han creado en Atlassian Jira para trabajar con ellas y en la Figura 3.2 se puede observar lo que sería el *Backlog* inicial.

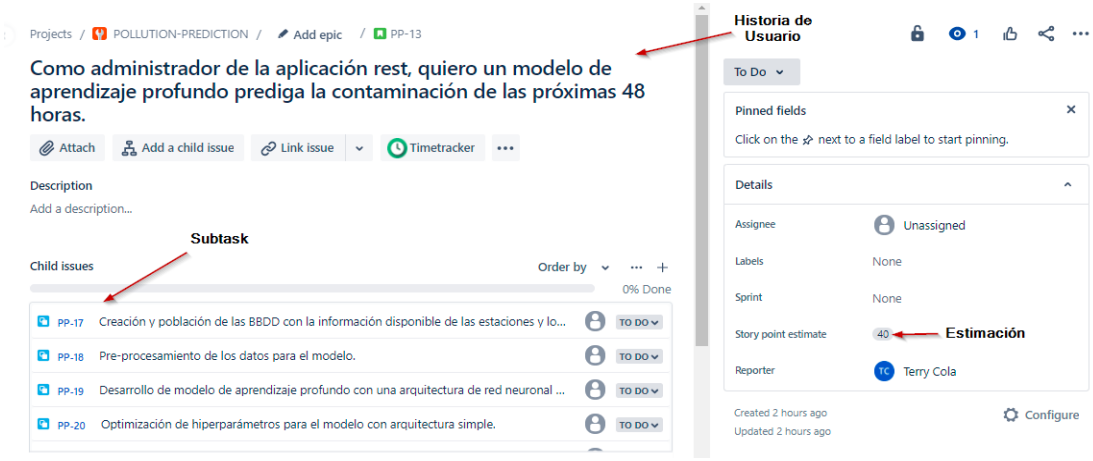


Figura 3.1: Ejemplo de Story.

Figura 3.2: *Backlog* Inicial.

Una vez creadas todas las historias y tareas planificadas se crea el Sprint, al cual se le asignará las *Story* o *Task* pertinentes. En este caso, al ser el primer Sprint se añadirá únicamente la *Story* 13 y sus *Subtask* correspondientes. Por último, se comenzará el Sprint indicándole la fecha de inicio y fin, además del objetivo.

Una vez creado y comenzado el Sprint se nos generará un tablero que por defecto nos muestra únicamente las *issues* de tipo *Story* y de tipo *Task*. Esto permite ver el avance a grandes rasgos, pero en este proyecto en concreto da muy poca información y visibilidad, ya que se ha desglosado gran parte del trabajo en *Subtask* por lo que para poder observar también las *Subtask* se tiene que agrupar por *Subtask* las *issues*. En la Figura 3.3 se puede observar un ejemplo de la poca información que aporta el tablero mostrando únicamente las *Story* y las *Task*. Mientras que en la Figura 3.4 se puede ver un ejemplo del tablero en medio del Sprint 1 con las *issues* agrupadas por *Subtask*.

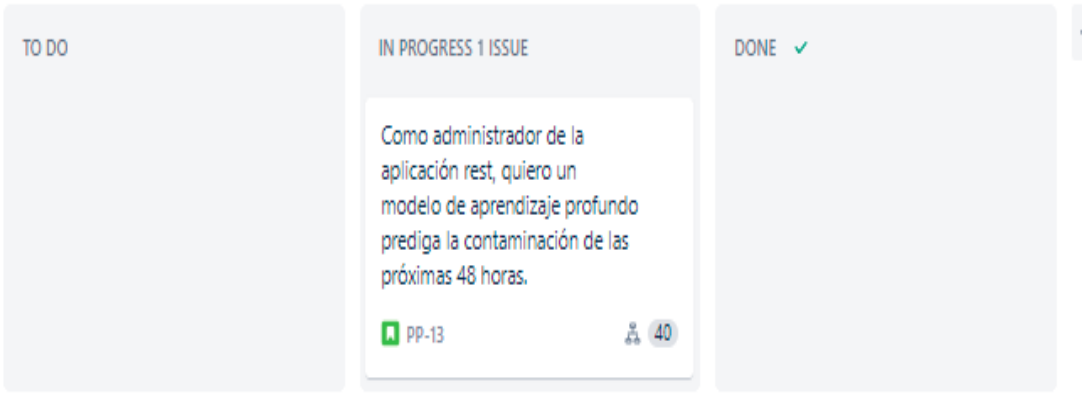


Figura 3.3: Tablero por defecto.

Las *issues Task* y *Subtask* tienen un campo llamado *Time tracking* que es para inputar las horas gastadas en ese *issue* y es con ese campo con lo que se hará un seguimiento y control del tiempo gastado en las diferentes tareas. Además, para poder crear informes y llevar un mejor control del tiempo gastado en cada *Subtask* y *Task* se hará uso de una App de Atlassian Jira, que viene a ser como un *plugin*, para poder generar diferentes informes del tiempo gastado en cada *issue*. En la figura 3.5 se puede observar un ejemplo de uno de los informes que se pueden generar con la App del Sprint 1.

3.3 Costes del proyecto

El tiempo total del proyecto ha sido de 480 horas, de las cuales 360 se han dedicado a tareas técnicas mientras que las otras 120 horas se han dedicado a reuniones, llamadas y/o tareas de

gestión y seguimiento, como por ejemplo, la reunión diaria o la retrospectiva.

En cuanto al coste del proyecto, lo habitual es que dependiendo de la tarea se requieran diferentes perfiles, cada uno con un sueldo medio diferente, y a pesar de haber desempeñado diferentes perfiles a lo largo del proyecto, al final los ha realizado la misma persona por lo que el coste/hora realmente es siempre el mismo. En España, el sueldo medio de un programador junior es de entre 17.000€ y 21.000€ brutos anuales dependiendo de la empresa y la localización geográfica. En concreto, en Galicia el sueldo medio del programador junior está en 17.067€ brutos, que vendría a ser 8,20€ a la hora [24, 25].

Por lo tanto, si el proyecto ha durado 480 horas y un programador junior gallego empieza cobrando unos 8,20€ la hora el proyecto ha tenido un coste total de 3.936€.

Cabe destacar que actualmente está usándose la capa gratuita de AWS y de la API de Google Maps. En caso de querer ampliar, dependiendo del tamaño del escalado el precio aumentaría, pero estos gastos se incluirían dentro del mantenimiento del proyecto.

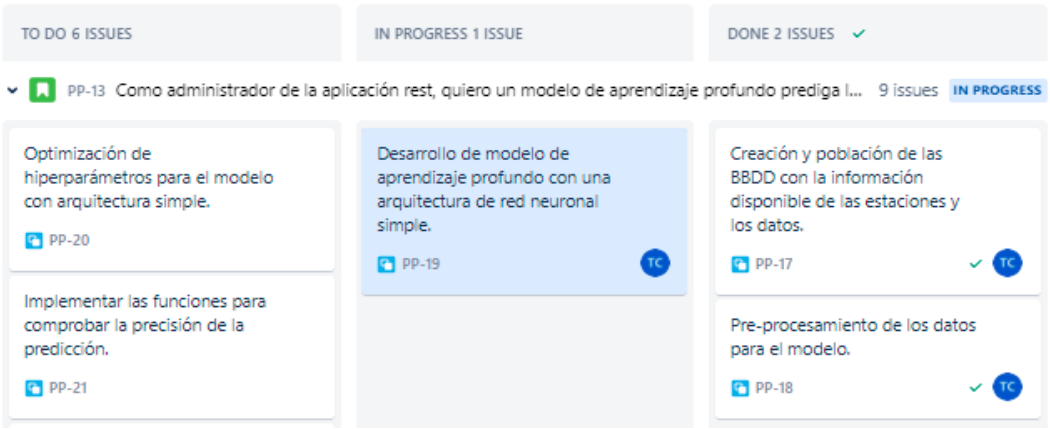
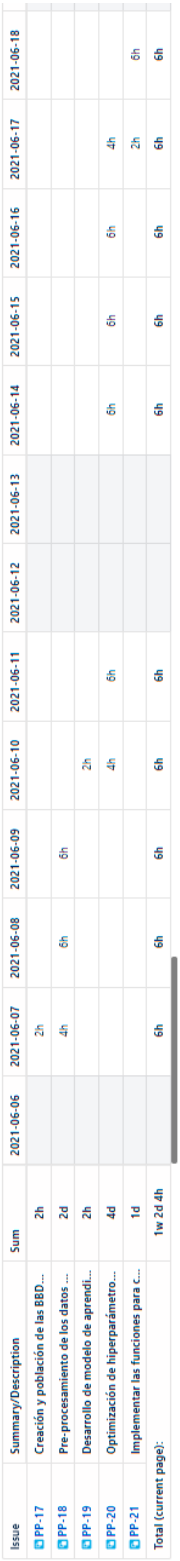


Figura 3.4: Tablero agrupado por *Subtask*.



Issue	Summary/Description	Sum	2021-06-06	2021-06-07	2021-06-08	2021-06-09	2021-06-10	2021-06-11	2021-06-12	2021-06-13	2021-06-14	2021-06-15	2021-06-16	2021-06-17	2021-06-18
PP-17	Creación y población de las BBD...	2h		2h											
PP-18	Pre-procesamiento de los datos ...	2d		4h	6h	6h									
PP-19	Desarrollo de modelo de aprendi...	2h					2h								
PP-20	Optimización de hiperparámetro...	4d					4h	6h			6h	6h	6h	4h	
PP-21	Implementar las funciones para c...	1d												2h	6h
Total (current page):		1w 2d 4h		6h	6h	6h	6h	6h			6h	6h	6h	6h	6h

Figura 3.5: Ejemplo de informe del Sprint 1.

Capítulo 4

Desarrollo

En este capítulo se comentará en detalle cada uno de los Sprints resumidos en el apartado 3.2.1 del capítulo Metodología y Planificación.

4.1 Primer Sprint

En este primer Sprint se tratará uno de los primeros pasos más importantes a la hora de desarrollar un modelo de aprendizaje: el análisis y preprocesamiento de los datos.

Para ello, antes de empezar el análisis se ha diseñado una base de datos en la que guardar todos los datos históricos disponibles. Como se puede apreciar en la Figura 4.1 para cada estación se guardará su histórico de datos correspondiente.

Debido a que el proyecto se desarrolla en diversos dispositivos, para poder utilizar los mismos datos sin pasar por retrabajos de crear y actualizar la base de datos en cada dispositivo se decide crear la base de datos utilizando los servicios de AWS. En concreto, se hace uso de *Amazon Relational Database Service* (RDS) para crear una base de datos en Postgres en la cual se insertan todos los datos disponibles.

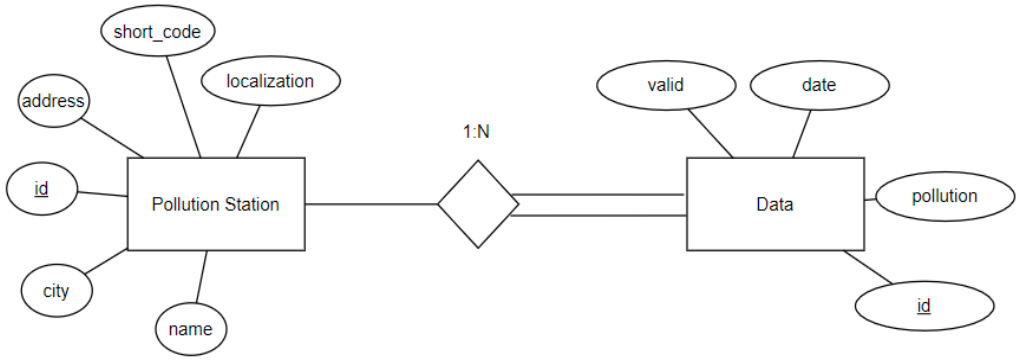


Figura 4.1: Modelo Entidad Relación.

4.1.1 Análisis de la serie temporal

Como se comentó anteriormente, este es uno de los pasos más importantes antes de empezar a desarrollar el modelo de aprendizaje. Esto se debe a que es necesario entender los datos para saber qué puede ser útil y poder hacer unos pronósticos significativos y precisos.

Los datos disponibles están registrados con una frecuencia horaria por lo que en un mes cada estación dispondrá de 720 muestras de media.

Para analizar la serie temporal primeramente se hace un estudio empírico en el que se grafica los datos para ver si se observa algún tipo de patrón de manera visual y luego se contrasta con un estudio científico.

El resultado final del estudio es que los datos presentan estacionalidad diaria, es decir, se repiten diariamente valores similares a las mismas horas. En concreto, se observan dos picos de contaminación: uno a las 9:00 AM y otro sobre las 12:00 AM; y dos claros valles de contaminación: a las 5:00 AM y a las 16:00 PM. En la Figura 4.2 se muestra la media de cada hora del día de los dos últimos años, y también se puede observar los períodos de actividad e inactividad mencionados anteriormente.

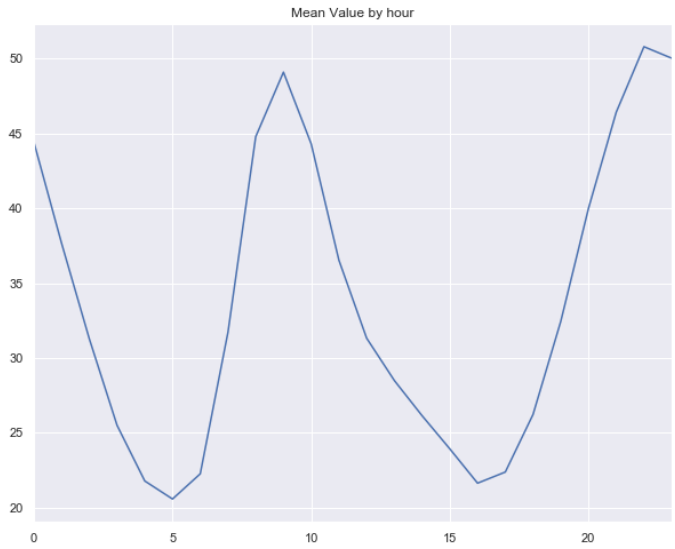


Figura 4.2: Estación de Vallecas: Media por horas.

Además, se ha descubierto que los valores más correlacionados son los dos inmediamente anteriores y el de hace 24 horas.

4.1.2 Preprocesado de la serie temporal

Una vez analizada la serie temporal se procede a procesar los datos.

El primer paso en el preprocesamiento es estandarizar los datos al intervalo [0,1] con el método `MinMaxScaler` proporcionado por el módulo *scikit-learn*.

Y se modifican los datos para poder convertir el problema a uno de tipo supervisado. Es decir, se separan los datos en un conjunto de datos de entrada y otro con el conjunto de salida deseado para cada conjunto de entrada.

En el Cuadro 4.1 se puede ver un ejemplo de cómo quedarían los datos con un retardo de 4. En este caso se estaría usando como entradas los valores de las columnas de (t-4) a t(-1) y cómo salida deseada el valor (t).

Fecha	var1(t-4)	var1(t-3)	var1(t-2)	var1(t-1)	var1(t)
2019-06-01 04:00:00	105.0	37.0	36.0	27.0	16.0
2019-06-01 05:00:00	37.0	36.0	27.0	16.0	15.0
2019-06-01 06:00:00	36.0	27.0	16.0	15.0	18.0

Cuadro 4.1: Transformación a serie supervisada con retraso = 4.

Teniendo en cuenta el resultado del análisis anterior (estacionalidad y autocorrelación) se puede intuir que utilizar un retraso $k = 1$ o $k = 24$ debería dar buenos resultados.

4.1.3 Desarrollo del modelo

Una vez acabado el preprocesamiento de los datos comienza el desarrollo de la primera aproximación de nuestro modelo. Para ello, se decide comenzar creando una red neuronal sencilla con *Keras* y su estructura principal *Sequential* mencionada en la subsección 2.4.2.

Durante el proceso de entrenamiento existen diversas configuraciones con las que se puede experimentar para obtener mejores resultados, conocidos como hiperparámetros. Algunos ejemplos de estos hiperparámetros son: el optimizador, la función de activación, la función de pérdida, el número de *epochs* ...

4.1.4 Optimización de hiperparámetros

Para este primer Sprint se ha jugado con el número de retardos, el número de neuronas en la capa oculta y el número de capas ocultas buscando el mejor resultado posible.

Al principio se empieza con valores aleatorios para cada hiperparámetro y se van variando, comprobando qué valores dan mejores resultados, hasta alcanzar un porcentaje de precisión aceptable para este primer sprint.

4.1.5 Resumen

En este Sprint se ha creado una base de datos con el histórico de cada estación, se ha analizado la serie temporal en busca de tendencias y estacionalidad y se ha desarrollado un modelo que prediga la contaminación.

Comparando los resultados predichos contra los reales en la Figura 4.3 se observa que las tendencias se predicen correctamente ya que los valores predichos suben cuando tienen que subir y bajan cuando tienen que bajar. El mayor problema son picos puntuales que superan el valor predicho por la red de aprendizaje profundo.

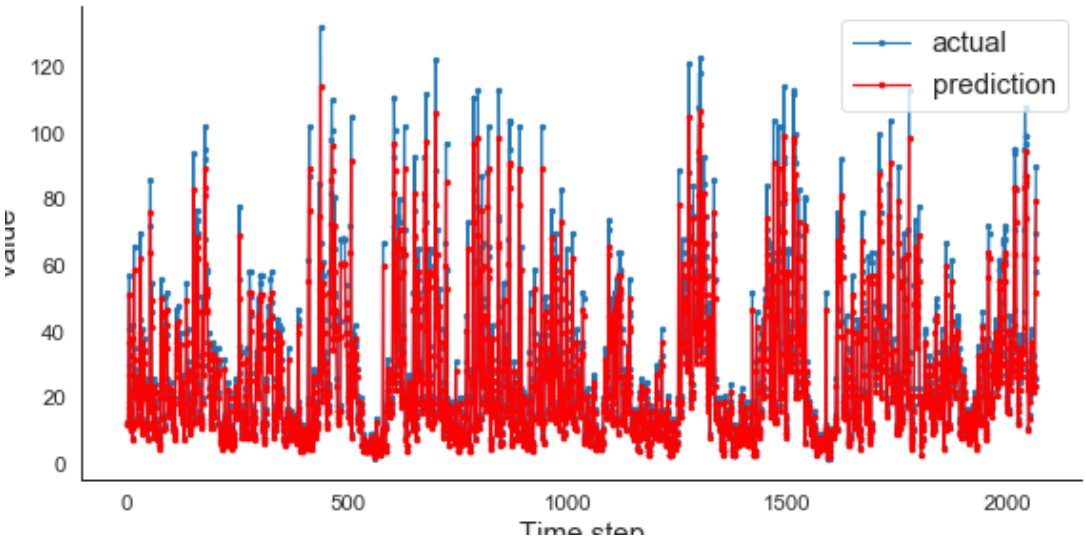


Figura 4.3: Datos predichos frente a datos reales.

4.2 Segundo Sprint

En este segundo Sprint se seguirá trabajando con el modelo de aprendizaje profundo para intentar mejorar todo lo posible la precisión de las predicciones.

Para ello se probarán otros tipos de arquitecturas más complejas, en concreto se probará con las arquitecturas LSTM y GRU. Además, se seguirá probando con diferentes valores de hiperparámetros en busca de los valores óptimos para estas nuevas arquitecturas.

4.2.1 LSTM y GRU

Para comenzar se crea una red neuronal LSTM y una red neuronal GRU lo cual, gracias a la librería *Keras*, es relativamente sencillo de hacer.

En primer lugar se parte de los hiperparámetros óptimos obtenidos en el sprint anterior.

En el Cuadro 4.2 se puede apreciar que con los hiperparámetros óptimos para una RNA simple tanto con LSTM como con GRU se obtiene una peor precisión y, aún por encima, la duración del entrenamiento es mayor.

Arquitectura	MASE	MAPE
RNA	0.186	12.062 %
LSTM	0.273	15.98 %
GRU	0.269	16.744 %

Cuadro 4.2: Comparación MAPE cambiando número de retardos.

Al igual que en el Sprint anterior se realiza una optimización de hiperparámetros en busca de los valores óptimos.

4.2.2 Resumen

Se optimizan los hiperparámetros para las aquitecturas LSTM y GRU y tras comparar los mejores resultados se decide utilizar la arquitectura LSTM. En la Figura 4.4 se puede observar como se vería la comparación de los datos reales versus los predichos con la arquitectura LSTM.

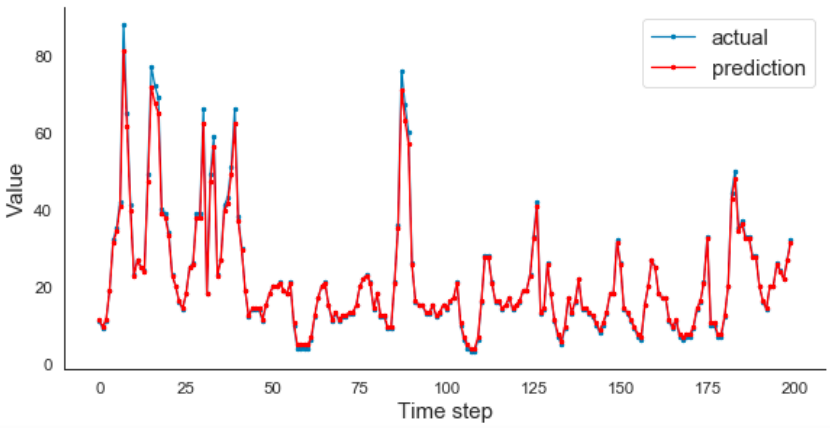


Figura 4.4: Gráfica comparativa de valores predichos frente a reales.

4.3 Tercer Sprint

Una vez definido el modelo el siguiente paso fue crear con *Django* una aplicación REST para explotar los datos predichos por el modelo.

La aplicación REST hará uso de la base de datos con la información histórica para entrenar los modelos correspondientes a cada estación y realizar una predicción de las siguientes 48 horas.

El primer paso ha sido añadir a la base de datos una tabla para los modelos que estará relacionada con las estaciones de calidad de aire, habiendo una entrada en la base de datos por cada estación con los valores de hiperparámetros como campos de la nueva tabla, con esto, si se añade alguna estación nueva y los hiperparámetros estudiados no producen un buen resultado se le puede hacer un nuevo estudio y modificar los hiperparámetros únicamente para el modelo de esa estación.

En la Figura 4.5 se muestra como quedaría el Diagrama UML de la aplicación REST. Como se puede observar, se tienen 3 modelos cada uno con sus propios *Fields* que en Python por defecto son siempre públicos. Esto se debe a que en Python se puede ”restringir” el acceso a los atributos al igual que en Java o C++, pero lo que lo diferencia es que, en Python, esta restricción es simbólica, ya que se puede obviar fácilmente y obtener de igual manera los valores de los atributos por lo que suelen dejarse públicos. Además, los modelos creados heredan varios métodos de la clase `Models` como `save` o `delete`, pero por limpieza en el UML se han mostrado únicamente los métodos creados a mayores para cada modelo.

Para poder enviar la información por REST se hará uso de la clase `Serializer` de *Django*. Esta clase permite convertir datos complejos como *QuerySets* en datos nativos de Python para facilitar su transformación a formatos como XML o JSON. Además, esta clase permite ”deserializar” para reconvertir de los datos nativos de Python a los datos complejos originales.

De las clases creadas, todos los atributos pueden serializarse sin ningún inconveniente excepto el atributo *localization* que es de tipo `PointField` propio de *GeoDjango* y no hay una función por defecto para serializarlo. Para este atributo se ha tenido que crear una función especial para serializarlo de manera personalizada.

```
1 #Ejemplo serializacion por defecto
2 name = serializers.CharField(max_length=100)
3
4 #Ejemplo serializacion personalizada
5 localization = GeometrySerializerMethodField()
6
7 #Función que utiliza "GeometrySerializerMethodField" para obtener
      el \emph{PointField}
8 def get_station_point(self, obj):
9     return Point(obj.station.localization.x,
```

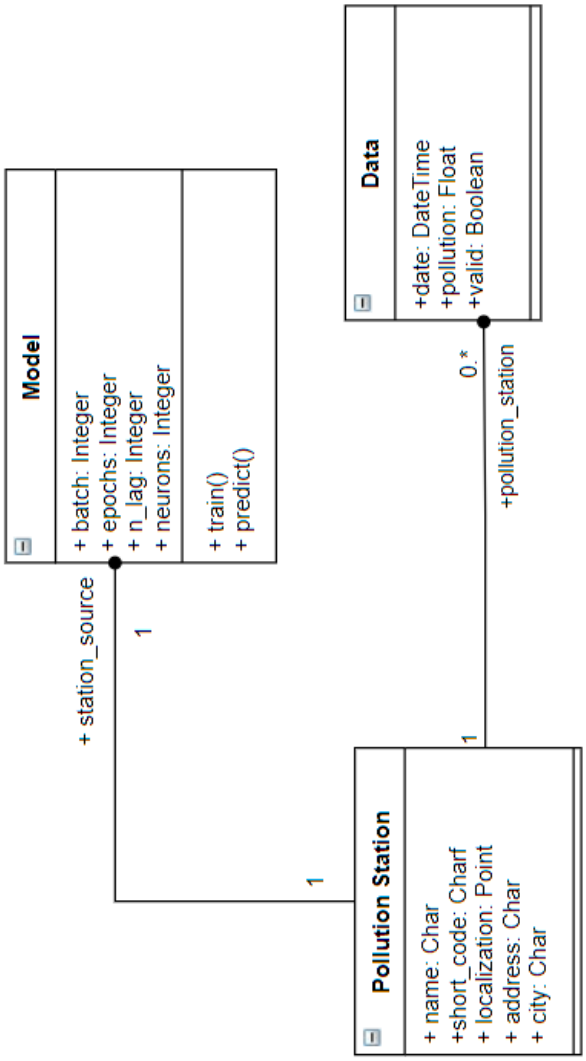


Figura 4.5: Diagrama UML.

```
    obj.station.localization.y)
```

Por último se crean 2 servicios REST para poder explotar la información desde la aplicación web:

- `getPollutionInterval`: recibe una petición POST con fecha de inicio del intervalo y la fecha fin y devuelve todos los datos que se tienen de cada estación en ese intervalo.
- `predictions`: recibe una petición GET que devuelve todos los datos predichos por los modelos.

Además, se crean dos servicios para el uso único de los administradores:

- `train`: recibe una petición POST para entrenar el modelo de una estación en concreto.
- `predict`: recibe una petición POST para que el modelo de una estación en concreto prediga las próximas 48 horas.

Una vez construida la aplicación REST, se automatiza la obtención de datos oficiales del día, se reentrenan los modelos asociados a cada estación y se realiza una predicción de las próximas 48 horas.

Para la automatización se hace uso de la librería *apscheduler*. En concreto, de *apscheduler* se importa `BackgroundScheduler` para crear el autómata y a continuación se le añade un *job* indicándole la periodicidad con la que ejecutarse y la función a la que llamar.

En nuestro caso, se ha creado un *job* que se lanza todas las noches a las 12:30 AM y ejecuta una función llamada `update_data_train_and_predict`. Esto en código quedaría de la siguiente forma:

```
1 def start():
2     scheduler = BackgroundScheduler()
3     scheduler.add_job(updaterAPI.update_data, 'cron', hour=12,
      minute=30)
4     scheduler.start()
```

La función `update_data` del `updaterAPI` se encarga de descargar los datos de contaminación del último día de la página oficial para añadirlos a la base de datos, así se mantienen los datos continuamente actualizados. Y, con estos nuevos datos, se re-entrenarán los modelos para que tengan en cuenta cualquier cambio de comportamiento reciente en las predicciones.

Una vez re-entrenado el modelo se guardarán los pesos de cada modelo para poder cargarlos en cualquier momento. Esto nos permite tener una versión del modelo reciente siempre disponible en caso de que haya algún problema al re-entrenar los modelos por la noche. También nos permite cargar un modelo en concreto y hacerle predecir las futuras contaminaciones en caso de necesitarlo.

Por último la función utilizará los modelos re-entrenados para predecir las próximas 48 horas. Se pone un período mayor a las 24 horas por si, en caso de haber algún problema o retraso en la ejecución del automatismo nocturno, haya aún datos futuros para seguir usando por la aplicación web y evitar un fallo en cadena.

Una vez terminada la aplicación REST se despliega en una instancia EC2 con una imagen Linux de AWS. Para ello se instala Miniconda y las librerías necesarias para el correcto funcionamiento de la aplicación REST.

Debido a que los servidores web tradicionales no pueden, o no saben, ejecutar aplicaciones Python la comunidad de Python ha implementado *Web Server Gateway Interface* (WSGI). WSGI define una serie de reglas para que el servidor web se pueda comunicar con la aplicación Python. De entre los diversos servidores HTTP WSGI disponibles se decide utilizar Gunicorn y Nginx como proxy inverso.

Para poder iniciar y detener Gunicorn con nuestra aplicación a necesidad, se hará uso del `systemd` de Linux.

`Systemd` está formado por un conjunto de demonios, bibliotecas y herramientas para gestionar la administración y configuración del sistema. Entre otras cosas utiliza la activación de socket para inicializar los servicios y permite el inicio de los demonios bajo demanda.

Para que `systemd` inicie de forma automática el proceso de Gunicorn para manejar la conexión se creará e iniciará un socket para que esté escuchando las conexiones. El socket se creará con privilegios sudo en `/etc/systemd/system/gunicorn.socket`.

Además, por cada `.socket` que se cree tiene que haber un `.service` en el que se describa lo que se va a ejecutar cuando entre una conexión al socket. Este `.service` por norma general debe tener el mismo nombre que el `.socket`, por lo que se creará también un `/etc/systemd/system/gunicorn.service`. Es en este `gunicorn.service` donde se le indica que se ejecute nuestra aplicación con Gunicorn. En el archivo quedaría de la siguiente manera:

```
....

[Service]
WorkingDirectory=/home/ec2-user/django/predicctionapi/prediction
ExecStart=/home/ec2-user/miniconda3/envs/envTFG/bin/gunicorn
    --access-logfile -  --workers 1 --bind unix:/run/gunicorn.sock
    prediction.wsgi:application

....
```

Una vez activado el socket y comprobado que todo funciona correctamente solo quedaría configurar el Nginx para transferir el tráfico al socket.

Para ello se creará el archivo `/etc/nginx/sites-available/prediction`

en el que se especificará las reglas a seguir por Nginx.

```
server {
  listen 80;
  server_name 35.180.90.66;
  location = /favicon.ico { access_log off; log_not_found off; }
  location /static/ {
    root /home/ec2-user/django/predicctionapi/prediction;
  }

  location / {
    proxy_set_header Host $http_host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_pass http://unix:/run/gunicorn.sock;
  }
}
```

Como se puede observar en el código de arriba, en el archivo de configuración se está diciendo que escuche las peticiones al puerto 80, esto es, las HTTP, al servidor 35.180.90.66, y que las peticiones se las pase al socket `http://unix:/run/gunicorn.sock` creado anteriormente.

Por último se crearía un enlace simbólico de este archivo de configuración en `/etc/nginx/sites-enabled` y con esto ya quedaría desplegado y funcionando nuestra aplicación REST en la instancia EC2.

4.3.1 Resumen

En este Sprint se ha desarrollado e implementado una aplicación REST que se encarga de:

- Obtener los datos de polución de las diferentes estaciones de calidad del aire.
- Reentrenar los modelos asociados a cada estación con los últimos datos obtenidos para una mejor adaptación a nuevos comportamientos.
- Predecir las próximas 48h de contaminación con el nuevo modelo.
- Ofrecer los siguientes servicios REST:
  - Un servicio REST para obtener datos de polución en un período de tiempo.
  - Un servicio REST para obtener las predicciones.
  - Un servicio REST para entrenar un modelo en concreto.
  - Un servicio REST para que un modelo realice sus predicciones de nuevo.

4.4 Cuarto Sprint

Antes de comenzar a hablar sobre este Sprint y de los siguientes, cabe destacar que todas las imágenes que se mostrarán por el momento del lado cliente están sin maquetar, ya que el objetivo principal de estas historias de usuario es la lógica y sus funcionalidades y se crea lo necesario e indispensable en el lado cliente para asegurar el correcto funcionamiento. Para hacer la aplicación más bonita, intuitiva y sencilla de usar para el usuario existe una historia de usuario en el último Sprint en el que se tratará todo ese tema.

En este cuarto Sprint se crea la base para la aplicación web y se conecta con la aplicación REST para poder explotar la información que ofrece en nuestro beneficio.

Para asegurar la correcta comunicación entre la aplicación REST y la aplicación web se comienza terminando la historia de usuario 1 siguiendo lo planificado.

Primero se creará una pantalla en la que el usuario introducirá la hora en la que quiere saber la polución que habrá en la ciudad. Esta pantalla mostrará por defecto un *input* en el que se introducirá la fecha y hora en la que se quiera obtener la contaminación y un mapa limpio de la zona de Madrid.

Una vez introducida la fecha y hora se validará y se hará una petición POST al servicio `getPollutionInterval` publicado por la aplicación REST para conseguir los valores de polución de todas las estaciones en el intervalo de tiempo solicitado, siempre y cuando la fecha sea anterior al propio día en cuestión; si la fecha es del propio día aún no tendremos los datos actualizados en base de datos, pero sí las predicciones por lo que se hará una petición GET al servicio `predictions` publicado por la aplicación REST para conseguir los datos predichos por el modelo de aprendizaje profundo.

Para realizar las llamadas a la aplicación REST se hará uso de la librería *request* que tiene diversas funciones dependiendo del tipo de petición que se quiera mandar. En el caso del servicio `getPollutionInterval` se hará una petición POST indicando la URL en la que está desplegada la aplicación REST y el servicio que queremos llamar; y le pasamos el intervalo en formato JSON. Esto en código quedaría de la siguiente forma:

```
post_data = {
    "dateFrom": dt_date_from.strftime("%Y-%m-%d %H:%M"),
    "dateTo":dt_date_to.strftime("%Y-%m-%d %H:%M")
}
url = 'http://ec2-35-180-90-66.eu-west-3.compute.amazonaws.com/'

method = 'data/getPollutionInterval'

response = requests.post(url+method, json=post_data)
```

Una vez obtenida la contaminación de las estaciones en la fecha y hora solicitada se ne-

cesita obtener los contornos de las diferentes áreas de contaminación en Madrid. Para ello se crea una función que se puede reutilizar más adelante para mostrar la evolución de la contaminación en un intervalo de tiempo más largo. Esta función se llamará `get_paths()`.

En la función se realizará una interpolación de los valores de contaminación entre las diferentes estaciones para obtener una aproximación de los valores intermedios. Esto se consigue con la ayuda de la librería *scipy* y la función `griddata`. A esta función se le llama de la siguiente manera:

$$zi = griddata(x, y, v, xq, yq)$$

Dónde:

- ”x” sería un array con las longitudes de las estaciones.
- ”y” sería un array con las latitudes de las estaciones.
- ”v” sería un array con la contaminación de cada estación.
- ”xq” e ”yq” sería una matriz que indica en qué valores intermedios se quiere obtener el valor interpolado. Es decir, entre ambos serían las coordenadas de puntos intermedios entre las diferentes estaciones.
- ”zi” sería una matriz con los valores de contaminación interpolados correspondiente a cada valor de la matriz formada por ”xq” e ”yq”.

Una vez interpolado los valores entre estaciones se usará la función `contourf` de la librería *Matplotlib*. Esta función recibe las coordenadas de los puntos intermedios y sus valores correspondientes y en función de los umbrales de agrupación que se le pasen agrupará los valores dentro de un mismo rango en un color formando así diferentes contornos coloreados. En este caso, se usarán los límites y los colores oficiales del Índice de Calidad del Aire (ICA) para generar los diferentes contornos:

- Buena: Color verde (ICA de 0 a 50)
- Moderada: Color amarillo (ICA de 51 a 100)
- Dañina a la salud para grupos sensibles: Color naranja (ICA de 101 a 150)
- Dañina a la salud: Color rojo (ICA 151 a 200)
- Muy dañina a la salud: Color morado (ICA 201 a 300)

Obtenidos los contornos quedaría algo como lo que se puede observar en la Figura 4.6. Para poder llevar esta información a un mapa se obtienen los diferentes contornos generados por la función y se guardan en una lista los colores asociados a cada contorno y la posición de

sus bordes. Esta lista con la información final será lo que devuelva la función `get_paths()`.

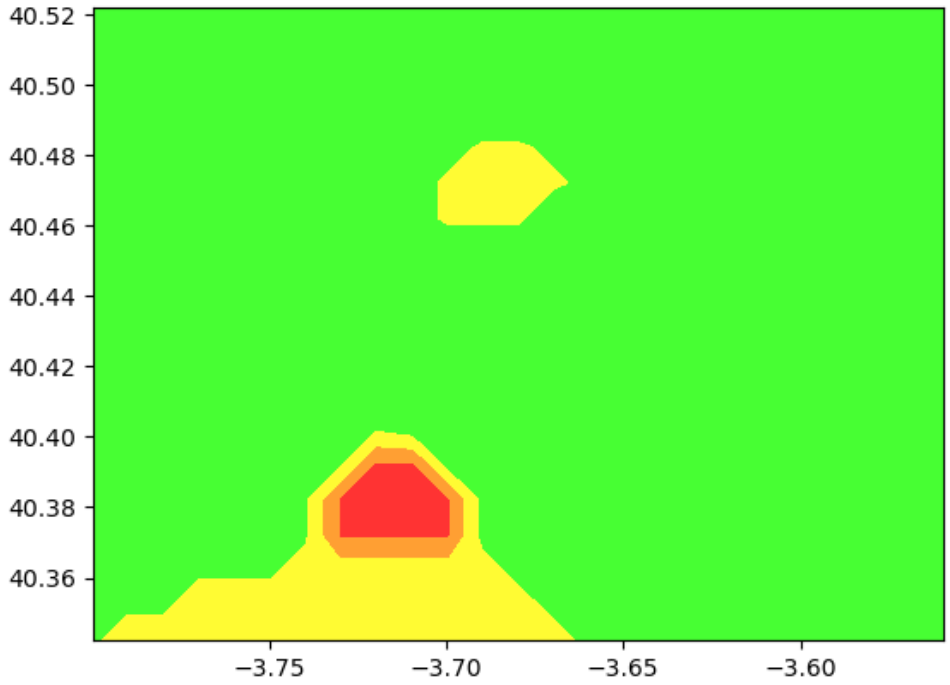


Figura 4.6: Resultado de *contourf*.

Por último se recargará la pantalla inicial cargando el JSON con la información de los colores y los bordes obtenidos de la función `get_paths()` y desde el HTML con JavaScript y *Leaflet* se dibujará en el mapa los contornos con las diferentes poluciones. El código JavaScript quedaría así:

```
1  for (i = 0; i < 3; i++) {
2    color=paths[i][0];
3    latlngs=paths[i].slice(1,paths[i].length);
4    var polygon = L.polygon(latlngs, {color: color}).addTo(map);
5  }
```

En la Figura 4.7 se puede observar el resultado final.

Una vez terminada la historia de usuario y comprobado que no hay problemas de comunicación con la aplicación REST se pasa a la tarea más complicada de la aplicación web: obtener las rutas y calcular la contaminación en cada ruta; lo cual se corresponde con la historia de

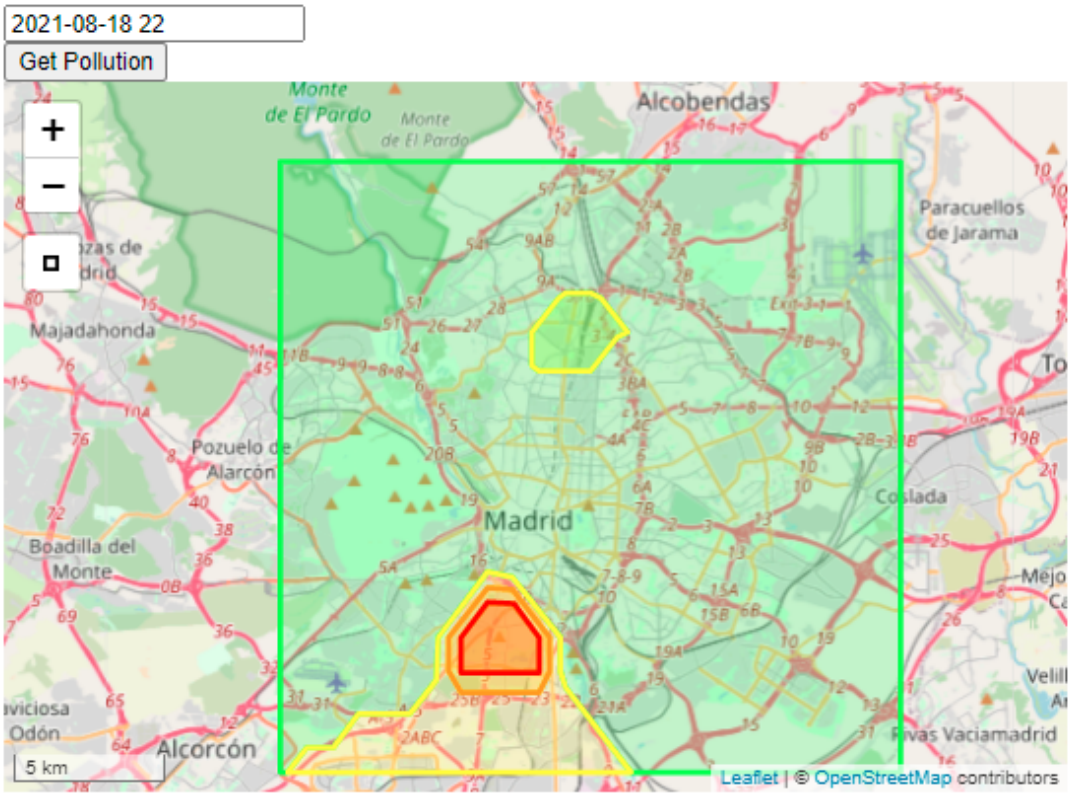


Figura 4.7: Ejemplo de la obtención de la contaminación.

usuario 5.

Para esta tarea se crea una pantalla en la que el usuario introducirá: el tipo de viaje (a pie, en bici o en coche), el lugar de salida, el destino y la fecha y hora en la que realizará el viaje.

En la Figura 4.8 se puede observar como se ve la pantalla inicial.

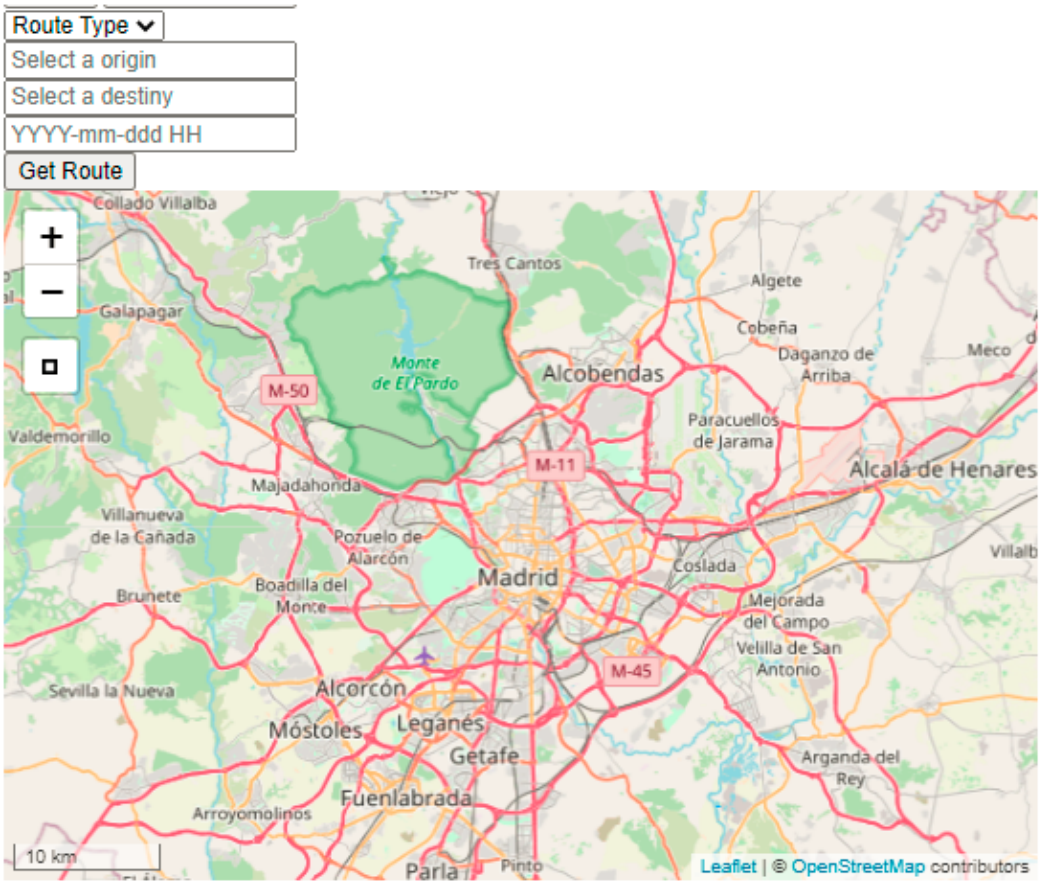


Figura 4.8: Imagen de la pantalla inicial.

Una vez introducidos los datos requeridos se hará una validación de que los datos son correctos, en caso de que falle se recargará la página mostrando un mensaje de error indicando que campo ha provocado el problema.

Una vez validados todos los campos se le hará una petición POST a la API de GoogleMaps pasándole el punto de origen y el punto de destino introducidos además del tipo de viaje y que se quiere más de una ruta y la API devolverá un JSON con toda la información.

Entre otras cosas, dentro del JSON devuelto por Maps habrá una entrada con las diferentes rutas y en cada ruta hay los diferentes *steps* que se tienen que recorrer para llegar al destino y dentro de esos *steps* lo que más interesa es el campo *polyline* que contiene una *polyline*

codificada con todos los puntos por los que hay que pasar en cada *step*. Para decodificarlo se hará uso de la librería *polyline* de Python y se juntarán todos en un único listado.

```
route_steps =
    [[route['legs'][0]['steps'][0]['start_location']['lat'],
    route['legs'][0]['steps'][0]['start_location']['lng']]]
        for step in route['legs'][0]['steps']:
            points = polyline.decode(step['polyline']['points'])
            for coord in points:
                route_steps.append([coord[0], coord[1]])
```

En la Figura 4.9 se puede observar un ejemplo de todos los puntos decodificados marcados en el mapa y se puede apreciar como hay muy poca distancia entre punto y punto pudiendo saber así por dónde discurre la ruta en todo momento.

Figura 4.9: Ejemplo de los puntos de las *polyline* decodificada.

Además se necesitará la contaminación en la fecha y hora solicitada, sea futura o pasada. Para ello se realizará una petición GET a la aplicación REST para obtener los datos de contaminación y en función de si es una petición a fecha futura se usará la función para obtener las predicciones y si es pasada se llamará a la función que saca la información de la base de datos.

Teniendo las rutas y la contaminación se calculará la contaminación para cada posible

ruta con ayuda de la función `griddata` utilizada anteriormente. En esta ocasión se pasará las coordenadas de la ruta como puntos intermedios en los que interpolar la contaminación obteniendo como resultado la contaminación en cada punto de la ruta. Una vez que se tiene la contaminación interpolada de cada punto de la ruta se le realizará una media aritmética para obtener la contaminación media de la ruta. Esto en código quedaría de la siguiente manera:

```
x = np.asarray(stations_lat)
y = np.asarray(stations_lng)
z = np.asarray(pollution_values)
xi = [points_route[0][0]]
yi = [points_route[0][1]]
for point in points_route[1:]:
    yi.append(point[0])
    xi.append(point[1])

zi = griddata((x, y), z, (xi, yi), method='linear')
weight = 0
weight = np.nansum(zi)
total_nums = len(zi) - np.count_nonzero(np.isnan(zi))
pollution_route = (weight / total_nums)
```

Una vez obtenida la contaminación media de la ruta se crea una lista con la información de cada ruta con los siguientes datos:

- Duración de la ruta
- Distancia que se recorre en la ruta
- Contaminación media de la ruta
- Coordenadas que marcan la ruta
- Color con el que se coloreará la ruta.

Se ordenarán las rutas en función de su contaminación para que la primera sea siempre la que presenta una media menor de contaminación y se le asignará el color verde, y al resto de rutas se les irá asignando los colores amarillo, rojo y negro respectivamente, aunque no suele haber más de tres en total.

Por último, gracias al estudio que se hizo de la serie temporal se sabe que a lo largo del día la contaminación tiene períodos pico y valle, por lo que es posible que busquen rutas en períodos pico. En caso de que el usuario esté registrado si solicita una ruta en período pico, se comprueba si hay eventos cerca del punto de salida por si el usuario quiere asistir al evento para hacer tiempo mientras no baje la contaminación a un período valle.

Para saber si hay algún evento cercano todas las noches se obtendrá un listado de los eventos que hay programados en Madrid con la información más actualizada posible y cuando

el usuario haga la petición de la ruta se recorrerá el listado y se comprobará si hay algún evento programado cercano al punto de salida de la ruta en la hora solicitada. En caso de que haya algún evento cercano se le pasará la información a la pantalla junto al listado con la información de las diferentes rutas; y en caso de que no haya ningún evento se enviará únicamente la información de las rutas.

Para dibujar las rutas sobre el mapa se hará uso de *Leaflet* y JavaScript. En JavaScript se recorre el listado con la información de las rutas y para cada ruta se dibujará una *polyline* con el color asociado, además de marcar los puntos de salida y llegada. En la Figura 4.10 se puede observar como se vería una ruta dibujada en el mapa y a continuación se puede ver como se vería esto en código:

```
steps = routesInfo[0][3]
lastStep = routesInfo[0][3].length
L.marker(steps[0]).addTo(map);
L.marker(steps[lastStep-1]).addTo(map);
i = 0
while  (i < routesInfo.length) {
    points = routesInfo[i][3]
    color_route = routesInfo[i][4]
    var polyline = L.polyline(points, {color:
    color_route}).addTo(map);
    i = i + 1
}
```

Y para mostrar la información de cada ruta (distancia, duración...) se añadirá una leyenda debajo del mapa indicando toda esa información indicando cuál es la recomendada. En la Figura 4.11 se puede observar un ejemplo del resultado completo.

En caso de que se encuentre algún evento se le dará una explicación al usuario y se le da la opción de que escoja si quiere ver los eventos cercanos o ignorar el tema. En la Figura 4.12 se observa el mensaje informativo que se le muestra al usuario; mientras que en la Figura 4.13 se puede ver cómo se muestran en el mapa los eventos cercanos junto a su información.

4.4.1 Pruebas

Para cada historia de usuario se han realizado los test unitarios pertinentes en los que se comprueban diferentes casos de prueba como:

- La validación de todos los parámetros de entrada.
- El correcto funcionamiento de todas las lógicas.
- La gestión de los errores esperados.

Para ello se ha hecho uso de la librería *Django-test* en la que viene incluidas clases como la *Client* que se utiliza para simular las peticiones que recibiría la aplicación a través de las pantallas. Otra librería a destacar que ha sido de gran ayuda es la librería *mock* que se utiliza para falsear el resultado de ciertas funciones, esto nos permite forzar situaciones especiales para confirmar que la aplicación reacciona ante ellas como se espera o fingir resultados de llamadas a aplicaciones de terceros para no tener que depender de su disponibilidad.

Por ejemplo, para forzar que buscando las rutas ha habido un problema con el punto de salida seleccionado se falsea con la librería *mock* la llamada que se hace a la API de Google Maps y se hace que devuelva un JSON incluyendo un error con el punto de salida. Para conseguir falsear la función de la llamada a la API la librería *mock* permite el uso de decoradores encima de la función de test para indicar que función queremos falsear y luego en el código de la función de test indicamos que valor queremos que devuelva. Esto en código quedaría de la siguiente manera:

```
1 @patch('user.views.get_info_routes_maps')
2 def test_validation_origin(self, mock_get_info_routes_maps):
3     c = Client()
4     mock_get_info_routes_maps.return_value =
      JSON_CON_PROBLEMA_EN_SALIDA
5     date = (datetime.now() + timedelta(hours=1)).strftime("%Y-%m-%d
      %H")
6     response = c.post('/', {'transport': route_type, 'origin':
      origin, 'destiny': destiny,
7                             'date': date})
```

Por último, para llevar un control de la efectividad de las pruebas se controlará la cobertura de los test. Para ello se hace uso de la herramienta *coverage* de Python, la cual permite sacar informes indicando el porcentaje de líneas probadas e incluso podemos sacar que líneas nos quedan por probar.

La mayor parte de la lógica reside en el `views.py` por lo que será el foco de las pruebas y lo más importante de los informes sacados por *coverage*.

Las primeras pruebas e informe corresponde a la historia de usuario 1. En la Figura 4.14 se muestra el informe de *coverage* y en él se puede observar que se ha alcanzado el 100% de cobertura en casi todas las clases, incluyendo la `views.py`, por lo que se ha conseguido testear todo el código implementado para la historia de usuario 1.

En la Figura 4.15 se puede ver el informe para la historia de usuario 5. Como se puede observar se sigue con una cobertura del 100% en el `views.py` por lo que todas las funcionalidades nuevas han sido comprobadas. En este informe es necesario aclarar porque hay algunos archivos con un porcentaje de cobertura bajo.

Por un lado se tiene el `externarls_calls.py` en el que se recogen todas las fun-

ciones que realizan llamadas a terceros, como por ejemplo, a la API de Google Maps y a la aplicación REST con el modelo de aprendizaje profundo. Estas llamadas son falseadas con la librería *mock* por los motivos comentados anteriormente y es por eso que su porcentaje de cobertura es bajo, ya que no hay nada que probar en una función que simplemente obtiene la respuesta de un tercero.

Por otro lado, se tiene `updaterAPI.py` y `updater.py`, las cuales son utilizadas para crear el automatismo que descarga diarimente los eventos de la página de datos públicos de Madrid. Al igual que antes, como estas clases son utilizadas para simplemente obtener el resultado de un tercero no hay nada que comprobar y no se le realizan ninguna prueba, teniendo como consecuencia lógica un porcentaje de cobertura bajo.

4.4.2 Resumen

En este Sprint se ha empezado el desarrollo de la aplicación web. En este comienzo se ha terminado de implementar la funcionalidad para mostrar la contaminación sobre el mapa de Madrid en una hora concreta.

También se ha implementado la segunda funcionalidad clave de la aplicación: la obtención de la mejor ruta. Ahora mismo sea un usuario registrado o anónimo podrá realizar búsquedas de rutas en algún instante de tiempo posterior y se le mostrarán varias rutas indicándole cuál es la que recomendamos en concreto.

Además, se han creado los test unitarios pertinentes para asegurar el correcto funcionamiento de estas funciones.

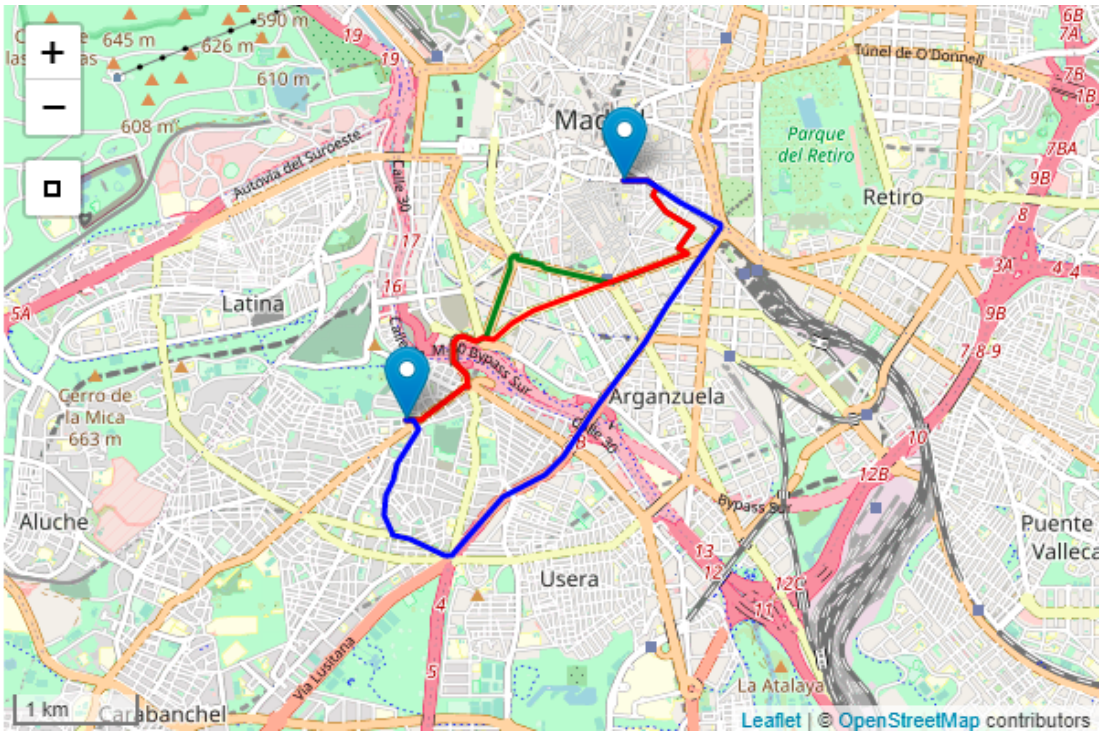


Figura 4.10: Ejemplo de la *polyline* dibujada sobre el mapa.

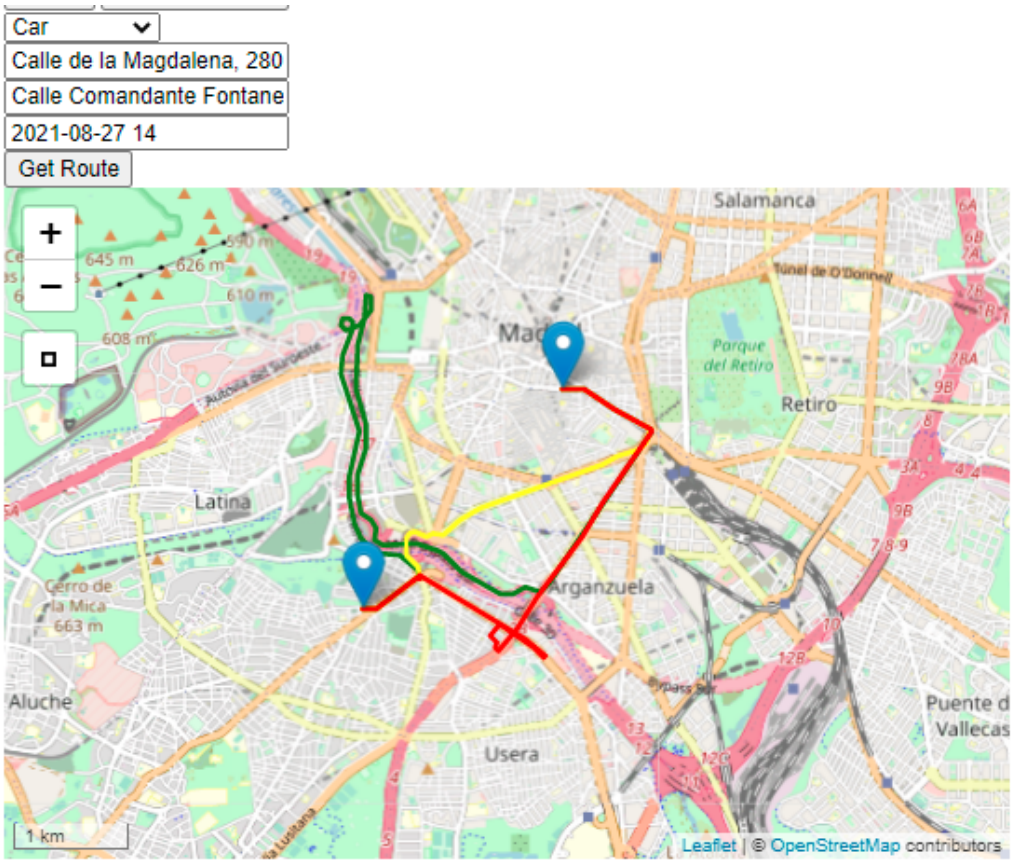


Figura 4.11: Ejemplo completo de una búsqueda.

Figura 4.12: Mensaje informativo de eventos cercanos.

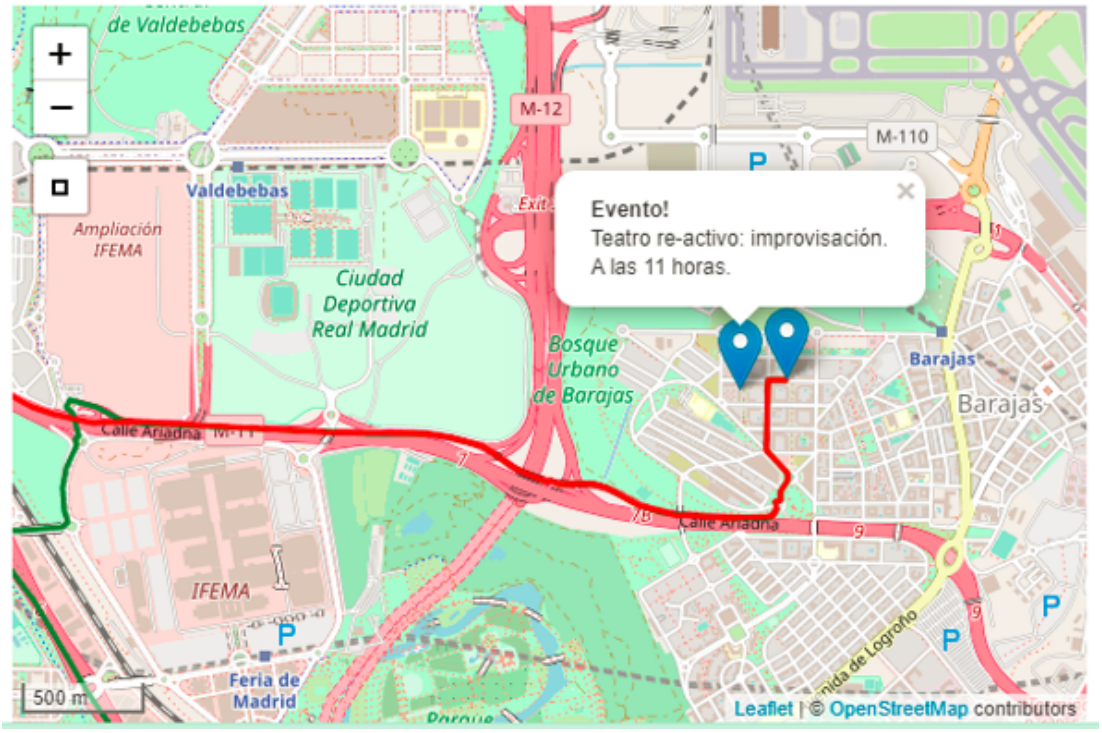


Figura 4.13: Mapa con evento cercano y rutas.

```
Name                                                  Stmts   Miss  Cover
-----------------------------------------------------------------------
aplicationweb\__init__.py                                 0      0   100%
aplicationweb\settings.py                                29      0   100%
aplicationweb\urls.py                                     6      0   100%
manage.py                                                12      2    83%
user\__init__.py                                          0      0   100%
user\admin.py                                            11      0   100%
user\apps.py                                              3      0   100%
user\externals_call.py                                   14      0   100%
user\forms.py                                            14      0   100%
user\migrations\0001_initial.py                           8      0   100%
user\migrations\0002_historic.py                          6      0   100%
user\migrations\0003_auto_20191227_1702.py                4      0   100%
user\migrations\0004_favourites.py                        7      0   100%
user\migrations\0005_favourites_alias.py                  4      0   100%
user\migrations\0006_auto_20191230_1617.py                4      0   100%
user\migrations\0007_auto_20191230_1708.py                4      0   100%
user\migrations\0008_auto_20210819_1100.py                4      0   100%
user\migrations\0009_auto_20210819_1101.py                4      0   100%
user\migrations\0010_historic_route_type.py               4      0   100%
user\migrations\0011_auto_20210821_2313.py                4      0   100%
user\migrations\0012_auto_20210824_1156.py                4      0   100%
user\migrations\__init__.py                               0      0   100%
user\models.py                                           27      2    93%
user\tests\__init__.py                                    0      0   100%
user\tests\tests_events.py                                0      0   100%
user\tests\tests_evollution_pollution.py                  0      0   100%
user\tests\tests_favourites.py                            0      0   100%
user\tests\tests_historic_pollution.py                   84      0   100%
user\tests\tests_historic_searchs.py                      0      0   100%
user\tests\tests_profile.py                               0      0   100%
user\tests\tests_redirections.py                         24      0   100%
user\tests\tests_search.py                                0      0   100%
user\urls.py                                              7      0   100%
user\views.py                                           143      0   100%
-----------------------------------------------------------------------
TOTAL                                                   431      4    99%
```

Figura 4.14: Informe de cobertura de la historia de usuario 1.

```
Name                                                Stmts   Miss  Cover
-----------------------------------------------------------------------
aplicationweb\__init__.py                               0      0   100%
aplicationweb\settings.py                              29      0   100%
aplicationweb\urls.py                                   7      0   100%
manage.py                                              12      2    83%
user\__init__.py                                        0      0   100%
user\admin.py                                          11      0   100%
user\apps.py                                            6      0   100%
user\externals_call.py                                 25     10    60%
user\forms.py                                          14      0   100%
user\migrations\0001_initial.py                         8      0   100%
user\migrations\0002_historic.py                        6      0   100%
user\migrations\0003_auto_20191227_1702.py              4      0   100%
user\migrations\0004_favourites.py                      7      0   100%
user\migrations\0005_favourites_alias.py                4      0   100%
user\migrations\0006_auto_20191230_1617.py              4      0   100%
user\migrations\0007_auto_20191230_1708.py              4      0   100%
user\migrations\0008_auto_20210819_1100.py              4      0   100%
user\migrations\0009_auto_20210819_1101.py              4      0   100%
user\migrations\0010_historic_route_type.py             4      0   100%
user\migrations\0011_auto_20210821_2313.py              4      0   100%
user\migrations\0012_auto_20210824_1156.py              4      0   100%
user\migrations\__init__.py                             0      0   100%
user\models.py                                         27      1    96%
user\tests\__init__.py                                  0      0   100%
user\tests\tests_events.py                             21      0   100%
user\tests\tests_evollution_pollution.py                0      0   100%
user\tests\tests_favourites.py                          0      0   100%
user\tests\tests_historic_pollution.py                 83      0   100%
user\tests\tests_historic_searchs.py                    0      0   100%
user\tests\tests_profile.py                             0      0   100%
user\tests\tests_redirections.py                       24      0   100%
user\tests\tests_search.py                            258      0   100%
user\updater.py                                         7      0   100%
user\updaterAPI.py                                     15     11    27%
user\urls.py                                            7      0   100%
user\views.py                                         313      0   100%
-----------------------------------------------------------------------
TOTAL                                                 916     24    97%
```

Figura 4.15: Informe de cobertura de la historia de usuario 5.

4.5 Quinto Sprint

En este quinto Sprint se realizarán las historias de usuario: 6, 7, 8, 2, 3, 4.

Se empezará por la historia de usuario 4 que es la que mayor complejidad tiene.

Se desarrolla una pantalla inicial en la que el usuario tendrá que indicar entre que dos fechas quiere ver la evolución de la contaminación sobre el mapa. Para mostrar la evolución de manera ágil y sencilla al usuario en vez de recargarle la página entera con los datos nuevos se trabajará toda la información y validaciones mediante JavaScript y AJAX, consiguiendo así recargar de manera dináminca y asíncrona unicamente la información mostrada en el mapa.

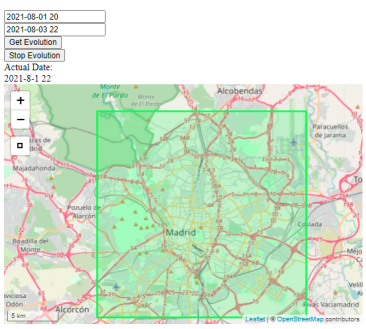
Con JavaScript se harán las validaciones pertinentes de la información introducida en la pantalla inicial y una vez comprobado que las fechas cumplen todos los requisitos se iterará hora a hora entre la fecha de inicio y la de fin mostrando para cada hora un mapa con la polución en Madrid coloreando las diferentes zonas en función del Índice de Calidad del Aire (ICA) al igual que se hizo en la historia de usuario 1. Para que el usuario tenga tiempo a procesar la imagen en cada hora se necesita algún tipo de tiempo de espera entre iteración e iteración, lo cual se consigue gracias a la función *setTimeout* de JavaScript, la cual llama a la función que le indiques pasado un tiempo especificado. Con esta función y recursividad se consigue que mientras no se alcance la fecha fin se continúe avanzando en el tiempo y que haya un tiempo de espera de 1 segundo entre iteración e iteración.

En cada iteración se limpiará cualquier tipo de *layer* dibujado sobre el mapa en antiguas iteraciones para tener el mapa limpio para la iteración actual; una vez limpiado el mapa se hará una llamada mediante AJAX a la aplicación web que devolverá un listado con los contornos y el color correspondiente para cada contorno al igual que en la historia de usuario 1. Con el resultado de la petición AJAX se dibujarán los contornos sobre el mapa y en todo momento se irá indicando por la pantalla la fecha y hora correspondiente a la información mostrada en el mapa. En la Figura 4.16a se puede observar un ejemplo; y en caso de que por algún motivo no se tengan datos de alguna fecha se indicará en esa iteración y se seguirá iterando sin mayor complicación, un ejemplo sería lo mostrado por la Figura 4.17.

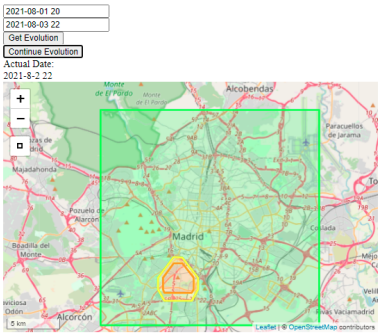
Por último, se crea un botón "*Stop Evolution*" que detiene el bucle iterativo cuando se pulsa por si por algún motivo el usuario quiere detener la evolución en algún punto en concreto. Además, este botón se cambia por "*Continue Evolution*" por si se quiere retomar la evolución dónde se dejó. Se puede ver un ejemplo de este comportamiento en la Figura 4.16b.

Tanto para la historia de usuario 2 como para la historia de usuario 3 se necesita un sistema de login y control de usuarios por lo que se hará primero las historias de usuario 6, 7, 8. En concreto, se empezará por la historia de usuario 7, el registro de usuarios, ya que para que puedan iniciar y cerrar sesión los usuarios primero tienen que estar registrados.

El primer paso sería crear en la base de datos una tabla para los usuarios con varios datos



(a) Ejemplo de una evolución normal.



(b) Ejemplo de una evolución parada.

Figura 4.16: Ejemplo demostración de la evolución de la contaminación.

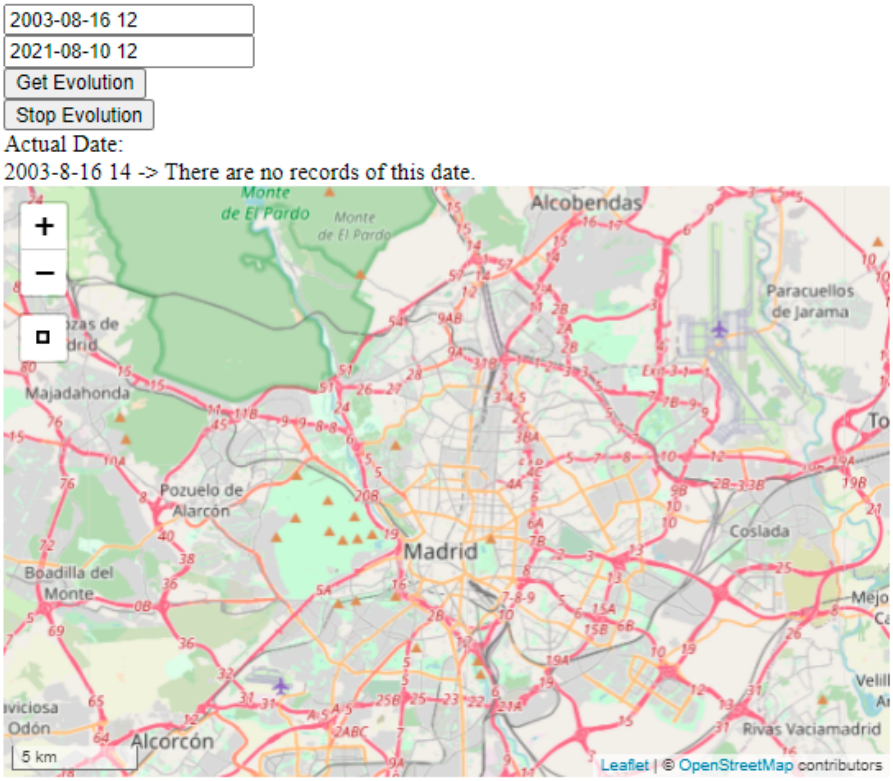


Figura 4.17: Ejemplo de una evolución sin datos.

personales del usuario. En concreto, al usuario se le pedirá:

- Un nombre de usuario para identificarlo.
- Un correo para poder comunicarse con el usuario.
- El país en el que reside.

Una vez creada la tabla en base de datos se pasa a la creación de la pantalla en la que se le pedirá los datos al usuario. El framework *Django* proporciona unas clases genéricas para realizar diferentes lógicas estándares en toda aplicación, y entre ellas se encuentra la clase *CreateView*. Esta clase muestra un *form* para la creación de un objeto, y en caso de que haya algún error de validación se muestra la página con el error. Lo primero es crear el *form* con los campos que se necesitan, lo cual en código quedaría así:

```
class CustomUserCreationForm(UserCreationForm):
    country = forms.ChoiceField(choices=CustomUser.COUNTRY_LIST)
    class Meta:
        model = CustomUser
        fields = ('username', 'email', 'country')
```

El campo *country* es finito, es decir, solo existe un número fijo de países, los cuales están metidos en la base de datos. Por lo que para mostrarlos en el formulario se carga la lista de países en el campo country para poder mostrarlos en la pantalla en forma de *dropdown* con todos los países posibles como opción a seleccionar.

Por último para utilizar la clase *CreateView* se crea una clase a la que se le pasa el *CreateView* por parámetro y se le indica el *form* que va a utilizarse, a dónde redirigir si todo sale correcto, y que plantilla HTML usar para mostrar el formulario. Esto en código quedaría de la siguiente manera:

```
class SignUpView(CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

Además, relacionada con esta historia de usuario se crea otra pantalla en la que el usuario puede visualizar sus datos de perfil y modificarlos en cualquier momento. Para modificar las contraseñas se hará uso de una de las facilidades proporcionadas por *Django*: *django.contrib.auth.url*. Esta librería proporciona todo lo necesario para gestionar las contraseñas además de permitir crear páginas de autenticación y gestionar el *login* y el *logout*. Por lo que con esta ayuda también se realizarán las historias de usuario 6 y 8. Esta librería aporta el mapeador de URL, las vistas *views* y los formularios *forms*. Lo único que no incluiría y tendría que hacerse manualmente serían las pantallas HTML.

El primer paso es añadirlo a las URL de la siguiente manera:

```
urlpatterns += [
    path('user/', include('django.contrib.auth.urls')),
]
```

Esto generará automáticamente todas estas URL's:

- ^user/login/$ [name='login']
- ^user/logout/$ [name='logout']
- ^user/password_change/$ [name='password_change']
- ^user/password_change/done/$ [name='password_change_done']
- ^user/password_reset/$ [name='password_reset']
- ^user/password_reset/done/$ [name='password_reset_done']
- ^user/reset/done/$ [name='password_reset_complete']
- ^accounts/reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]1,13-[0-9A-Za-z]1,20)/$ [name='password_reset_confirm']

Todas estas URL's buscarán sus correspondientes plantillas bajo el directorio *registration* dentro del directorio de plantillas.

Para el cambio de contraseñas desde el profile se hará uso de las URL's:

- ^user/password_change/$ [name='password_change']
- ^user/password_change/done/$ [name='password_change_done']

Para ello se deben crear los correspondientes HTML que son: password_change_form.html y password_change_done.html. Una vez creadas las pantallas la librería *django.contrib.auth.url* ya hace el resto y solo quedaría realizar las pruebas pertinentes.

A continuación se empieza la historia de usuario 6 en la que como se comentó anteriormente se seguirá haciendo uso de la librería *django.contrib.auth.urls*. En esta ocasión se utilizarán las siguientes URL's:

- ^user/login/$ [name='login']
- ^user/password_reset/$ [name='password_reset']
- ^user/password_reset/done/$ [name='password_reset_done']

- ^user/reset/done/$ [name='password_reset_complete']
- ^accounts/reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]1,13-[0-9A-Za-z]1,20)/$ [name='password_reset_confirm']

El punto principal de esta historia de usuario es el login y se consigue con la URL ^user/login/$ que cargará la plantilla de login y en caso de que todos los datos sean correctos iniciará sesión y nos redirigirá de nuevo al *home* pero con la sesión ya iniciada.

Pero a mayores se implementará un sistema para recuperar contraseña en caso de que se le haya olvidado al usuario y esto se implementa a través del resto de URL's mencionadas anteriormente. Para ello, se mostrará en la pantalla de *login* un link para recuperar la contraseña en caso de olvido que nos llevará a una pantalla solicitando la dirección de correo del usuario. En caso de que exista un usuario con esa dirección de correo se le enviará un correo con una URL con un *token* al final asociado a la cuenta para que pueda cambiar la contraseña.

Para poder enviar correos se crea una cuenta de Gmail para la aplicación llamada `routingpollution@gmail.com`. En los ajustes de *Django* se configurarán los siguientes valores para poder enviar correos con esa cuenta:

```
1 EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
2 EMAIL_HOST = 'smtp.gmail.com'
3 EMAIL_PORT = 587
4 EMAIL_USE_TLS = True
5 EMAIL_HOST_USER = 'routingpollution@gmail.com'
6 EMAIL_HOST_PASSWORD = 'xxxxxxx'
```

Una vez enviado el correo se accede a la URL facilitada en el correo y se cambia la contraseña por una nueva para poder acceder de nuevo a la cuenta de usuario.

Por última en esta historia de usuario se hará uso del decorador `login_required(login_url='/user/login/')` que se pondrá delante de cada función que requiera tener cuenta de usuario para poder acceder. En nuestro caso, y por el momento, se pondrá delante de las vistas relacionadas con la historia de usuario 1 y la historia de usuario 4.

Por último, una vez más gracias a la librería *django.contrib.auth.url* para la historia de usuario 8 solo se necesita hacer redirigir a la URL ^user/logout/$ [name='logout'] para cerrar la sesión.

Una vez que se tiene un registro de usuarios se puede empezar a realizar las dos historias de usuario restantes: 3 y 4.

Se empieza por la historia de usuario 3 y para ello se crea una tabla donde se guardará cada búsqueda que haga un usuario, en concreto, se guardarán los siguientes valores: tipo de ruta, punto de salida, punto de llegada y fecha de salida.

Para guardar los datos en la base de datos cuando se realice una búsqueda se comprobará si la petición viene de un usuario registrado y en caso de ser así se guarda en la base de datos un registro de la búsqueda. Esto en código quedaría de la siguiente manera:

```
if request.user.is_authenticated:
    user = CustomUser.objects.get(username=request.user)
    origin = origin
    destiny = destiny
    historic_entry = Historic(origin=origin, destiny=destiny,
    date=date_dt, user_id=user.id)
    historic_entry.save()
```

Además, se dará la opción a los usuarios de poder ver el registro de sus búsquedas, en las que se mostrará en una tabla paginada su histórico de búsqueda. Para conseguir paginar el histórico se usará la librería *django.core.paginator* la cual tiene la clase `Paginator` a la que se le puede pasar el *QuerySet* con todo el histórico de búsquedas y el número de resultados que se quiere mostrar en cada página. Esto en código quedaría de la siguiente manera:

```
historic_list = historic_list.order_by('date')
paginator = Paginator(historic_list, 10)
page = request.GET.get('page', 1)
try:
    historic_result = paginator.page(page)
except PageNotAnInteger:
    historic_result = paginator.page(1)
except EmptyPage:
    historic_result = paginator.page(paginator.num_pages)
```

Para especificar la página se pasará por parámetro a través de una petición GET, en caso de que no se indique ninguna página se mostrará por defecto la primera página y en caso de que se solicite una página que no existe, se devolverá la última existente.

También, se implementará un sistema de filtrado para facilitar al usuario el encontrar cualquier búsqueda del histórico. El filtro se podrá hacer por: el tipo de ruta (andando, conduciendo, en bici); el punto de salida; el punto de destino; y/o el día que se realizaría el viaje. El filtrado se puede realizar por múltiples campos, además se ofrece un botón que limpia todos los campos del filtro.

Por último, se da la opción de eliminar cualquier entrada del histórico de búsquedas que se quiera, ya se consiga eliminar o no se mostrará un aviso informando el resultado de la operación para la comodidad del usuario. Además, si se elimina con el filtro activado, se mantendrá el filtro en todo momento y se actualizará la página sin la búsqueda eliminada.

En la Figura 4.18 se puede observar un ejemplo de cómo quedaría esta historia de usuario.

A continuación, se desarrolla la historia de usuario 4. Primero se creará una nueva tabla en base de datos con los atributos: dirección; ciudad; estado o comunidad autónoma; país;

alias es decir, un nombre amigable por el que reconocer la ubicación; coordenadas del lugar; y contador con el número de veces que se ha usado el lugar.

Una vez creada esta tabla ya se tendría el estado final de la base de datos. En la Figura 4.19 se puede observar el diagrama Entidad-Relación representativo.

En la Figura 4.20 se puede observar el diagrama UML de la aplicación web con los 3 modelos que se han creado, y cómo nuestro modelo de usuarios hereda de la clase abstracta que ofrece *Django*. Como se comentó en el Sprint 3.2.1, los *Fields* de Python son públicos por defecto ya que su manera de restringir el acceso es simbólica y se puede obviar fácilmente por lo que no se suele modificar. Además, nuestros modelos heredan de la clase *Models* ciertos métodos genéricos como *save* o *delete* pero por motivos de limpieza no se han incluido en el UML.

El siguiente paso ha sido crear una pantalla en la que mostrar en una tabla todos lugares favoritos que se hayan creado. Al igual que en la historia de usuario anterior la tabla será paginada y se podrá mover entre las diferentes páginas indicándolo a través de los parámetros de la petición GET. Desde esta misma pantalla se podrá eliminar o editar cualquier Favorito deseado y se informará del resultado de la operación al usuario. Además, también se dispondrá de un botón para acceder a la pantalla de creación de lugares favoritos. En la Figura 4.21 se puede ver un ejemplo de cómo quedaría. También se añade un filtro de búsqueda para facilitar al usuario el encontrar un lugar o lugares en concreto; el filtro permitirá realizar búsquedas en función de la ciudad y/o el alias.

En la pantalla de creación de favoritos se tendrá que especificar primero la dirección, ciudad, estado y país del lugar. Una vez indicado estos valores el usuario deberá pulsar un botón para obtener las coordenadas del lugar, lo cual ejecutará una llamada a través de AJAX a nuestra aplicación web, dónde se obtendrán las coordenadas del lugar haciendo una petición a la API de Google Maps. Una vez obtenidas las coordenadas, estas se mostrarán en un mapa para que el usuario pueda comprobar visualmente que se han obtenido las coordenadas correctas del lugar que desean. Por último, deben añadir un alias con el que se identificará el lugar, cabe destacar que para cada usuario no puede haber dos alias iguales. En la Figura 4.22 se puede apreciar la pantalla donde se registran los lugares favoritos ya con las coordenadas obtenidas.

Además, cuando se vaya a realizar una búsqueda de ruta se mostrará como sugerencias los 3 lugares favoritos más utilizados, y según se vaya escribiendo en el *input* se irá autocompletando con los posibles lugares favoritos que contengan esa cadena de texto; y por cada vez que se use para una búsqueda aumentará en uno el número de usos en la base de datos. Para poder mostrar las sugerencias se le pasa al lado cliente todos los favoritos del usuario y luego con JavaScript por cada evento de escritura en los *inputs* se obtiene el texto introducido, se comprueba que lugares contienen el texto introducido en su alias y se muestran los 3 primeros. En código quedaría de la siguiente manera:

Historic Searchs

Route Type | Select a origin | Select a destiny | YYYY-mm-ddd | Reset Filter | Filter

Identify	Route Type	Origin	Destiny	Date	Actions
23	walking	Calle Inglaterra, 8, 28019 Madrid	Calle de la Magdalena, 28012 Madrid	Aug. 22, 2021, 4 a.m.	
19	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 5 a.m.	
10	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 6 p.m.	
8	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
7	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
6	walking	Calle de la Magdalena, 28012 Madrid	Calle de la Magdalena, 28012 Madrid	Aug. 22, 2021, 8 p.m.	
5	driving	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
4	driving	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
3	driving	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
2	driving	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	

Figura 4.18: Ejemplo del historial de búsquedas.

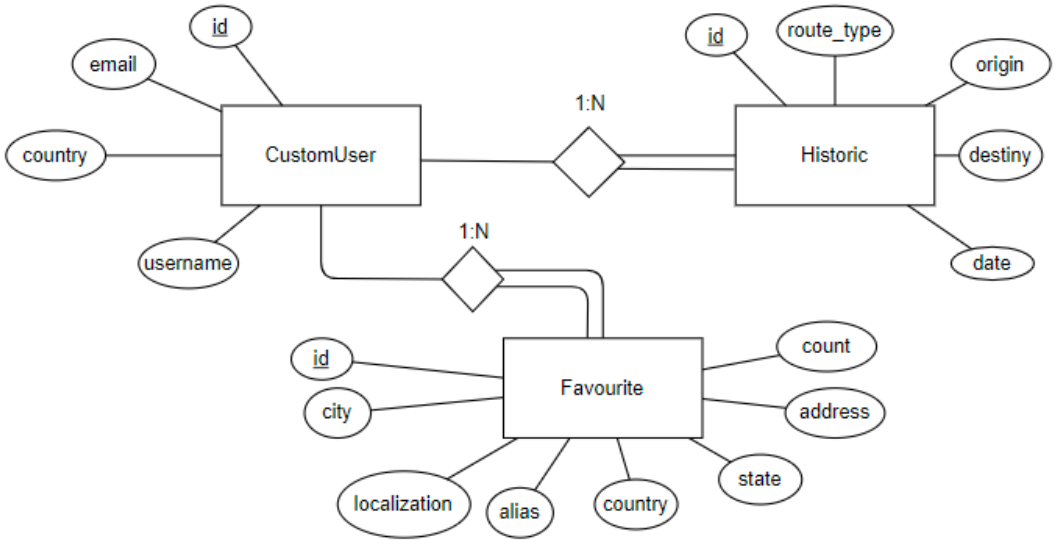


Figura 4.19: Diagrama Entidad-Relación definitivo.

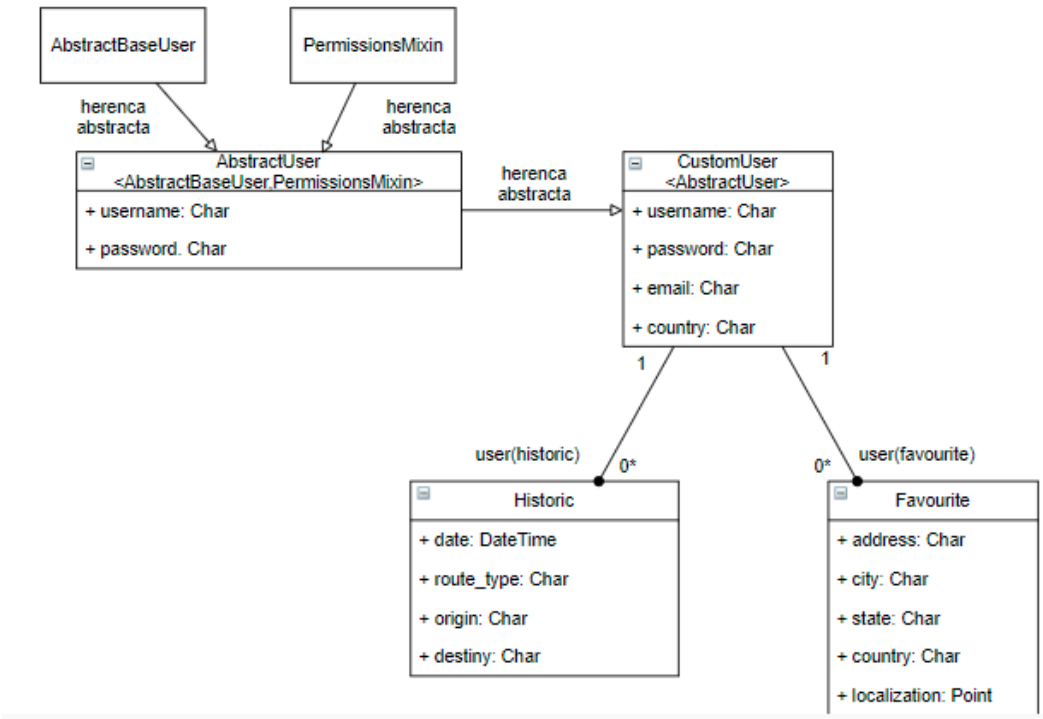


Figura 4.20: Diagrama UML definitivo.

Figura 4.21: Ejemplo del historial de lugares favoritos.

```
var origin = document.querySelector('#id_origin');
var origin_favs = document.querySelector('#origin_favs');
var origin_favs_template =
    document.querySelector('#origin_favs_template').content;

origin.addEventListener('keyup', function handler(event) {
    while (origin_favs.children.length)
    origin_favs.removeChild(origin_favs.firstChild);
    var inputVal = new RegExp(origin.value.trim(), 'i');
    var clonedOptions = origin_favs_template.cloneNode(true);
    var set = Array.prototype.reduce.call(clonedOptions.children,
    function searchFilter(frag, el) {
        if (inputVal.test(el.textContent) && frag.children.length <
    5) frag.appendChild(el);
        return frag;
    }, document.createDocumentFragment());
    origin_favs.appendChild(set);
    });
```

4.5.1 Pruebas

En la Figura 4.23 se puede observar el informe de cobertura que incluye los test pertinentes a la historia de usuario 4, y una vez más se mantiene el 100% en `views.py` por lo que se está comprobando todo el código nuevo relacionado con esta historia de usuario.

Para la historia de usuario 7 se realiza únicamente pruebas para la lógica de mostrar y editar el perfil, ya que la lógica para la creación del usuario así como el cambio y reseteo de contraseña vienen incluidos en la librería de *Django* y se da por supuesto que funciona correctamente. Esta misma premisa se aplican también para las historias de usuario 6 y 8. Además, a raíz de estas historias de usuario se añaden unas cuantas modificaciones y pruebas nuevas a otras historias de usuario, ya que ahora se requiere estar registrado para acceder a las funcionalidades de las historias de usuario 1 y 4.

Para comprobar si se puede acceder sin estar registrado primero se probará a hacer una llamada a las URL's pertinentes sin estar registrado; la respuesta deberá contenter el código *HTTP 302 Found*, el cual indica que se ha realizado una redirección, y la URL que devuelve es la del *login*. Además, se realizará una petición estando registrados a las mismas URL's y, en esta ocasión, la respuesta debe contener el código *HTTP 200 OK*.

Para realizar peticiones fingiendo estar el usuario registrado primero se deberá crear un usuario. Como se va a necesitar el usuario en todo el conjunto de pruebas se creará un usuario común para todas las pruebas del conjunto. Esto se consigue haciendo uso de la función `setUp`, la cual es un método de la superclase `TestCase`, que se ejecuta al comienzo de cada conjunto de pruebas.

Figura 4.22: Ejemplo de cómo agregar un lugar favorito.

Name	Stmts	Miss	Cover
aplicationweb__init__.py	0	0	100%
aplicationweb\settings.py	29	0	100%
aplicationweb\urls.py	7	0	100%
manage.py	12	2	83%
user__init__.py	0	0	100%
user\admin.py	11	0	100%
user\apps.py	6	0	100%
user\externals_call.py	25	10	60%
user\forms.py	14	0	100%
user\migrations\0001_initial.py	8	0	100%
user\migrations\0002_historic.py	6	0	100%
user\migrations\0003_auto_20191227_1702.py	4	0	100%
user\migrations\0004_favourites.py	7	0	100%
user\migrations\0005_favourites_alias.py	4	0	100%
user\migrations\0006_auto_20191230_1617.py	4	0	100%
user\migrations\0007_auto_20191230_1708.py	4	0	100%
user\migrations\0008_auto_20210819_1100.py	4	0	100%
user\migrations\0009_auto_20210819_1101.py	4	0	100%
user\migrations\0010_historic_route_type.py	4	0	100%
user\migrations\0011_auto_20210821_2313.py	4	0	100%
user\migrations\0012_auto_20210824_1156.py	4	0	100%
user\migrations__init__.py	0	0	100%
user\models.py	27	1	96%
user\tests__init__.py	0	0	100%
user\tests\tests_events.py	21	0	100%
user\tests\tests_evollution_pollution.py	108	0	100%
user\tests\tests_favourites.py	0	0	100%
user\tests\tests_historic_pollution.py	83	0	100%
user\tests\tests_historic_searchs.py	0	0	100%
user\tests\tests_profile.py	0	0	100%
user\tests\tests_redirections.py	24	0	100%
user\tests\tests_search.py	258	0	100%
user\updater.py	7	0	100%
user\updaterAPI.py	15	11	27%
user\urls.py	7	0	100%
user\views.py	368	0	100%
TOTAL	1079	24	98%

Figura 4.23: Informe de cobertura para la historia de usuario 4.

Una vez creado el usuario se tiene que forzar la autenticación del usuario a través de un método que nos proporciona el objeto *Client*. A continuación se muestra un ejemplo de cómo se vería en código:

```
c = Client()
response = c.post('/user/findPlace',
    {'address': self.address, 'city': self.city, 'state':
    self.state, 'country': self.country})
self.assertEquals(302, response.status_code)
self.assertEquals("/user/login/?next=/user/findPlace", response.url)

c.force_login(self.user1, backend=None)
response = c.post('/user/findPlace',
                  {'address': self.address, 'city': self.city,
    'state': self.state, 'country': self.country})
self.assertEquals(200, response.status_code)
```

En la Figura 4.24 se puede observar el informe de cobertura con las pruebas unitarias para la edición y visualización del perfil además de la añadidura y modificación de las funcionalidades que requieren autorización. Cabe destacar que se sigue consiguiendo un 100% de cobertura en el views.py.

En la Figura 4.25 se puede observar los informes de cobertura para las historia de usuario 2 y en la Figura 4.26 se muestra la cobertura para la historia de usuario 3. En ambas se sigue manteniendo una cobertura del 100% lo cual asegura un mínimo de fiabilidad en nuestras pruebas.

4.5.2 Resumen

En este Sprint se ha realizado la última historia de usuario relacionada con las llamadas a la aplicación REST, que es la historia de usuario 4. Para poder mostrar al usuario de manera sencilla e intuitiva la evolución de la contaminación sin recargar la página completa se ha hecho uso por primera vez AJAX y poder de esta manera recargar únicamente la información del mapa de forma asíncrona.

Para desarrollar la historia de usuario 6 se ha trabajado con la clase genérica que proporciona *Django*, llamada *CreateView*, la cual se apoya en los *forms* para obtener y validar la información para el modelo.

Para las historias de usuario 7 y 8 se ha aprendido a usar la librería *django.contrib.auth.url* que proporciona *Django*. Esta librería agiliza y facilita enormemente el desarrollo para la gestión de usuarios. Además se ha modificado las funcionalidades anteriores para que diferencien entre usuarios anónimos y usuarios registrados.

En las historias de usuario 2 y 3 se ha trabajado con el objeto `Paginator` para pagi-

```
Name                                                Stmts   Miss  Cover
---------------------------------------------------------------------
aplicationweb\__init__.py                               0      0   100%
aplicationweb\settings.py                              29      0   100%
aplicationweb\urls.py                                   7      0   100%
manage.py                                              12      2    83%
user\__init__.py                                        0      0   100%
user\admin.py                                          11      0   100%
user\apps.py                                            6      0   100%
user\externals_call.py                                 25     10    60%
user\forms.py                                          14      0   100%
user\migrations\0001_initial.py                         8      0   100%
user\migrations\0002_historic.py                        6      0   100%
user\migrations\0003_auto_20191227_1702.py              4      0   100%
user\migrations\0004_favourites.py                      7      0   100%
user\migrations\0005_favourites_alias.py                4      0   100%
user\migrations\0006_auto_20191230_1617.py              4      0   100%
user\migrations\0007_auto_20191230_1708.py              4      0   100%
user\migrations\0008_auto_20210819_1100.py              4      0   100%
user\migrations\0009_auto_20210819_1101.py              4      0   100%
user\migrations\0010_historic_route_type.py             4      0   100%
user\migrations\0011_auto_20210821_2313.py              4      0   100%
user\migrations\0012_auto_20210824_1156.py              4      0   100%
user\migrations\__init__.py                             0      0   100%
user\models.py                                         27      1    96%
user\tests\__init__.py                                  0      0   100%
user\tests\tests_events.py                             21      0   100%
user\tests\tests_evollution_pollution.py              108      0   100%
user\tests\tests_favourites.py                          0      0   100%
user\tests\tests_historic_pollution.py                 83      0   100%
user\tests\tests_historic_searchs.py                    0      0   100%
user\tests\tests_profile.py                            47      0   100%
user\tests\tests_redirections.py                       24      0   100%
user\tests\tests_search.py                            258      0   100%
user\updater.py                                         7      0   100%
user\updaterAPI.py                                     15     11    27%
user\urls.py                                            9      0   100%
user\views.py                                         393      0   100%
---------------------------------------------------------------------
TOTAL                                                1153     24    98%
```

Figura 4.24: Informe de cobertura para las pruebas del perfil y las funcionalidades con autenticación.

Name	Stmts	Miss	Cover
aplicationweb__init__.py	0	0	100%
aplicationweb\settings.py	29	0	100%
aplicationweb\urls.py	7	0	100%
manage.py	12	2	83%
user__init__.py	0	0	100%
user\admin.py	11	0	100%
user\apps.py	6	0	100%
user\externals_call.py	29	13	55%
user\forms.py	14	0	100%
user\migrations\0001_initial.py	8	0	100%
user\migrations\0002_historic.py	6	0	100%
user\migrations\0003_auto_20191227_1702.py	4	0	100%
user\migrations\0004_favourites.py	7	0	100%
user\migrations\0005_favourites_alias.py	4	0	100%
user\migrations\0006_auto_20191230_1617.py	4	0	100%
user\migrations\0007_auto_20191230_1708.py	4	0	100%
user\migrations\0008_auto_20210819_1100.py	4	0	100%
user\migrations\0009_auto_20210819_1101.py	4	0	100%
user\migrations\0010_historic_route_type.py	4	0	100%
user\migrations\0011_auto_20210821_2313.py	4	0	100%
user\migrations\0012_auto_20210824_1156.py	4	0	100%
user\migrations__init__.py	0	0	100%
user\models.py	27	1	96%
user\tests__init__.py	0	0	100%
user\tests\tests_events.py	20	0	100%
user\tests\tests_evollution_pollution.py	108	0	100%
user\tests\tests_favourites.py	0	0	100%
user\tests\tests_historic_pollution.py	83	0	100%
user\tests\tests_historic_searchs.py	157	0	100%
user\tests\tests_profile.py	47	0	100%
user\tests\tests_redirections.py	24	0	100%
user\tests\tests_search.py	258	0	100%
user\updater.py	7	0	100%
user\updaterAPI.py	15	11	27%
user\urls.py	9	0	100%
user\views.py	445	0	100%
TOTAL	1365	27	98%

Figura 4.25: Informe de cobertura para la historia de usuario 2.

```
Name                                                Stmts   Miss  Cover
-----------------------------------------------------------------------
aplicationweb\__init__.py                               0      0   100%
aplicationweb\settings.py                              29      0   100%
aplicationweb\urls.py                                   7      0   100%
manage.py                                              12      2    83%
user\__init__.py                                        0      0   100%
user\admin.py                                          11      0   100%
user\apps.py                                            6      0   100%
user\externals_call.py                                 29     13    55%
user\forms.py                                          14      0   100%
user\migrations\0001_initial.py                         8      0   100%
user\migrations\0002_historic.py                        6      0   100%
user\migrations\0003_auto_20191227_1702.py              4      0   100%
user\migrations\0004_favourites.py                      7      0   100%
user\migrations\0005_favourites_alias.py                4      0   100%
user\migrations\0006_auto_20191230_1617.py              4      0   100%
user\migrations\0007_auto_20191230_1708.py              4      0   100%
user\migrations\0008_auto_20210819_1100.py              4      0   100%
user\migrations\0009_auto_20210819_1101.py              4      0   100%
user\migrations\0010_historic_route_type.py             4      0   100%
user\migrations\0011_auto_20210821_2313.py              4      0   100%
user\migrations\0012_auto_20210824_1156.py              4      0   100%
user\migrations\__init__.py                             0      0   100%
user\models.py                                         27      0   100%
user\tests\__init__.py                                  0      0   100%
user\tests\tests_events.py                             20      0   100%
user\tests\tests_evollution_pollution.py              108      0   100%
user\tests\tests_favourites.py                        248      0   100%
user\tests\tests_historic_pollution.py                 83      0   100%
user\tests\tests_historic_searchs.py                  157      0   100%
user\tests\tests_profile.py                            47      0   100%
user\tests\tests_redirections.py                       24      0   100%
user\tests\tests_search.py                            258      0   100%
user\updater.py                                         7      0   100%
user\updaterAPI.py                                     15     11    27%
user\urls.py                                            9      0   100%
user\views.py                                         576      0   100%
-----------------------------------------------------------------------
TOTAL                                                1744     26    99%
```

Figura 4.26: Informe de cobertura para la historia de usuario 3.

nar tanto el historial de búsquedas como el listado de lugares favoritos. Además de crear las funcionalidades de eliminación; y en el caso de los lugares favoritos la creación.

4.6 Sexto Sprint

Una de las funcionalidades más útiles e interesantes de *Django* es su interfaz para crear una capa de administración automática para todos los modelos de nuestra aplicación, y para la historia de usuario 11 se va a hacer uso de ella.

Para empezar, se creará un super usuario inicial con el que poder acceder a esta interfaz de administración, y para crearlo se debe ejecutar el comando *python manage.py createsuperuser*. Una vez creado el super usuario se procede a implementar la administración para nuestros modelos. Para ello, existe en el directorio de nuestra aplicación un archivo llamado `admins.py` en el que se desarrollará todo lo relacionado con la interfaz de administración.

Para la representación de un modelo en la interfaz de administración se hará uso de la clase `ModelAdmin`. En el caso de la administración de los usuarios si se hubiera usado la clase de `Users` propia de Django ahora se podría usar la clase `UserAdmin` que, como el nombre indica, es propia para la administración de usuarios; pero se ha desarrollado un usuario personalizado así que no aplica en esta situación.

Para conseguir una administración básica del modelo se necesita crear una clase que herede del `ModelAdmin` y habría que registrarla a través del método `admin.site.register()` para que se tenga en cuenta a la hora de crear toda la interfaz de administración, pero con solo esto la interfaz queda demasiado simple y poco intuitiva.

Es en este momento dónde esta interfaz de administración muestra todo su potencial, ya que ofrece una gran cantidad de opciones para personalizar como se quiera la interfaz de administración para cada modelo.

En este caso se hará uso de tres opciones en concreto: `ModelAdmin.form`, `ModelAdmin.list_filter` y `ModelAdmin.list_display`.

Por defecto se crea un `Form` para cada modelo para utilizar en la creación y edición de sus objetos, pero si se quiere utilizar un `Form` propio se puede sobrescribir el creado por defecto y asignar el personalizado gracias a `ModelAdmin.form`, como es el caso de nuestra interfaz de usuarios, a la cual se le hizo un `Form` propio en su momento.

El `ModelAdmin.list_display` permite indicar que campos del objeto que se quieran mostrar cuando se listen los objetos del modelo. Además dentro esta opción se tienen varias sub-opciones como por ejemplo: ordenar por el campo que se quiera, que se muestre algún valor por defecto ante campos vacíos o añadir una descripción a cada campo mostrado.

Por último, con `ModelAdmin.list_filter` se puede escoger diferentes campos por

los que filtrar el listado de objetos para agilizar las búsquedas de algún objeto en concreto.

Además, por defecto los objetos se listan de manera paginada con un límite de 100 objetos por página, y una vez más gracias a la gran capacidad de personalización de esta interfaz se puede modificar a nuestro gusto con métodos como `ModelAdmin.list_per_page`.

A continuación, se muestra como quedaría todo en código y en la Figura 4.27 se puede observar como se vería.

```
class CustomUserAdmin(admin.ModelAdmin):
    form = CustomUserCreationForm
    list_display = ('email', 'username', 'country')
    list_filter = ('username', 'country')

admin.site.register(CustomUser, CustomUserAdmin)
```

Tanto para la historia de usuario 9 y 10 se ha utilizado la herramienta Bootstrap, ya que es una de las herramientas más populares y utilizadas en el mundo y aporta todo lo necesario para diseñar las pantallas de manera que sean agradables e intuitivas para el usuario y nos permite personalizarlas para los dispositivos móviles.

Lo primero que se ha hecho es crear una barra de navegación común para todas las pantallas permitiéndole al usuario moverse entre las diferentes pantallas de forma cómoda. En caso de que el dispositivo de visualización sea pequeño y no se puedan mostrar de forma sencilla todos los enlaces en la barra de navegación, se crea un botón que engloba todos los enlaces y clickando en él se desplegará un menú con todos los enlaces. En la Figura 4.28 se puede observar un ejemplo de ambos casos mencionados.

A continuación se ha ido pantalla por pantalla realizando el maquetado y los cambios pertinentes para hacerlo *responsive*. A continuación se pueden observar algunos ejemplos de como han quedado las diversas funcionalidades, como por ejemplo en la Figura 4.29 se puede observar como quedaría el aviso de que hay eventos cercanos. En la Figura 4.30 se puede observar como se muestra la tabla con el historial de favoritos.

Por último, para desplegar la aplicación web se siguen los mismos pasos que en el Sprint 3 para desplegar la aplicación REST. Se lanza una instancia EC2, se instala Miniconda y se crea un entorno virtual en el que se descargarán todas las librerías necesarias para ejecutar la aplicación web.

Una vez que se tiene el entorno a punto se descarga de git nuestro proyecto y se ejecutan las pruebas para confirmar que está todo correcto. Cuando esté todo listo, se crea el `.socket` y el `.service` que estará corriendo continuamente nuestra aplicación web. Para terminar se instalará Nginx y se configurará para que las peticiones que entren al puerto 80 sean atendidas por el `.service` en el que corre nuestra aplicación.

Además, para que cualquier persona pueda acceder a la aplicación web, se configuran las reglas de entrada y salida del grupo de seguridad asignado al EC2 para que acepte cualquier

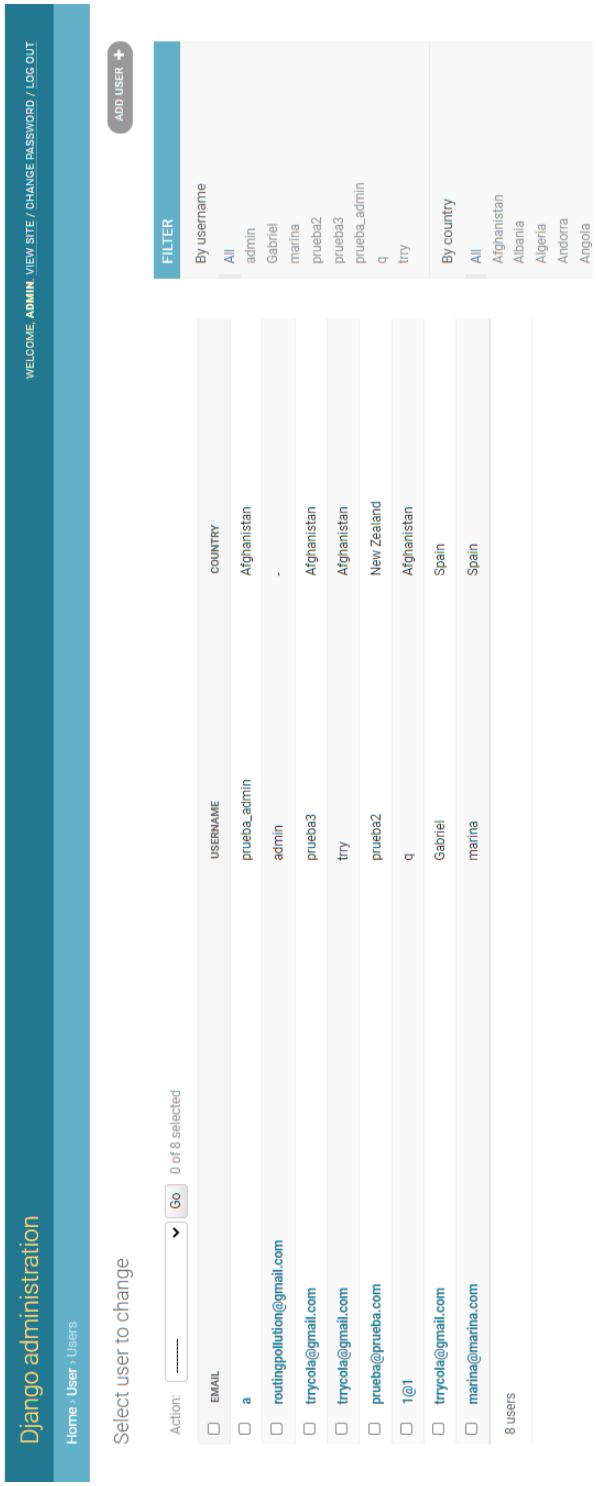
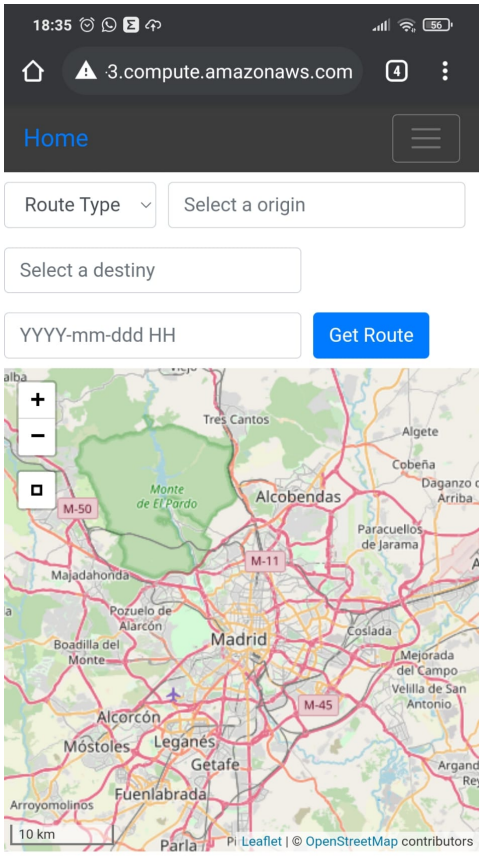
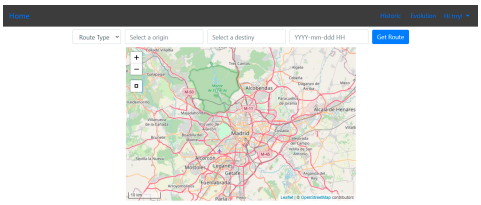


Figura 4.27: Ejemplo de interfaz de adiministración de usuarios.

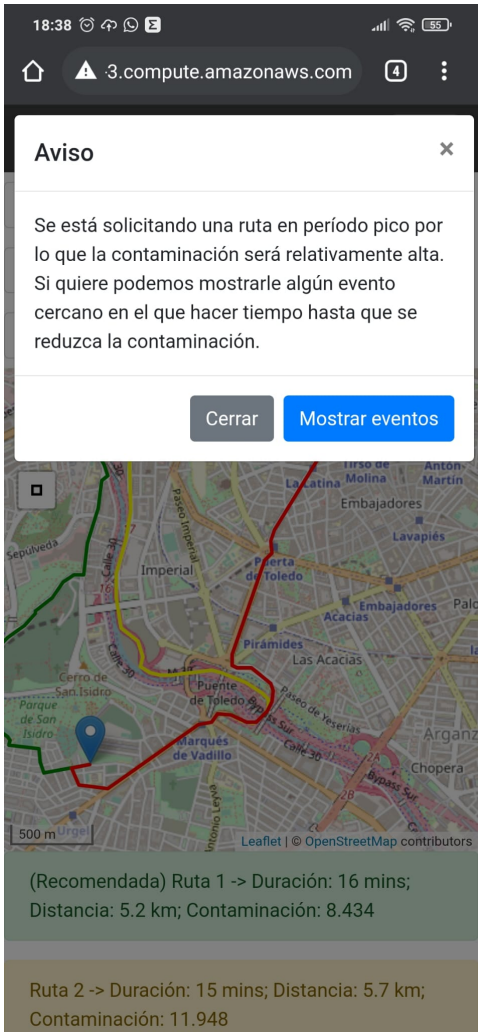


(a) Ejemplo de la barra de navegación en dispositivos pequeños.

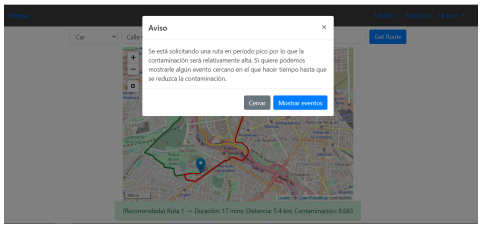


(b) Ejemplo de la barra de navegación en dispositivos grandes.

Figura 4.28: Ejemplos de la barra de navegación.

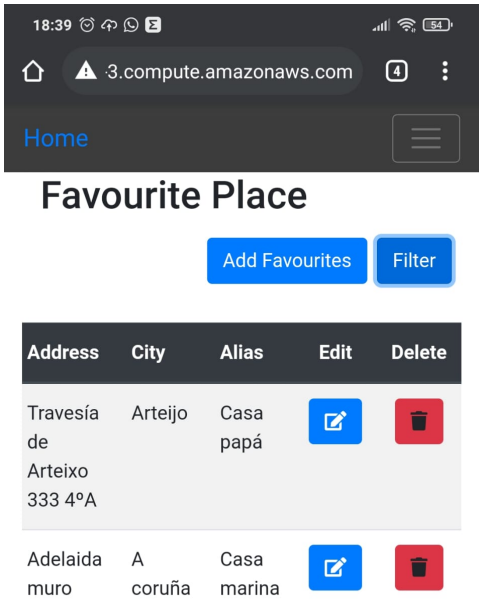


(a) Ejemplo de aviso de evento en dispositivo pequeño.

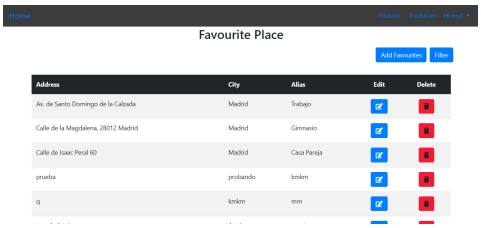


(b) Ejemplo de aviso de evento en dispositivo grande.

Figura 4.29: Ejemplos de aviso de evento.



(a) Ejemplo de historial de favoritos en dispositivo pequeño.



(b) Ejemplo de historial de favoritos en dispositivo grande.

Figura 4.30: Ejemplo de historial de favoritos.

petición HTTP o HTTPS.

Capítulo 5
Conclusiones

En este capítulo se tratarán las principales características finales del proyecto, así como todo lo aprendido en el proceso.

5.1 Objetivos Conseguidos

El objetivo final de este proyecto era poder recomendar las rutas con menor contaminación en tiempo real y futuro entre dos puntos de Madrid. Para ello había dos grandes puntos que conseguir: un modelo de aprendizaje profundo que prediga la contaminación en instantes futuros con una precisión aceptable y una aplicación web que explotase esa información para calcular las rutas con menor contaminación y mostrarlas sobre un mapa; y ambos puntos se han conseguido desarrollar e implementar de manera satisfactoria. Demostrando de esta manera estar versado en las competencias de la Ingeniería de Software, ya que se ha analizado un problema, abstraído los requisitos de usuario, planificado, diseñado e implementado una solución software, además de las correspondientes pruebas a lo implementado.

5.1.1 Lecciones aprendidas

A lo largo de este proyecto se ha aprendido a planificar y gestionar un proyecto a través de la herramienta Atlassian Jira. Manejando la herramienta no ha habido ningún problema y lo básico lo ha permitido realizar de manera excepcional, pero sí que se han detectado ciertos detalles a tener en cuenta que buscando se ha descubierto que están pendientes de mejorar. Por ejemplo, una vez terminado el Sprint se elimina su tablero Kanban y no es posible recuperarlo de ninguna manera; así que si en algún momento quieres revisar lo sucedido en Sprints anteriores no se puede mostrar su tablero correspondiente. Para este inconveniente, existe una especie de *workaround* en el que todas las *issues* tienen un campo que indica en que Sprint estuvieron involucradas y por lo tanto se podría hacer un filtro para obtener las tareas del Sprint en cuestión, pero seguirían sin mostrarse sobre el tablero Kanban.

Cabe destacar que la planificación ha ido según lo estimado y sin ninguna desviación significativa con respecto a lo esperado. En parte, esto se debe al colchón de 2 horas diarias destinado a reuniones e interrupciones varias, ya que hubo algún caso en el que no se agotaron estas 2 horas y se pudieron invertir en la tarea de turno dándo así un pequeño margen de maniobra.

También se ha aprendido a desarrollar aplicaciones web y REST en *GeoDjango*/*Django* y desplegarlas en servicios AWS. En comparación con el framework *Spring Boot* de Java, *Django* ha resultado ser todo un descubrimiento como framework para el desarrollo de aplicaciones. *Django* permite crear aplicaciones de manera más rápida y sencilla, facilita en gran medida todo el tema de gestión de usuarios y la administración de todas las clases que queramos; además de facilitar y fomentar la reutilización de código.

Por último también se ha aprendido a trabajar con herramientas relacionadas con mapas, como son la API de GoogleMaps y *Leaflet*. Como se contó en el Capítulo 2, inicialmente se iba a utilizar MapQuest principalmente debido a que permite más peticiones gratuitas que GoogeMaps, pero cuanto más se usaba más problemas se detectaban y tras varias comunicaciones con el soporte de la herramienta que acabaron de manera brusca se decide cambiar a GoogleMaps. La experiencia tanto con GoogleMaps como con *Leaflet* ha sido agradable, al contrario que con MapQuest, debido a toda la documentación y comunidad que los respalda. Si que es verdad, que al no haber trabajado nunca con mapas y coordenadas se tuvo que hacer un ligero esfuerzo para obtener los conocimientos básicos, pero una vez obtenidos todo ha seguido el curso esperado.

5.2 Trabajo Futuro

A pesar del resultado final satisfactorio aún quedan varias líneas de mejoras abiertas a futuro.

Algunas de ellas serían:

- Se podría crear un entorno escalable y de alta disponibilidad para las aplicaciones: se puede añadir un auto-balanceador para las peticiones y políticas de auto-escalado para asegurar que siempre se da servicio.
- Ampliar el rango de efectividad añadiendo estaciones de más ciudades.
- Internacionalizar el idioma.
- Permitir escoger una ruta en concreto y comenzar el viaje con seguimiento e indicaciones.
- Una pasarela de pago para poder crearse cuenta y monetizar así la aplicación.

Apéndices

Apéndice A

Manual de usuario

En este apéndice se hablará de como utilizar la aplicación REST y la aplicación web.

A.1 Aplicación REST

Para hacer uso de las funciones publicadas por la aplicación REST lo primero es llamar a la dirección IP o al nombre del dominio de la instancia EC2 en la que está desplegada nuestra aplicación REST, que en este caso es ec2-35-180-90-66.eu-west-3.compute.amazonaws.com.

A.1.1 Lectura de contaminación

La primera función que describiremos será la usada para obtener la contaminación de las estaciones en un intervalo de tiempo solicitado, conocida como `getPollutionInterval`. Esta función solo admite peticiones POST en las que en el cuerpo se debe incluir los dos siguientes parámetros:

- ”dateFrom”: fecha inicial del intervalo con formato %Y%m%d %H.
- ”dateTo”: fecha final del intervalo con formato %Y%m%d %H.

Por último, la llamada se debe realizar a la URL http://ec2-35-180-90-66.eu-west-3.compute.amazonaws.com/data/getPollutionInterval.

A.1.2 Obtención de predicciones

La siguiente función es `predictions`, esta función está disponible a través de una petición GET a la URL http://ec2-35-180-90-66.eu-west-3.compute.amazonaws.com/model/predictions/ y nos daría todos los datos de contaminación predichos para cada estación.

A.1.3 Administración

Por último se tienen dos funciones a las que solo puede acceder un usuario con rol administrador que permiten forzar el entrenamiento y la predicción de un modelo en concreto. Estas funciones serían:

- http://ec2-35-180-90-66.eu-west-3.compute.amazonaws.com/model/<int:station>/train/
- http://ec2-35-180-90-66.eu-west-3.compute.amazonaws.com/model/<int:station>/predict/

Dónde <int:station> sería el identificador de la estación en la que queremos forzar su entrenamiento o predicción.

A.2 Aplicación Web

La aplicación web está desplegada en una instancia EC2 de AWS con IP 35.180.196.18 y DNS ec2-15-237-114-113.eu-west-3.compute.amazonaws.com. Por lo que para acceder a ella solo con poner el DNS en nuestro navegador ya se nos cargará la página inicial. A partir de ahí se puede navegar por todas las funcionalidades a través de los enlaces de la aplicación sin necesidad de modificar la URL a mano.

Como usuario anónimo solo se podrán buscar rutas, en las que se debe introducir en la pantalla inicial el tipo de ruta que se va a hacer, el punto de salida y el punto de llegada, así como la hora en la que se va a realizar el viaje y la aplicación mostrará las posibles rutas y la contaminación media de cada ruta. En la Figura A.1 se puede observar un ejemplo del resultado de una búsqueda para un usuario anónimo.

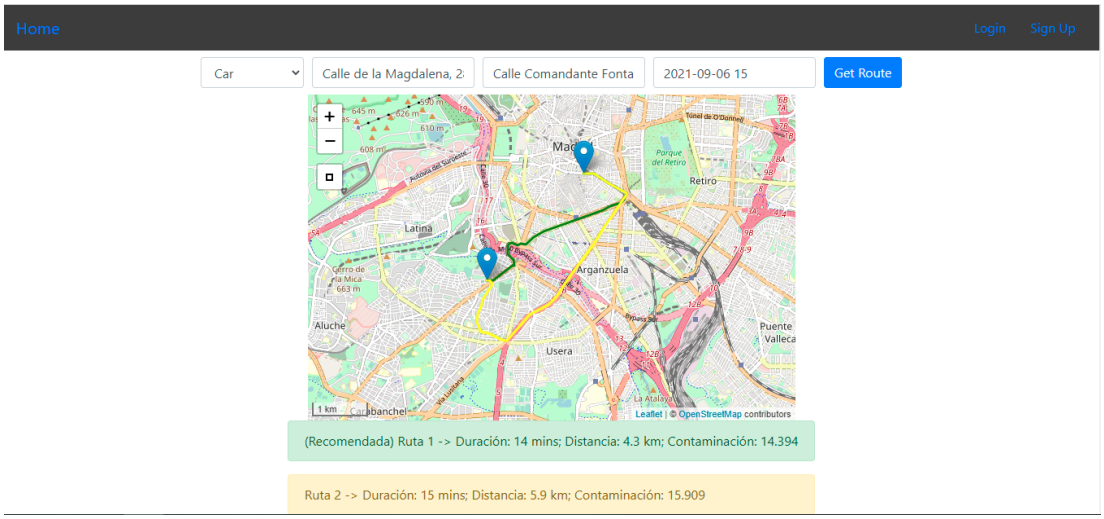


Figura A.1: Ejemplo de obtención de rutas.

Como usuario registrado, se ofrecen muchas más funcionalidades y podrá acceder a todas ellas a través de la barra de navegación fijada en la parte superior de la pantalla. Algunas de ellas son la capacidad de obtener un mapa de la contaminación de Madrid en un momento dado o en un período de tiempo. Por ejemplo, en la Figura A.2 se muestra el resultado de obtener la contaminación en un instante de tiempo concreto.

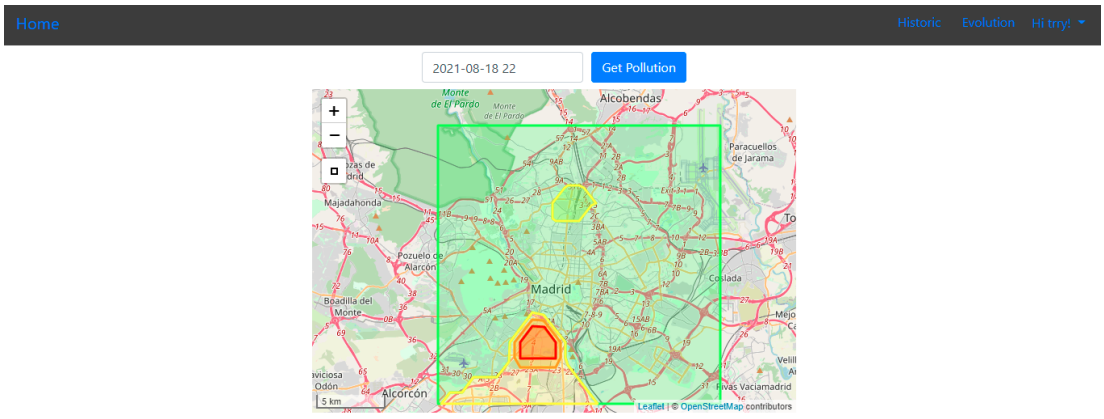
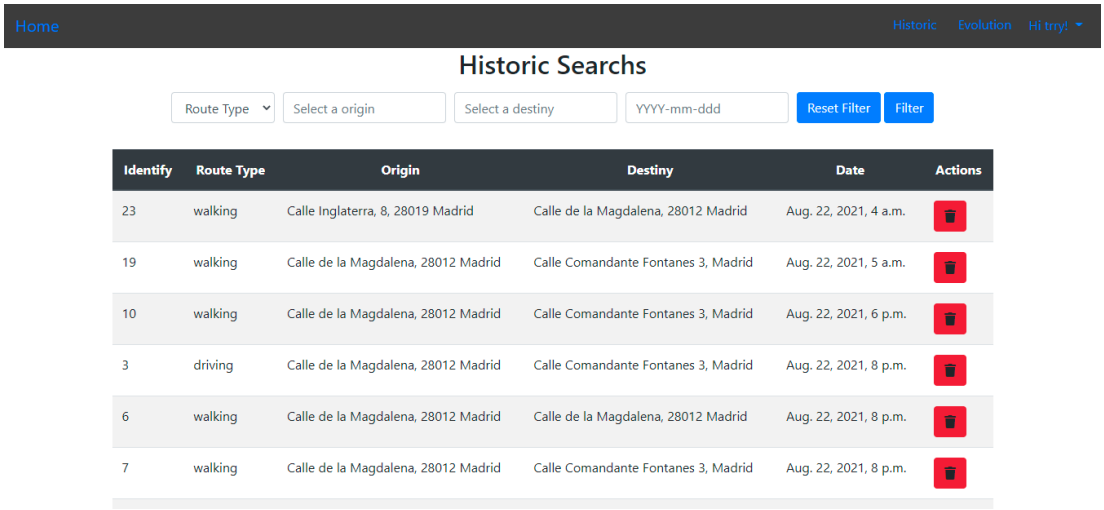


Figura A.2: Ejemplo de contaminación en un instante de tiempo concreto.

También se guarda un historial de búsquedas que se puede revisar en cualquier momento y eliminar las búsquedas que se quiera. Además, para poder encontrar una búsqueda concreta de manera sencilla se ofrece la opción de filtrar las búsquedas en función de diferentes parámetros: tipo de ruta, origen, destino y/o día del viaje. En la Figura A.3 se puede observar un ejemplo del historial de búsquedas realizadas por un usuario



Identify	Route Type	Origin	Destiny	Date	Actions
23	walking	Calle Inglaterra, 8, 28019 Madrid	Calle de la Magdalena, 28012 Madrid	Aug. 22, 2021, 4 a.m.	
19	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 5 a.m.	
10	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 6 p.m.	
3	driving	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	
6	walking	Calle de la Magdalena, 28012 Madrid	Calle de la Magdalena, 28012 Madrid	Aug. 22, 2021, 8 p.m.	
7	walking	Calle de la Magdalena, 28012 Madrid	Calle Comandante Fontanes 3, Madrid	Aug. 22, 2021, 8 p.m.	

Figura A.3: Ejemplo del historial de búsquedas.

Otra funcionalidad es la de guardar los lugares favoritos para que cuando se vaya a buscar

una ruta se muestren como sugerencia inicial los 3 lugares más utilizados, y según el usuario vaya escribiendo se irán actualizando las sugerencias con los lugares favoritos que contengan el texto introducido por el usuario. También se da la opción de visualizar un listado de los lugares favoritos y eliminar los que queramos. En la Figura A.4 se puede observar el listado de lugares Favoritos creados por un usuario.

Figura A.4: Ejemplo del listado de Favoritos.

Otra funcionalidad es que cuando se buscan rutas, si se busca una ruta en un período pico de contaminación se comprobará si hay eventos cercanos y en caso de que los haya se le comunica al usuario que tiene eventos cerca a la hora de salida por si quiere hacer tiempo hasta que pase el período pico. Un ejemplo sería lo mostrado en la Figura A.5.

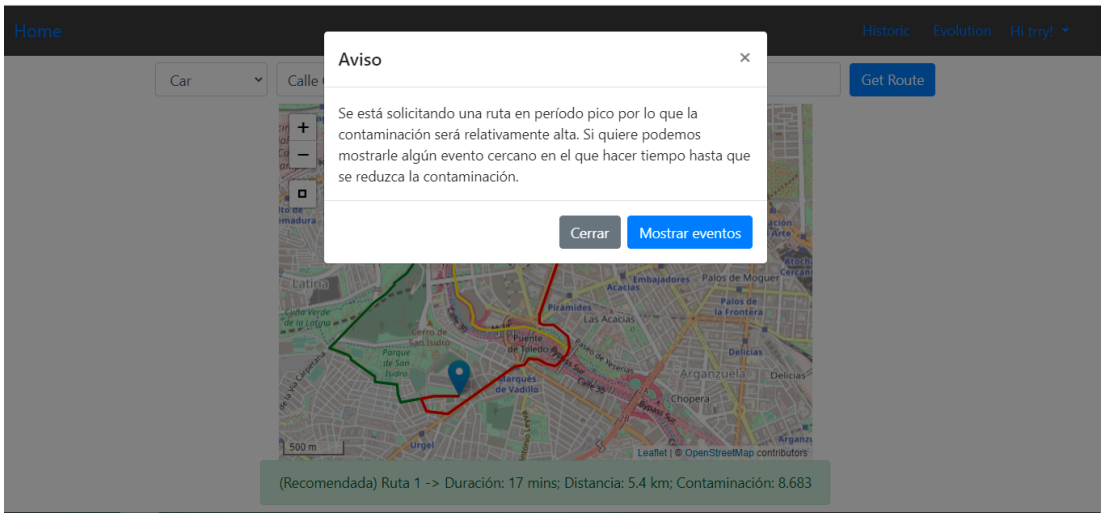


Figura A.5: Ejemplo de aviso de evento.

Por último, el usuario podrá visualizar y modificar cualquier dato de su perfil cuando quiera. En la Figura A.6 se muestra un ejemplo de ello.

Como administrador, se podrá modificar en cualquier momento cualquier información con respecto al usuario, su historial de búsquedas o sus lugares fa-

voritos. Para poder acceder a las funcionalidades de administrador hace falta que el administrador se autentique a través de la url http://ec2-15-237-114-113.eu-west-3.compute.amazonaws.com/admin/login/?next=/admin/ ya que la administración es un tema privado y por lo tanto no hay ningún enlace en las navegaciones de usuario que nos lleven a las pantallas de administración.

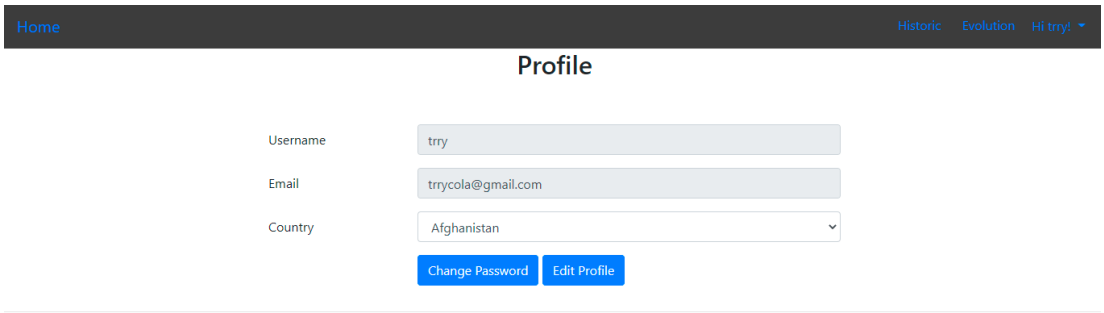


Figura A.6: Ejemplo de ver el perfil.

Lista de acrónimos

GSI *Geospatial Information Authority.*

OGC *Open Geospatial Consortium.*

KISS *Keep It Simple Silly.*

DRY *Don't Repeat Yourself.*

MVC *Model-View-Controler.*

MVT *Model-View-Template.*

REST *Representational State Transfer.*

AWS *Amazon Web Services.*

EC2 *Elastic Compute Cloud.*

RDS *Relational Database Service.*

WSGI *Web Server Gateway Interface.*

ICA *Índice de Calidad del Aire.*

RNN *Recurrent Neural Network.*

LSTM *Long Short-Term Memory.*

GRU *Gated Recurrent Unit.*

Glosario

Bytecode Código independiente de la máquina que generan compiladores de determinados lenguajes (Java, Erlang,...) y que es ejecutado por el correspondiente intérprete.

Story Points Estimación adimensional del tamaño y complejidad de una historia de usuario.

Bibliografía

[1] H. Ritchie and M. Roser, "Air pollution," 2020. [En línea]. Disponible en: https://ourworldindata.org/air-pollution

[2] ——, "Outdoor air pollution," 2020. [En línea]. Disponible en: https://ourworldindata.org/outdoor-air-pollution

[3] W. H. Organization, "Air pollution," 2012. [En línea]. Disponible en: https://www.who.int/sustainable-development/transport/health-risks/air-pollution/en/

[4] J. C. Hidalgo, "Madrid y barcelona asumen el 30 % de las muertes por contaminación del aire," 2012. [En línea]. Disponible en: https://www.efe.com/efe/espana/sociedad/madrid-y-barcelona-asumen-el-30-de-las-muertes-por-contaminacion-del-aire/10004-4066939

[5] "Leaflet." [En línea]. Disponible en: https://leafletjs.com/index.html

[6] "Openlayers." [En línea]. Disponible en: https://openlayers.org/

[7] A. Morales, "Openlayers vs leaflet ¿cuál es mejor?" 2016. [En línea]. Disponible en: https://mappinggis.com/2016/11/openlayers-vs-leaflet-mejor/

[8] "Leaflet vs openlayers," 2019. [En línea]. Disponible en: https://stackshare.io/stackups/leaflet-vs-openlayers

[9] "Mapquest." [En línea]. Disponible en: https://developer.mapquest.com/documentation/data-manager/

[10] C. de MDN, "Javascript," 2020. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/JavaScript

[11] C. de FreeCodeCamp, "Javascript." [En línea]. Disponible en: https://guide.freecodecamp.org/javascript/advantages-and-disadvantages-of-javascript/

[12] R. R. Content, "¿qué es java? conoce las particularidades de este lenguaje de programación," 2019. [En línea]. Disponible en: https://rockcontent.com/es/blog/que-es-java/

[13] "The good and the bad of java programming," 2018. [En línea]. Disponible en: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-java-programming/

[14] A. SAHOUANE, "The pros and cons of python," 2016. [En línea]. Disponible en: https://www.supinfo.com/articles/single/3425-the-pros-and-cons-of-python

[15] C. Olah, "Recurrent neural networks," 2015. [En línea]. Disponible en: https://www.luisllamas.es/machine-learning-con-tensorflow-y-keras-en-python/

[16] "Todo lo que necesitas saber sobre tensorflow, la plataforma para inteligencia artificial de google," 2019. [En línea]. Disponible en: https://puentesdigitales.com/2018/02/14/todo-lo-que-necesitas-saber-sobre-tensorflow-la-plataforma-para-inteligencia-arti\protect\discretionary{\char\hyphenchar\font}{}{}ficial-de-google/#:~:text=TensorFlow%20es%20una%20biblioteca%20de,(tensores)%20comunicadas%20entre%20ellos.

[17] J. Torres, "Todo lo que necesitas saber sobre tensorflow, la plataforma para inteligencia artificial de google." [En línea]. Disponible en: https://torres.ai/deep-learning-inteligencia-artificial-keras/

[18] [En línea]. Disponible en: https://www.djangoproject.com/

[19] "Django - descripción general." [En línea]. Disponible en: https://www.tutorialspoint.com/django/django_overview.htm

[20] [En línea]. Disponible en: https://docs.djangoproject.com/en/3.0/ref/contrib/gis/tutorial/#introduction

[21] "Metodología ágil," 2014. [En línea]. Disponible en: https://www.ecured.cu/Metodolog%C3%ADa_%C3%A1gil

[22] V. R. Villán, "Las metodologías ágiles más utilizadas y sus ventajas dentro de la empresa," 2019. [En línea]. Disponible en: https://www.iebschool.com/blog/que-son-metodologias-agiles-agile-scrum/

[23] "El manifiesto ágil," 2014. [En línea]. Disponible en: https://www.scrummanager.net/bok/index.php?title=El_manifiesto_%C3%A1gil

[24] [En línea]. Disponible en: https://es.indeed.com/career/programador-junior/salaries/Galicia

[25] [En línea]. Disponible en: https://www.hackaboss.com/blog/salario-programador-espana