UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería de Computadores

Planificación de rutas para patinetes eléctricos

Route planning for electric scooters

Realizado por
Meead Meyfour Asadi

Tutorizado por
Gabriel Jesús Luque Polo
José Francisco Chicano García

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2023

E. T. S. DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

Planificación de rutas para patinetes eléctricos
Route planning for electric scooters

Realizado por
Meead Meyfour Asadi
Tutorizado por
Gabriel Jesús Luque Polo
Cotutorizado por
José Francisco Chicano García
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, 06/09/2023

Agradecimientos

Agradezco a mis padres que, a pesar de que nuestra situación no era la mejor, siempre antepusieron mis estudios y confiaron plenamente en que en algún momento los acabaría.

Agradezco a mis tutores Gabriel y Francisco por no tirar la toalla conmigo y seguir apoyándome en este proyecto a pesar de mis tardíos y escasos avances.

Agradezco a mi gran amigo Daniel Ruiz por ser solidario y no abandonarme en estos 8 hermosos años de carrera juntos.

Agradezco a mi buen amigo Pedro Miguel por encontrarse un balón de voleibol en su jardín y darle un giro inesperado a mi vida.

Por último agradezco a todos mis compañeros caídos en esta dura batalla, esto va por vosotros.

Resumen:

Los sistemas de navegación son usados hoy en día a gran escala en todo el mundo para recorrer las calles. Esto se debe al gran tiempo que se puede ahorrar navegando por las calles de la forma más eficiente posible, teniendo en cuenta diversos factores como el tráfico, la velocidad permitida en la vía y el ahorro de combustible. Estos factores son muy importantes para los conductores ya que les permiten tomar la mejor ruta con respecto a su necesidad. Pero ninguno de estos factores es realmente importante para el Patinete Eléctrico. Este vehículo es utilizado principalmente por jóvenes, que desconocen las normas viales y a menudo tienden a ser imprudentes. Para este grupo de usuarios los factores de los principales sistemas de navegación no sirven ya que no se les puede garantizar la seguridad de las vías por las que transitan. Para solucionar este problema, hemos desarrollado una aplicación Android que planea rutas inteligentes para el usuario teniendo en cuenta sus necesidades en cuanto a seguridad y tiempo. De esta forma la aplicación recopila la información de rutas navegadas de los usuarios y, en base a esa información, consulta un servidor en Python que, mediante el uso de aprendizaje automático, mejora los tiempos estimados de las rutas.

Palabras claves: Android, Python, Recomendación de rutas, aprendizaje automático

---

Abstract:

Nowadays navigation systems are used on a large scale all over the world to navigate the streets. This is due to the large amount of time that can be saved by navigating the streets in the most efficient way possible, taking into account various factors such as traffic, speed limits and fuel saving. These factors are very important to drivers as it allows them to take the best route with respect to their need. But none of these factors are really important for the Electric Scooters. This vehicle is mainly used by young people, who are unaware of road rules and often tend to be reckless. For this group of users the factors of the main navigation systems are not very helpful as their safety on the roads cannot be guaranteed. To solve this problem, we have developed an Android application that plans intelligent routes for the user taking into account their needs in terms of safety and time. In this way the application collects the information of navigated routes from the users and, based on that information, it consults a Python server that, through the use of Machine Learning, improves the estimated times of the routes.

Keywords: Android, Python, Route Recommendation, Machine Learning

Índice de contenidos

CAPÍTULO 1

Introducción

1.1. Motivación

En la actualidad los patinetes eléctricos se han convertido en uno de los principales vehículos de movilidad personal en España, contando con más de 800.000 patinetes en circulación [1], debido a su eficiencia y versatilidad. Este vehículo es principalmente utilizado por jóvenes de entre 18 y 29 años quienes prefieren esta opción frente a la imposibilidad de comprar un coche debido a su alto precio. Además, existen empresas dedicadas al alquiler temporal de patinetes los cuales aumentan todavía más su popularidad y facilitan el acceso a este vehículo [2].

Para moverse por las calles estos usuarios utilizan sistemas de navegación como Google Maps [3] que si bien es capaz de medir precisamente el tiempo que tarda un coche o una bicicleta teniendo en cuenta el tráfico, el estado de los carriles y sus normativas, no es capaz de generar rutas específicas para patinetes que garanticen su seguridad y el cumplimiento de sus normativas en cuanto a velocidad y vías permitidas.

Como consecuencia surge la idea de desarrollar una aplicación capaz de garantizar la seguridad de estos usuarios respetando las normativas específicas, mostrándoles el porcentaje de carriles seguros que componen la ruta diseñada y capaz de adaptarse a sus necesidades en cuanto a tiempo o seguridad.

1.2. Objetivos

Atendiendo a la idea indicada en la sección previa, el objetivo principal de este trabajo de fin de grado será el desarrollo de una aplicación móvil que recomiende rutas de viajes para usuarios de patinetes. Esta aplicación sería capaz de generar rutas que tengan en cuenta los carriles específicos para este tipo de vehículos, otorgaría al usuario la capacidad de elegir cómo de segura o rápida quiere que sea su ruta y generar una ruta acorde a los requerimientos de este, siendo las rutas más seguras las que utilizan mayor número de carriles bici o carriles mixtos. Además, recopilaría los datos de los usuarios para después aplicar técnicas de aprendizaje automático para aprender y ser capaz de estimar el tiempo específico que se tarda en recorrer un tramo en la fecha y hora correspondiente y utilizar esta información a la hora de generar las rutas para recomendar el mejor camino.

Para lograr nuestro objetivo, la aplicación móvil generará sesiones únicas asociadas a cada uso de los usuarios y recopilará los datos de estas, tales como el lugar geográfico en el que se encuentra, la fecha y la hora y los enviará al servidor junto con su id de la sesión, que los guardará como `datos no procesados`. Estos datos serán luego procesados por el servidor y utilizados para entrenar modelos de aprendizaje automático.

Existen 5 hitos clave en este proyecto:

- Crear la aplicación móvil.
- Crear el servidor.
- Crear una base de datos.
- Entrenar el modelo de aprendizaje automático.
- Crear una consulta capaz de utilizar el modelo para proponer una ruta.

1.3. Estado del arte

Una vez definidos los objetivos del TFG vamos a buscar las aplicaciones que existen en el mercado y qué funcionalidades aporta nuestro desarrollo sobre los existentes.

Existen cuatro aplicaciones principales de navegación que componen el 98 % de todo el mercado de este sector, siendo Google Maps [3] el más utilizado con un 67 % de uso [4]. Los usuarios prefieren esta aplicación sobre las rivales debido a que proporciona direcciones más claras, tiene mejores características, es más fácil de utilizar, proporciona direcciones más entendibles para gente no conductora o conductores no habituales y, por último, porque nunca han utilizado otra aplicación. La siguiente aplicación más utilizada es Waze

[5], esta aplicación no es muy diferente a las demás, se distingue principalmente debido a que tiene una comunidad muy activa y muy dedicada que indica muy rápido sobre accidentes y radares móviles. Apple Maps [6] es la siguiente aplicación en la lista, la principal razón de uso de esta aplicación es que es nativa de los móviles de Apple, lo cual facilita su uso. La última aplicación de la lista es MapQuest [7] que si bien no ofrece nada nuevo, lleva en el mercado tanto tiempo como Google Maps e incluye el coste estimado de gasolina de los trayectos [8]. Si bien estas aplicaciones ofrecen rutas muy variadas y estudiadas para los coches y hasta para bicicletas, ninguna de ellas aborda el problema que suponen los patinetes eléctricos.

Con respecto a este tema encontramos una singular aplicación llamada ScootRoute [9], una pequeña aplicación dedicada a la navegación para micro movilidad. Esta aplicación es capaz de generar rutas teniendo en cuenta carriles bici, tiene navegación por voz y respeta los limites de velocidad para este tipo de vehículos. Sin embargo, a diferencia de nuestra aplicación, no tiene en cuenta la normativa española, pues está diseñada para ser utilizada en Estados Unidos con medidas tales como millas por hora para la velocidad. Además, la clave de nuestra aplicación consiste en el uso del aprendizaje automático para analizar datos históricos y calcular la duración estimada de la ruta, que esta aplicación tampoco hace.

1.4. Tecnologías utilizadas

Como ya hemos definido los objetivos y estudiado el mercado actual, ahora definimos las tecnologías a utilizar para el proyecto.

1.4.1. Aplicación Móvil

Con respecto a crear una aplicación móvil, se ha optado por desarrollarla para el sistema Android [10], debido a que es muy accesible y más fácil de tratar. Para ello, existen dos vías principales de desarrollo. La primera es mediante aplicaciones nativas usando Java [11] o Kotlin [12] y la segunda es mediante tecnologías web adaptadas a móviles. Se ha optado por la primera debido al rendimiento y porque era un objetivo personal aprender esta tecnología en concreto. Además, se podía utilizar Java [11] para el desarrollo, lenguaje que hemos estudiado en la carrera, con el cual estamos muy familiarizados y es multi-plataforma, que hace que el código sea el mismo en cualquier parte, lo que hace que consultar información sea muy fácil y toda la experiencia adquirida sirva para futuros proyectos. Como recurso fundamental se ha utilizado la biblioteca Osmdroid [13] que, además de permitir a la aplicación acceder a funcionalidades de OpenStreetMap [14], ha sido la encargada de mostrar el mapa sobre la aplicación y de dibujar las rutas en ella.

Como entorno de desarrollo se ha optado por Android Studio [15] debido a 3 claves que hacen que destaque sobre toda la competencia. La primera pieza clave es el hecho de poder desarrollar visualmente en su editor de diseño, dando la posibilidad de ver y modificar en tiempo real como se ven los componentes en la aplicación. La segunda gran diferencia reside en que Android Studio utiliza Módulos en lugar de espacios de trabajo para la organización de la estructura del proyecto, esto hace que sea mucho más fácil de manejar. Por último tenemos la emulación, es realmente sencillo e intuitivo crear una instancia de un dispositivo virtual e instalar nuestra aplicación en él para realizar pruebas [16].

1.4.2. Servidor

Existen diversas tecnologías para la creación del servidor, sin embargo, se ha optado por Python [17] debido a su facilidad de aprendizaje y uso, su manejo de mapas, su rápida inclusión de técnicas de Machine Learning y su eficacia para crear un API REST. Para la creación del API REST se ha optado por usar el framework FastAPI [18] ya que es muy rápido de implementar, muy intuitivo y rinde extremadamente bien. Para el modelo de Machine Learning vamos a utilizar la biblioteca Scikit-Learn [19] que nos permite acceder a un gran número de modelos de forma rápida e intuitiva. En la parte del servidor también necesitábamos conectar con OpenStreetMap [14] para acceder a sus funciones, por lo que se ha utilizado la biblioteca OSMNX [20]. Esta biblioteca es capaz de hacer cosas como localizar el nodo más cercano a una latitud y longitud dada o crear rutas utilizando el algoritmo de Dijkstra con opción a cambiar la función para calcular los pesos, del cual hablaremos más tarde. Como entorno de desarrollo vamos a utilizar PyCharm [21] ya que es más accesible para el usuario nuevo y no es necesario configurar nada para empezar a programar.

1.4.3. Base de datos

Como base de datos se utilizará PostgreSQL [22] que, junto a su aplicación de escritorio pgAdmin [23], es muy cómodo de utilizar, cuenta con una interfaz gráfica y puede manejar datos geoposicionados.

1.4.4. Creación de documentos

Para la creación de este documento se ha optado por utilizar LaTeX[24] mediante el editor de texto en línea Overleaf [25], que permiten escribir documentos de muy alta calidad de forma online, con una organización impecable y con diversos recursos que facilitan la generación de referencias.

1.5. Metodología

Este proyecto ha seguido una metodología Scrum [26], que se basa en aplicar un conjunto de buenas prácticas con el equipo de manera regular con el objetivo de obtener el mejor resultado posible en un proyecto. Para lograr esto, se realizan entregas parciales de forma regular, se evalúan y los requisitos pueden ir cambiando a medida que avanza el proyecto. De esta forma, el proyecto puede expandirse o cambiar dependiendo de ello. Esta metodología es muy útil en situaciones como la nuestra donde los requisitos están poco definidos y a medida que avanzamos en el proyecto podemos optar por realizar cambios en algunas funcionalidades.

En esta metodología se realizan reuniones en intervalos fijos en nuestro de caso de 1 mes, conocidos como sprints, en los cuales se debe haber cumplido un objetivo del resultado final. Al finalizar estos sprints se realiza una reunión con el Product Owner, en este caso los tutores, para que evalúen el desempeño realizado y se planifican los objetivos para el siguiente sprint. Usualmente durante un Sprint se realizan reuniones diarias de 15 minutos, pero en el caso de este proyecto hemos obviado esa parte debido a que solo contamos con un desarrollador.

Durante el primer sprint de este proyecto se ha realizado la especificación de los requisitos y la investigación sobre las herramientas a utilizar para la creación de la aplicación. Durante el segundo sprint y los consiguientes hasta el penúltimo se han ido realizando el modelado de la aplicación, el desarrollo del código y las pruebas respectivas a ello. Para finalizar, en el último sprint se han realizado las últimas pruebas de integración y sistema y se ha realizado la memoria y el manual de usuario.

1.6. Estructura de la memoria

La memoria se divide en los siguientes apartados:

- **Introducción:** Es el presente capítulo donde se ha motivado y descrito el objetivo de este TFG. También se ha descrito la metodología y tecnologías utilizadas.
- **Diseño y especificación:** Se describe la lógica de las aplicaciones sin entrar en mucho detalle y se presentan sus clases, interfaces, requisitos y casos de uso.
- **Desarrollo frontend:** Se abordará la parte más técnica del proyecto relacionada con la aplicación móvil y sus funciones principales.

- **Desarrollo backend:** Este apartado compone la parte técnica del servidor, sus métodos principales, cómo se aplica el aprendizaje automático y su conexión con la base de datos.
- **Pruebas:** Para comprobar el correcto funcionamiento de las diversas aplicaciones se han realizado pruebas de diversa índole.
- **Conclusiones y trabajos futuros:** Se estudian los resultados obtenidos y se teoriza sobre futuras mejoras para el proyecto
- **Manual de usuario:** Este anexo describe de forma visual y cómoda para el usuario cómo utilizar la aplicación móvil desarrollada en el proyecto

CAPÍTULO 2

Diseño y especificación

Este segundo capítulo aborda los aspectos fundamentales de la aplicación móvil y del servidor en cuanto a diseño y especificación. En el primer Apartado 2.1 se describe de forma general la estructura utilizada. En el segundo Apartado 2.3 encontraremos el modelo de datos en el cual se detallan el tipo de datos que hay en la base y la forma en la que se relacionan. En el tercer Apartado 2.2 observamos el diagrama de uso de la aplicación, en el cual se detallan las principales funcionalidades que tiene. En el último Apartado 2.4 se detallan los elementos que componen la interfaz y su utilidad.

2.1. Arquitectura

La arquitectura que utilizamos se basa en el modelo Cliente/Servidor (ver Figura 2.1). En esta arquitectura, el sistema se divide en dos componentes principales: el cliente y el servidor. El cliente es la parte de la aplicación con la que los usuarios interactúan directamente, mientras que el servidor es responsable de procesar las solicitudes del cliente, gestionar los datos y proporcionar respuestas.

El cliente es la aplicación Android que se ejecuta en el dispositivo móvil. Su función principal es interactuar con el usuario y proporcionar una interfaz de usuario amigable. Cuando el usuario realiza una acción en la aplicación, como solicitar la generación de una ruta, el cliente empaqueta esa solicitud y la envía al servidor.

El servidor es precisamente nuestro servidor Python. Su función es recibir, procesar y responder a las solicitudes del cliente. En nuestro caso, el servidor implementa una API REST en Python que permite la comunicación entre el cliente y el servidor a través de solicitudes HTTP, utilizando los métodos GET y POST. El servidor además se conecta a una base de datos para almacenar y recuperar información. Esto es importante para guardar datos de uso de la aplicación y cualquier otra información relevante.

Tanto el cliente como el servidor se conectan a la API de OpenStreetMap para acceder a funcionalidades que esta contiene, como pueden ser mostrar el mapa u obtener información de un carril.

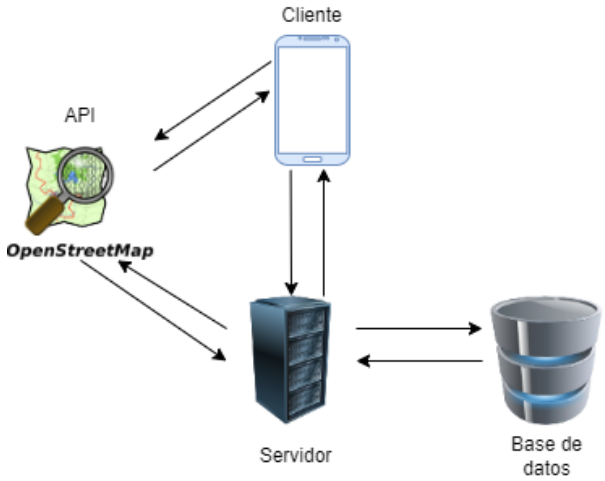


Figura 2.1: Diagrama de arquitectura

2.2. Requisitos y casos de uso

En este apartado se definen los requisitos que debe cumplir la aplicación y sus correspondientes casos de uso.

2.2.1. Requisitos funcionales

- RF-USUARIO-01: La aplicación debe permitir al usuario interactuar con el mapa.
- RF-USUARIO-02: La aplicación debe permitir al usuario elegir un objetivo para crear la ruta.
- RF-USUARIO-03: La aplicación debe permitir al usuario cambiar las preferencias en cuanto a seguridad y tiempo y generar una nueva ruta.
- RF-USUARIO-04: La aplicación debe permitir al usuario cancelar la ruta escogida.
- RF-APP-01: La aplicación debe mostrar el mapa correspondiente al usuario por pantalla.
- RF-APP-02: La aplicación debe conectarse al servidor y solicitar la ruta escogida por el usuario y mostrarla por pantalla.
- RF-APP-03: La aplicación debe recopilar de forma regular la ubicación del usuario y enviarla al servidor.

- RF-SERVIDOR-01: El servidor debe conectarse con la base de datos para almacenar y obtener los datos de los usuarios y los modelos.
- RF-SERVIDOR-02: El servidor debe entrenar un modelo de aprendizaje automático para cada par de nodos que los usuarios recorran.
- RF-SERVIDOR-03: El servidor debe realizar estimaciones de tiempo utilizando los modelos de aprendizaje automático.
- RF-SERVIDOR-04: El servidor debe generar rutas teniendo en cuenta las preferencias de seguridad del usuario mediante un algoritmo de Dijkstra que tiene en cuenta los modelos existentes de aprendizaje automático.

2.2.2. Requisitos no funcionales

- RNF-01: La aplicación y el servidor deben estar siempre conectados a la API de OpenStreetMap
- RNF-02: La aplicación debe tener una interfaz simple e intuitiva.
- RNF-03: El modelo de aprendizaje automático escogido debe escalar correctamente con grandes cantidades de datos.

2.2.3. Casos de uso

Nombre	**Descripción**
Caso de uso	CU-01: Interacción con el mapa
Actores	Usuario
Descripción	El usuario interactuará con el mapa.
Precondiciones	El usuario aún no ha interactuado con el mapa.
Requisito	RF-USUARIO-01
Escenario principal	1. El usuario desliza la pantalla para moverse a través de ella. 2. El usuario pellizca la pantalla para acercar o alejar la vista. 3. El usuario presiona los botones de la parte inferior de la pantalla para acercar o alejar la vista.
Escenario alternativo	No hay escenario alternativo.

Tabla 2.1: Caso de uso: Interacción con el mapa

Nombre	**Descripción**
Caso de uso	CU-02: Selección de destino
Actores	Usuario
Descripción	El usuario elige un destino para generar la ruta.
Precondiciones	No se está mostrando una ruta en pantalla.
Requisito	RF-USUARIO-02
Escenario principal	1. El usuario presiona en la pantalla en la ubicación escogida como destino. 2. La pantalla cambia y se muestra en el mapa una ruta que conecta al usuario con su destino.
Escenario alternativo	No existe ninguna ruta por la cual alcanzar el destino con un patinete y el usuario se mantiene en la primera pantalla.

Tabla 2.2: Caso de uso: Selección de destino

Nombre	**Descripción**
Caso de uso	CU-03: Cambio de preferencias
Actores	Usuario
Descripción	Cuando el usuario dispone de una ruta en la pantalla puede cambiar las preferencias de esta.
Precondiciones	El usuario debe disponer de una ruta en la pantalla.
Requisito	RF-USUARIO-03
Escenario principal	1. El usuario utiliza el slider para elegir el grado de seguridad que desea tener en la ruta. 2. El usuario presiona sobre el botón de volver a generar la ruta.
Escenario alternativo	No hay escenario alternativo.

Tabla 2.3: Caso de uso: Cambio de preferencias

Nombre	**Descripción**
Caso de uso	CU-04: Cancelación del viaje
Actores	Usuario
Descripción	El usuario tiene la opción de cancelar el viaje actual.
Precondiciones	El usuario debe disponer de una ruta en la pantalla.
Requisito	RF-USUARIO-04
Escenario principal	1. El usuario pulsa sobre el botón para cancelar el viaje. 2. La ruta actual desaparecerá y el usuario volverá a la pantalla anterior.
Escenario alternativo	No hay escenarios alternativos.

Tabla 2.4: Caso de uso: Cancelación del viaje

Nombre	**Descripción**
Caso de uso	CU-05: Visualización del mapa
Actores	Usuario
Descripción	Al abrir la aplicación se debe mostrar el mapa circundante al usuario.
Precondiciones	Se debe tener una conexión a internet.
Requisito	RF-APP-01
Escenario principal	1. Al abrir la aplicación se obtendrá la ubicación del usuario y se solicitará a la API de OpenStreetMap los datos del mapa. 2. OpenStreetMap devolverá los datos del mapa y se mostrarán por pantalla.
Escenario alternativo	1. El usuario rechaza la solicitud de obtención de su ubicación, en cuyo caso se cerrará la aplicación. 2. No se podrá realizar la conexión con OpenStreetMap para obtener los datos, en cuyo caso se cerrará la aplicación.

Tabla 2.5: Caso de uso: Visualización del mapa

Nombre	**Descripción**
Caso de uso	CU-06: Visualización de la ruta
Actores	Usuario
Descripción	Cuando el usuario solicita una ruta, la aplicación la solicitará al servidor y la mostrará por pantalla.
Precondiciones	1. Debe existir una conexión a internet. 2. El usuario debe solicitar una ruta.
Requisito	RF-APP-02
Escenario principal	1. La aplicación manda una petición POST de HTTP al servidor con los datos de ubicación de origen, destino y nivel de seguridad. 2. La aplicación obtiene una respuesta del servidor y muestra la ruta por pantalla.
Escenario alternativo	1. La ruta no es accesible mediante un patinete, en cuyo caso el servidor devolverá un error a la aplicación. 2. El servidor no responde, en cuyo caso se ignora la petición.

Tabla 2.6: Caso de uso: Visualización de la ruta

Nombre	**Descripción**
Caso de uso	CU-07: Recopilación de datos
Actores	Usuario
Descripción	La aplicación mandará al servidor la ubicación del usuario de forma periódica mientras el usuario recorre la ruta.
Precondiciones	1. Debe existir una conexión a internet. 2. El usuario debe disponer de una ruta.
Requisito	RF-APP-03
Escenario principal	Cada 30 segundos la aplicación manda una petición POST de HTTP al servidor con los datos geográficos del usuario.
Escenario alternativo	El servidor no responde, en cuyo caso se ignora la petición.

Tabla 2.7: Caso de uso: Recopilación de datos

Nombre	**Descripción**
Caso de uso	CU-08: Conexión con base de datos
Actores	Servidor
Descripción	El servidor realizará una conexión con la base de datos al arrancar.
Precondiciones	Ninguna
Requisito	RF-SERVIDOR-01
Escenario principal	Al arrancar el servidor se realizará una conexión con la base de datos y se mantendrá activa hasta su detención.
Escenario alternativo	Ninguna

Tabla 2.8: Caso de uso: Conexión con base de datos

Nombre	**Descripción**
Caso de uso	CU-09: Entrenamiento del modelo
Actores	Servidor
Descripción	El servidor se encargará de crear y entrenar modelos de aprendizaje automático para cada par de nodos que procese de los datos del usuario.
Precondiciones	Deben existir datos geográficos del usuario en nuestra base de datos.
Requisito	RF-SERVIDOR-02
Escenario principal	1. El servidor descargará de la base de datos los datos geográficos con sus respectivos tiempos de los usuarios. 2. Comprobará si para cada par de nodos que componen una ruta existe un modelo entrenado. 3. Para los pares que no tengan un modelo, creará un modelo y lo entrenará con esos datos. 4. Para los que sí exista un modelo, lo re-entrenará con estos nuevos datos.
Escenario alternativo	No hay escenario alternativo.

Tabla 2.9: Caso de uso: Entrenamiento del modelo

Nombre	**Descripción**
Caso de uso	CU-10: Estimación de tiempos
Actores	Servidor
Descripción	El servidor utilizará utilizará el modelo asociado a un par de nodos para estimar el tiempo que se tarda en recorrerlo según los datos de entrada.
Precondiciones	Existe un modelo entrenado para el par de nodos que queremos estimar.
Requisito	RF-SERVER-03
Escenario principal	1. Se obtendrá de la memoria el modelo asociado al par de nodos cuyo tiempo queremos estimar. 2. Se realizará la estimación utilizando la función `predict` del modelo de aprendizaje automático utilizando como datos de entrada la fecha y hora actuales.
Escenario alternativo	No hay escenario alternativo

Tabla 2.10: Caso de uso: Estimación de tiempos

Nombre	**Descripción**
Caso de uso	CU-11: Generación de rutas
Actores	Servidor
Descripción	El servidor generará una ruta mediante un algoritmo de Dijkstra con una función personalizada para asignar los pesos a esta.
Precondiciones	El usuario debe solicitar la generación de una ruta.
Requisito	RF-SERVIDOR-04
Escenario principal	1. El servidor llamará al algoritmo con una ubicación de origen, una ubicación destino, la fecha actual y el nivel de seguridad exigido por el usuario. 2. El algoritmo buscará el mejor camino mediante el uso de una función personalizada para asignar los pesos. 3. Esta función comprobará si existe un modelo entrenado para los pares de nodos que estudie y utilizará una estimación del tiempo que se tarda en recorrerlo con su modelo o de forma manual si no tiene modelo y multiplicará este valor por una constante según el tipo de carril que sea y le asignará ese valor como peso. 4. El algoritmo devolverá la ruta con menor peso según las preferencias del usuario.
Escenario alternativo	El destino solicitado es imposible de acceder mediante un patinete eléctrico, en cuyo caso la función devolverá un error.

Tabla 2.11: Caso de uso: Generación de rutas

2.3. Modelo de datos

Con respecto a la base de datos que comentábamos anteriormente, se trata de una base de datos relacional que contiene una serie de tablas que permiten el almacenamiento de los datos no procesados y procesados de nuestro programa, además de albergar la última versión de nuestros modelos de aprendizaje automático.

Con la utilidad ERD Tool de pgAdmin hemos obtenido la Figura 2.2, que muestra las tablas y las clases de nuestra base de datos.

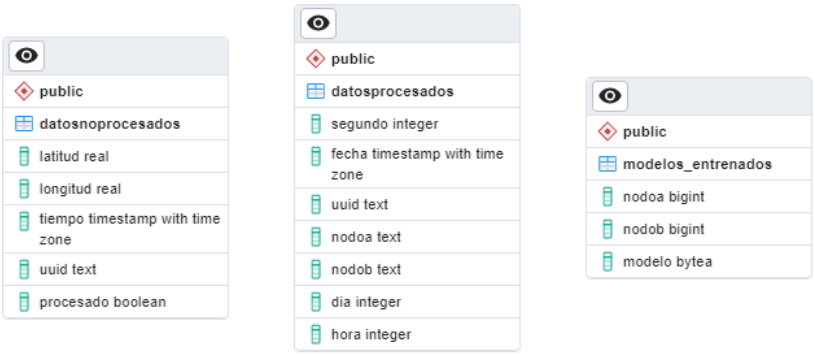


Figura 2.2: Modelo de datos

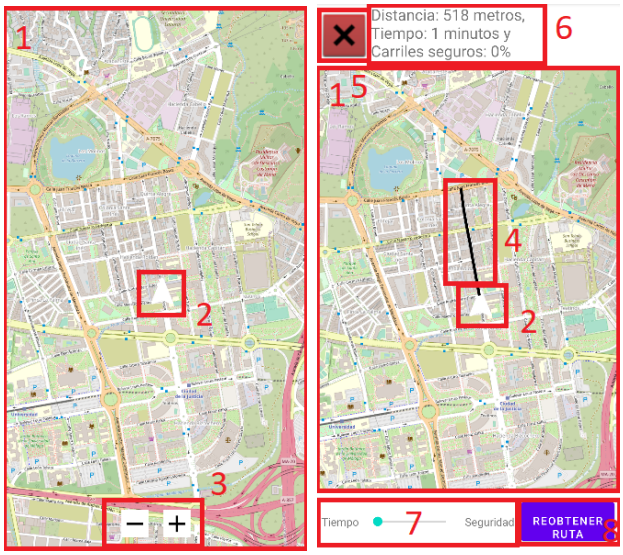
A continuación se describen cada una de las tablas y su utilidad:

- La tabla **datosnoprocesados** representa los datos tal cual se obtienen del usuario, sin ningún tipo de procesamiento adicional. Esta tabla está compuesta por una `latitud` y una `longitud` que son valores numéricos reales, el momento temporal en el cual se obtuvieron los datos en formato de fecha con zona horaria, un `uuid` por el cual se agrupan todos los datos que pertenecen a la misma sesión en formato de texto y un booleano para conservar los datos para un histórico sin riesgo de que estos vuelvan a ser procesados en el futuro.
- La tabla **datosprocesados** alberga los datos una vez han sido procesados por el servidor. Esta tabla se compone de un campo `uuid` en formato texto que alberga la misma id mencionada anteriormente, dos campos de texto `nodoa` y `nodob` que contienen la id que le asigna OpenStreetMap a ese lugar geográfico, junto a estos un campo numérico en formato entero que alberga los segundos que se ha tardado en recorrer del nodo A al nodo B. Además, para el entrenamiento del modelo, tenemos la fecha y hora exactas del suceso guardadas en un campo y, a parte, hemos categorizado el día de la semana y la hora exacta en dos campos en formato entero del 1 al 7 para los días de la semana y del 0 al 1440 que representa en minutos desde las 00:00 la hora en la que se tomó la ubicación.
- Por último tenemos la tabla **modelos_entrenados** que contiene todos los pares de nodos en formato bigint para los cuales tenemos un modelo entrenado en el campo `modelo` que guardamos en formato binario.

2.4. Interfaz

Siguiendo el apartado anterior, en este apartado mostraremos la interfaz del usuario y las acciones disponibles para el usuario.

A continuación se detallan todos los elementos que aparecen en las Figuras 2.3a y 2.3b:



(a) Interfaz sin ruta (b) Interfaz con ruta

Figura 2.3: Interfaces

1. Se trata del mapa local, podemos actuar sobre este mapa de tres formas diferentes: podemos mover el mapa con los dedos, podemos hacer zoom en el mapa con los dedos o podemos hacer click en el mapa con los dedos.
2. Se trata de un icono que representa la ubicación actual del usuario encima del mapa.
3. Estos botones pueden ser utilizados para realizar zoom en la pantalla.
4. Esta linea negra en la pantalla indicará la ruta recomendada por la aplicación para el usuario.
5. Este botón con forma de cruz se utiliza para deshacerse de la ruta mostrada y regresar a la interfaz anterior
6. Aquí mostramos unos datos que indican la distancia total de la ruta, el tiempo aproximado que se tarda y el porcentaje de carriles seguros que lo componen.
7. Este slider se utiliza en conjunto con el botón de reobtener ruta para indicar el porcentaje de seguridad o rapidez deseada en la ruta a generar
8. Este botón vuelve a generar una ruta para el mismo destino con las nuevas opciones de seguridad o tiempo deseadas.

CAPÍTULO 3

Desarrollo frontend

En este capítulo hablaremos sobre el desarrollo de las funciones clave de la Aplicación móvil, el objetivo principal en esta aplicación ha sido el de crear una aplicación simple y concisa, sin muchas funcionalidades, con la que poder mostrar de forma correcta las rutas generadas por nuestro servidor.

3.1. Gestión de permisos

El primer hito clave con el que nos cruzamos fue la gestión de los permisos. Esto es relevante ya que uno de nuestros objetivos principales es el de recolectar los datos geográficos de la sesión, para lo cual necesitamos que el usuario nos ceda los permisos correspondientes para acceder a la ubicación precisa del teléfono móvil.

Teóricamente algo sencillo: implementar una petición de permisos al abrir la aplicación y olvidarse del tema. Sin embargo, resultó que esto ya no podía hacerse así. A partir de la actualización de Android 6 Marshmallow los permisos tenían que pedirse en tiempo de ejecución y cerciorarse de que estaban concedidos a la hora de intentar utilizarlos [27].

Podemos ver la implementación final en la Figura 3.1, donde hemos creado una función auxiliar que llamamos en las instancias en las que necesitamos utilizar algún permiso, esta función se encarga de comprobar si disponemos del permiso y solicitarlo al usuario en caso de no disponer de él.

```
private void requestPermissionsIfNecessary(String[] permissions) {
    ArrayList<String> permissionsToRequest = new ArrayList<>();
    for (String permission : permissions) {
        if (ContextCompat.checkSelfPermission( context: this, permission)
                != PackageManager.PERMISSION_GRANTED) {
            // Permission is not granted
            permissionsToRequest.add(permission);
        }
    }
    if (permissionsToRequest.size() > 0) {
        ActivityCompat.requestPermissions(
                activity: this,
                permissionsToRequest.toArray(new String[0]),
                REQUEST_PERMISSIONS_REQUEST_CODE);
    }
}
```

Figura 3.1: Función para comprobar permisos

3.2. Obtención de ruta del servidor

El segundo hito en el desarrollo de la aplicación móvil fue el de, a través del servidor, obtener una ruta como lista de nodos. Para ello, primero teníamos que realizar un envío a la API REST del servidor con los datos de ubicación del usuario (latitud y longitud), con los de la ubicación deseada (latitud y longitud) y con el nivel de seguridad deseado, siendo el nivel más bajo que el usuario antepone reducir el tiempo del trayecto a la utilización de carriles seguros como pueden ser los carriles bici o los carriles mixtos y el nivel más alto lo contrario. Después, tendríamos que esperar a la respuesta del servidor y tratar los datos de la salida.

Para la primera parte se ha creado una función llamada **get** la cual obtiene los datos del usuario y la configuración del slider de seguridad y, después, crea una petición POST de HTTP con la url de la API, incluyéndole los datos de geolocalización del usuario, de su objetivo y su preferencia en cuanto a seguridad o tiempo, realiza una conexión HTTP con el servidor y lo envía.

Esta petición puede resultar en un error o en una respuesta correcta, si se trata de la segunda, tratamos la respuesta para obtener los datos en el formato que deseamos, obteniendo primero los datos de tiempo, distancia y seguridad de la ruta y luego una lista de nodos, formados por latitud y longitud.

Si la respuesta fuese errónea debido a que la ubicación elegida por el usuario no es accesible mediante patinete eléctrico (por ejemplo: debido a que hay que atravesar una autovia o el mar), se mostraría un mensaje indicando al usuario que esta ruta no está disponible y que lo intente con otro destino.

3.3. Visualización de ruta

Al final de la sección anterior, obteníamos una lista de nodos, formada por latitudes y longitudes que ahora necesitamos procesar para obtener de ellos una ruta visual que mostrar en la pantalla del usuario.

Nuestro primer acercamiento fue intentar usar el método propio de OSMDroid para generar las rutas entre cada par de nodos, pero el resultado no fue bueno y creaba rutas poco apropiadas. Para solventar este problema, se decidió crear la ruta manualmente.

Lo primero que hemos hecho ha sido crear una nueva lista, esta vez de GeoPoints que es la clase que utiliza nuestra biblioteca OSMDroid para tratar los puntos geográficos. A continuación, hemos inicializado una variable de clase polylinea que es la clase que tiene OSMDroid para trazar líneas sobre el mapa, le hemos añadido los GeoPoints y lo hemos añadido al overlay de nuestra aplicación, haciendo así que se muestre por pantalla la ruta.

3.4. Almacenamiento de datos del usuario

Al final el punto más importante para nosotros es el almacenamiento de los puntos geográficos por los que pasa el usuario, de esta manera podemos obtener datos de uso real que nos indican la duración de los trayectos en determinados días y horas.

Para ello, cuando la navegación comienza, llamamos a la función **enviarDatos** el cual obtiene los datos actuales de posición del usuario (comprobando los permisos primero y solicitándolos de nuevo si no se encuentran, ver Figura 3.1), la fecha y hora actuales y una id generada aleatoriamente asociada a la sesión del usuario y realiza una conexión HTTP con el servidor para realizar una petición de tipo POST enviando estos datos.

Este método se llama por primera vez nada más se genera la ruta del usuario y se configura a sí misma para volver a lanzarse transcurridos 30 segundos, de esta forma podemos obtener datos del usuario de forma constante desde que comienza la ruta hasta que la termina, momento en el cual deja de enviar datos.

CAPÍTULO 4

Desarrollo backend

En este capítulo detallaremos las principales funcionalidades que tiene nuestro servidor y el detalle de sus implementaciones. El objetivo principal del servidor es el de generar las rutas más precisas y acordes a las necesidades de los usuarios, para ello tiene que ser capaz de procesar los tiempos reales que extraemos de los usuarios y entrenar modelos de aprendizaje automático para la estimación de los tiempos en las rutas consiguientes. Así, el servidor irá re-entrenándose a medida que obtiene datos nuevos y será capaz de devolver estimaciones más precisas cuando tenga una amplia base de datos.

4.1. Conexión con la base de datos

El primer paso para poder trabajar con grandes cantidades de datos es tenerlas guardadas en una base de datos y acceder a ellas cuando sea necesario. Para ello al arrancar el servidor realizamos una conexión con nuestra base de datos local y mantenemos esta conexión abierta hasta que detenemos el servidor, así tan solo tenemos que ir realizando las consultas necesarias sin necesidad de conectar o desconectarnos.

Para ello utilizaremos la biblioteca **psycopg2** [28] la cual está diseñada para conectarse a bases de datos de PostgreSQL y nos permite una experiencia de uso sencilla y óptima.

Lo primero es crear la conexión, para ello utilizamos la función `connect` con el nombre de la base de datos, el usuario, la contraseña, la ip y el puerto a utilizar, en nuestro caso la base de datos está en la misma máquina que el servicio por lo que utilizaremos la ip local (véase la Figura 4.1).

El siguiente paso es generar un **Cursor** que es el encargado de ejecutar los comandos de PostgreSQL, mediante esta clase podemos ejecutar código SQL en la base de datos y obtener los resultados de las consultas, véase la Figura 4.2.

De esta forma podemos interactuar de forma innata con nuestra base de datos dentro del servidor y realizar consultas o modificar columnas del mismo en tiempo real.

```
conn = psycopg2.connect(database="postgres", user="postgres",
                        password="Mistermiad9", host="localhost", port="5432")
```

Figura 4.1: Conexión a la base de datos

```
cursor = conn.cursor()
cursor.execute("SELECT nodoa, nodob, modelo FROM public.modelos_entrenados;")
resultados = cursor.fetchall()
```

Figura 4.2: Creación del cursor

4.2. Procesado de datos

El procesamiento de datos es la parte más extensa e importante de nuestro programa, pues utilizando los datos de los usuarios que tenemos guardados en la base de datos, estudia todas las rutas recorridas por los usuarios y calcula el tiempo que tardaron en recorrer cada par de nodos de la ruta para así entrenar un modelo de aprendizaje automático para cada par.

Nuestra función de procesamiento de datos está programada para ejecutarse cada 60 minutos y comprobar si existen datos no procesados en la base de datos para procesarlos.

El primer paso es obtener los datos no procesados de la base de datos, esto se realiza como se explicó en el Apartado 4.1, utilizando un cursor y realizando una consulta SQL de tipo SELECT a nuestra base de datos, solicitando el primer grupo de datos no procesados que compartan la id única que tienen asociada.

Una vez obtenidos estos datos, que se tratan de una lista de pares de longitud y latitud, tiempo e id, se crea una lista con ellos y se utiliza el método **nearest_nodes** de OSMNX con cada par de coordenadas para obtener el nodo de OpenStreetMap más cercano. Una vez tenemos todos los nodos, debemos descubrir si existen nodos intermedios entre ellos ya que los datos del usuario son capturados cada 30 segundos y en ese intervalo de tiempo el usuario puede haber recorrido varios nodos. Para esto utilizamos el método **shortest_path** también de OSMNX que nos genera una lista de nodos intermedios que conectan nuestro primer nodo con el segundo. Una vez tenemos estos datos, creamos una lista con todas las coordenadas de nuestros nodos.

Debido al error del GPS, las coordenadas de sus ubicaciones no son precisas. Esto se debe a que la ubicación del dispositivo móvil tiende a ser aproximada por diversos motivos. Por un lado, puede que se pierda conectividad con algún satélite de GPS. Por otro lado, el sistema GPS introduce un error intencional para evitar el uso militar del GPS a quien no debe. Por otro lado, las ondas se reflejan en superficies y pueden confundir al receptor GPS. Para resolver este problema hemos realizado proyecciones de los puntos geográficos del usuario en el par de nodos más cercanos de la ruta que recorre. Así, obtenemos el punto exacto y su respectivo momento exacto. Estos datos los añadimos a nuestra lista de coordenadas, reemplazándolas por sus coordenadas originales.

El último paso para tener todas las estimaciones de tiempo es recorrer nuestra lista de coordenadas desde el principio y, para cada par de puntos que sí tienen una estimación de tiempo (las proyecciones del usuario, véase Figura 4.3), calcular la diferencia de tiempo entre estos dos puntos y asignarlos de acuerdo a la longitud de cada tramo a los diferentes nodos.

```
36.720937328284954 -4.48192142300788 2023-04-25 01:20:10.523000+02:00
36.7210104 -4.4817941 None
36.7214042 -4.4802351 None
36.7212575 -4.4799703 None
36.720937328284954 -4.48192142300788 2023-04-25 01:25:10.523000+02:00
```

Figura 4.3: Ejemplo de nodos intermedios

Finalmente, con todos estos datos, creamos un diccionario en el que guardamos cada par de nodos consecuentes que componen la lista y el tiempo en el que se recorre ese tramo. Adicionalmente, obtenemos el día de la semana que es y la hora exacta en minutos desde las 00:00 y todos estos datos los guardamos en la tabla de datos procesados de la base datos.

Una vez todos estos datos han sido procesados procedemos a entrenar un modelo de aprendizaje automático para cada par de nodos inexistentes en nuestra base de datos y a re-entrenar los modelos ya existentes con los nuevos datos obtenidos.

4.3. Creación y re-entrenamiento de modelos

Nuestro objetivo principal es el de obtener tiempos realistas a la hora de generar una ruta. Para ello, el uso de la distancia euclídea no era la mejor alternativa ya que esta no tiene en cuenta problemas como el tráfico que dependen de principalmente de una calle, día de la semana y hora específicas. Por ello proponemos el uso de técnicas de aprendizaje automático para obtener tiempos más realistas basados en datos históricos recopilados de los usuarios.

El primer paso fue escoger el método de implementar el modelo, para ello estudiamos 2 alternativas: un modelo general y uno por segmento.

El modelo general fue descartado debido a que implicaba utilizar las ids de los pares de nodos como variables de entrada, para ello tendríamos que transformarlas en variables categóricas y otorgarles un orden el cual desconocemos, además de que, al alimentar al modelo con datos tan diferentes, se dificultaba el aprendizaje.

El modelo por segmento implica que se debe utilizar un diccionario externo para relacionar cada par de nodos con su modelo correspondiente, esta solución es más eficiente y es la que hemos tomado.

Como se indica en el apartado anterior, se ha implementado una función capaz de crear o re-entrenar modelos de aprendizaje automático en nuestro servidor. Esta función comprueba si disponemos de un modelo ya creado en nuestro diccionario para cierto par de nodos. Este diccionario contiene un modelo para cada par de nodos que conocemos y es actualizado cada vez que creamos o entrenamos uno.

Los datos esenciales que necesitamos para entrenar un modelo son:

- Variables de entrada: Se trata del día de la semana y la hora. Esta primera variable es categórica, siendo el dia un número del 0 al 6. La hora es un número entre el 0 y el 1440 que indica la hora en minutos desde las 00:00.
- Variable de salida: Se trata del tiempo en segundos que se tarda en recorrer el segmento existente entre el par de nodos.

Para crear un modelo nuevo, llamamos al constructor del modelo, en nuestro caso utilizamos SGDRegressor [29] que es una opción adecuada para tareas de regresión a gran escala debido a su capacidad para manejar conjuntos grandes de datos y su rápido tiempo de entrenamiento. Además, este modelo es capaz de incorporar información adicional y no es necesario entrenarlo de cero, que es muy costoso, cuando obtengamos datos nuevos referentes a un par de nodos cuyo modelo ya existe.

El siguiente paso es simplemente utilizar el método `fit` del modelo para entrenarlo con nuestros datos de entrada y salida. Después, utilizamos la biblioteca Pickle [30], que sirve para serializar y deserializar objetos de Python, permitiendo así albergar los datos en memoria. Esto nos facilita que podamos guardar nuestro nuevo modelo en un diccionario en memoria, además de en la base de datos, para acceder a él de forma más rápida.

En el caso de existir un modelo para nuestro par de nodos en el diccionario, lo desearilzamos con Pickle y utilizamos la función `partial_fit` para re-entrenar nuestro modelo. Luego volvemos a serializarlo y lo actualizamos en nuestro diccionario y en la base de datos.

4.4. Cálculo de rutas

La generación de rutas es en teoría una acción simple, pues teniendo un punto A y un punto B utilizamos el algoritmo de Djikstra [31] para obtener la ruta más corta posible. Sin embargo, la complicación reside en cómo realizar la asignación de pesos (véase el Apartado 4.5) de forma que la ruta obtenida sea la más óptima en cuanto a tiempo y seguridad. Para obtener el factor del tiempo hemos de recurrir a nuestros modelos de aprendizaje automático para cada par de nodos y, si no disponemos de uno, realizar estimaciones manualmente con nuestros datos por defecto.

Como hemos definido anteriormente, el primer paso es, mediante unos puntos de coordenadas A y B, obtener los nodos de OpenStreetMap más cercanos, para esto, utilizaremos el método `distance.nearest_nodes` de la biblioteca OSMNX que nos lo encuentra.

El siguiente paso es utilizar el algoritmo de Dijkstra, para ello utilizaremos el método `dijkstra_path` de la biblioteca NetworkX [32] al cual le daremos un grafo de OSMNX que contiene todos los datos de los caminos de Málaga, un nodo origen, un nodo destino y una función con la cual asignar los pesos correspondientes (ver Apartado 4.5).

Una vez obtenida esta ruta, que vendrá como lista de ids de nodo de OpenStreetMap, se procede a calcular el tiempo total del viaje, la distancia total y el porcentaje de carriles seguros que lo componen.

Para el cálculo del tiempo total se realiza una sumatoria de los tiempos que se tarda en recorrer cada par de nodos (ver Apartado 4.6).

Para obtener la distancia total se realiza una sumatoria de la longitud de los carriles que componen la ruta.

Para el cálculo del porcentaje de carriles seguros se busca en los metadatos de los carriles si contienen alguna etiqueta que indique si son de tipo carril bici o mixto, estos se contemplan en la aparición de la etiqueta cycleway (ver Figura 4.4), y se realiza una media de la siguiente forma:

$$PorcentajeSeguro = \frac{CarrilesSeguros}{CarrilesTotales} * 100$$

Con estos tres datos más la ruta obtenida se genera un archivo JSON y se envía como respuesta a la aplicación móvil.

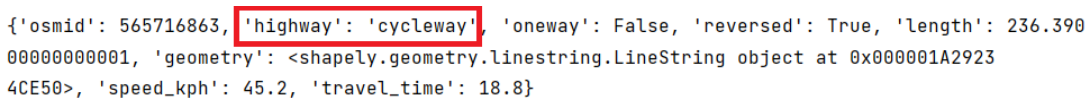


Figura 4.4: Etiqueta cycleway en los metadatos de un carril

4.5. Asignación de pesos para Dijkstra

Como indicamos en el apartado anterior, surgió la necesidad de diseñar una función para la asignación de pesos para el Algoritmo de Dijkstra.

Un variable importante que entra en juego en este apartado es nuestra lista de constantes de seguridad. Esta lista se compone de diez valores entre el cero y el uno que sirven para incrementar o disminuir los pesos generados en función de si el usuario quiere darle más importancia a los carriles seguros. De esta lista obtendremos una constante de seguridad inversamente proporcional al nivel de seguridad elegido por el usuario que utilizaremos en las fórmulas de cálculo de pesos.

En nuestra función el primer paso es comprobar si tenemos en el sistema un modelo entrenado para el par de nodos que se solicita, así sabremos si podemos utilizar el modelo para generar una estimación del tiempo o tendremos que calcularla manualmente.

Una vez sabemos lo anterior, comprobamos las etiquetas del carril que conecta ambos nodos para descubrir si se trata de un carril bici, mixto o un carril corriente para utilizar la fórmula correspondiente:

- En el caso del carril bici, se multiplica la constante de seguridad por la estimación del modelo sobre el par de nodos o, en su ausencia, se calcula manualmente utilizando la longitud del carril entre la velocidad media del patinete para este tipo de carril [33].

$$Peso = ConstanteSeguridad * EstimacionDeTiempo$$

- En el caso del carril mixto es similar la fórmula, sin embargo la velocidad media del patinete en este carril se estima que es menor por lo que el peso resultante será mayor que en el caso anterior.

$$Peso = ConstanteSeguridad * EstimacionDeTiempo * 1{,}5$$

- Para el resto de carriles no se tiene en cuenta la constante de seguridad por lo que el peso resultante siempre será mayor que en el resto excepto cuando el usuario indique que prefiere en su totalidad el tiempo ante la seguridad de la ruta, en ese caso la constante de seguridad valdrá uno igual que para este tipo de carril.

$$Peso = 1 * EstimacionDeTiempo * 1{,}5$$

Finalmente, utilizamos este peso para ese par de nodos en nuestro algoritmo de Dijkstra.

4.6. Estimación de tiempos de viaje

Como se indicaba en el Apartado 4.4, a la hora de crear rutas para los usuarios necesitamos calcular el tiempo total del viaje. Para ello, se realiza una sumatoria del tiempo que se tarda en recorrer cada par de nodos, el cual se obtiene o bien calculándolo de forma manual si no se dispone de un modelo o bien realizando una estimación utilizando el modelo existente.

Para cada par de nodos que no exista un modelo, realizaremos los cálculos de tiempo manualmente. Para ello, comprobaremos el tipo del carril que conecta estos dos nodos y, en consecuencia, utilizaremos una velocidad media estimada para patinetes eléctricos [33] y la longitud del carril para estimar el tiempo de tránsito.

Para el segundo caso tenemos una función que toma como datos de entrada un par de nodos y una fecha, comprueba que existan en el diccionario y obtenemos su modelo deserializado. En este método, a partir de la fecha se obtienen el día de la semana y la hora en los formatos comentados anteriormente y se utiliza el método `predict` del modelo, el cual nos devuelve una estimación del tiempo que se tarda en recorrerlo teniendo en cuenta los datos de entrada.

CAPÍTULO 5

Pruebas

Durante este capítulo comentaremos los diversos métodos mediante los cuales hemos probado el correcto funcionamiento de nuestro servidor y de su capacidad para realizar estimaciones correctas. Para realizar estas pruebas hemos utilizado dos métodos. El primer método ha sido mediante la aplicación de Postman [34], aplicación que sirve para hacer llamadas a la API del servidor y obtener sus resultados, la cual se ha utilizado a lo largo de todo el desarrollo para comprobar si las funciones de nuestro servidor obtenían el resultado esperado. El segundo método trata de crear una aplicación independiente que cree cientos de datos con sus respectivos tiempos y los introduzca en el modelo de Machine Learning para comprobar si las estimaciones se acercan a la realidad.

5.1. Pruebas con Postman

Como comentamos anteriormente, a lo largo del desarrollo hemos utilizado la aplicación Postman para realizar diversas pruebas que nos servían para comprobar el correcto funcionamiento de nuestro servidor sin tener que recurrir a la aplicación móvil o esperar a que el procesamiento de datos se realizase de forma automática.

Estas pruebas se logran mediante llamadas a la API REST con métodos GET o POST, dependiendo de la índole de la función, con los datos de entrada con los que queremos realizar la llamada.

La primera prueba que realizamos fue la de la creación de rutas. Para esto, mediante una llamada POST accedemos a la función `calcularuta` de nuestro servidor, indicándole las coordenadas de origen y coordenadas destino, además de la seguridad requerida por el usuario.

Con un pequeño cambio en el código del servidor podemos mostrar por pantalla la ruta generada en lugar de devolverla como respuesta. Utilizamos el método `plot_graph_route` de OSMNX para mostrar por pantalla la ruta generada (ver Figura 5.1).

Figura 5.1: Ruta generada

Ahora que tenemos una respuesta visual a nuestras consultas, podemos comprobar el correcto funcionamiento de la variable de seguridad. Para esto hemos realizado dos pruebas en las cuales, llamamos a esta función con las mismas coordenadas de origen y destino pero cambiamos la necesidad de seguridad por una de tiempo.

En la opción que tiene como preferencia la seguridad (ver Figura 5.2) podemos comprobar en la salida como se genera una ruta con una distancia mayor (el primer valor, representando 2803 metros) y mayor tiempo (segundo valor, representando 4 minutos), pero se utiliza la mayor cantidad de carriles seguros posibles (tercer valor, representando un 67 % de carriles seguros en el trayecto).

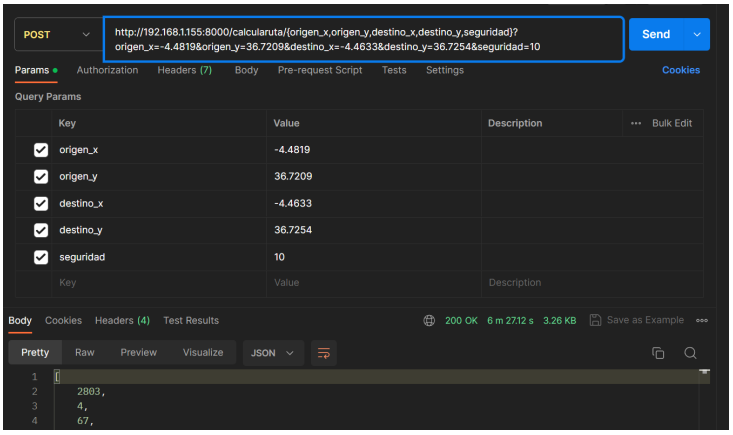


Figura 5.2: Prueba pata generar ruta una con seguridad 10

Como contra-parte tenemos la opción de generar la misma ruta pero, en lugar de enfocarse en la seguridad de los carriles, enfocarnos en el tiempo. De esta forma, utilizamos un 1 como valor en la variable de seguridad que le pasamos al sistema, obteniendo resultados acordes a esta solicitud. Comprobamos que obtenemos una distancia total de 2336 metros, con un tiempo medio de 4 minutos y un porcentaje de 31 % de carriles seguros (véase Figura 5.3. Aunque el tiempo medio en esta ocasión no cambie, podemos comprobar que la ruta es claramente más corta, más directa y no tiene tanto en cuenta los carriles seguros.

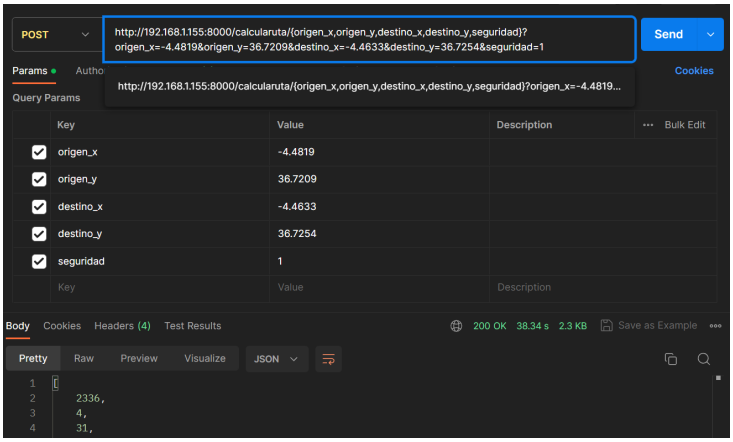


Figura 5.3: Prueba para generar una ruta con seguridad 1

Finalmente, podemos observar la diferencia entre estas rutas en las Figuras 5.4a y 5.4b.

(a) Ruta generada con seguridad 10 (b) Ruta generada ruta con seguridad 1

Figura 5.4: Visualizaciones de rutas

Además de poder generar rutas y mostrarlas por pantalla, mediante Postman tenemos la opción de guardar datos no procesados en la base datos como si de los datos del usuario se tratase (ver Figura 5.5), llamar a la función para procesar el primer bloque de datos no procesados de la base de datos (ver Figura 5.6) y, por último, realizar estimaciones de tiempo utilizando nuestro modelo de Machine Learning (ver Figura 5.7). Estos endpoints han sido añadidos exclusivamente para las pruebas y han sido removidos del código final.

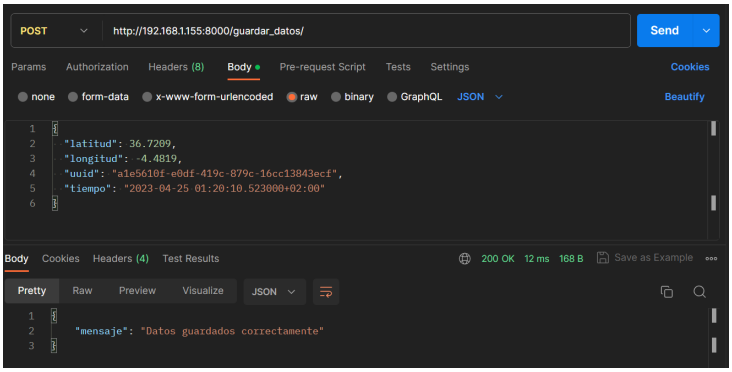


Figura 5.5: Prueba pata guardar datos

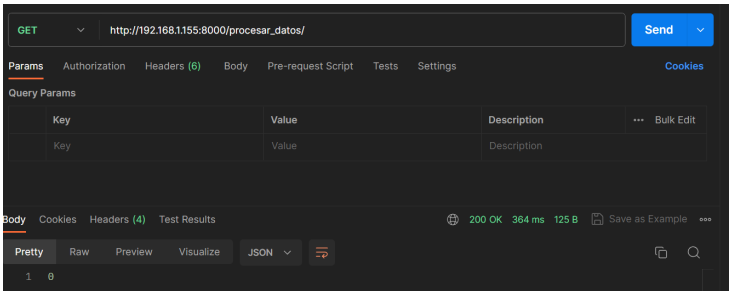


Figura 5.6: Prueba pata procesar datos

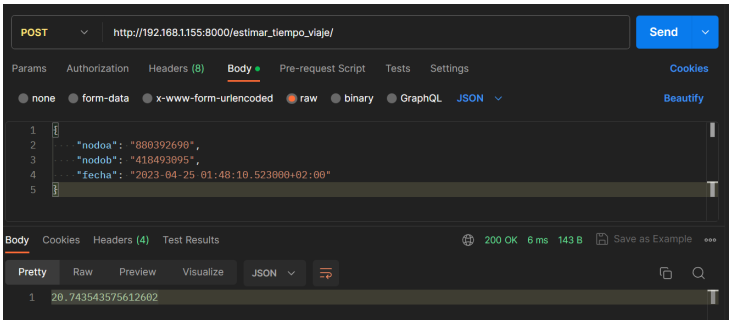


Figura 5.7: Prueba pata realizar estimaciones

5.2. Aplicación de testeo

Como forma de comprobar como responde este modelo de Machine Learning a grandes cantidades de datos, hemos diseñado un programa independiente que genera una gran cantidad de datos de ejemplo con sus tiempos correspondientes y para ellos crea y entrena sus modelos correspondientes.

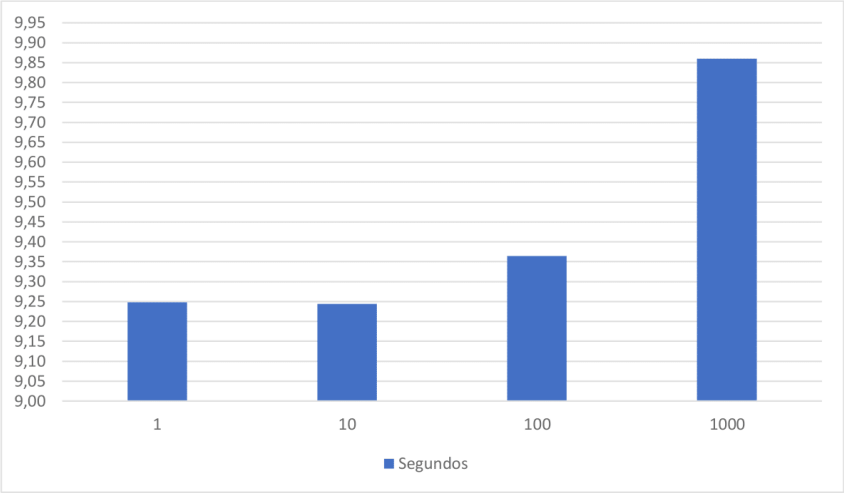
A continuación, pedimos estimaciones de tiempo para los pares de nodo correspondientes y comprobamos estos datos con nuestro datos de origen para comprobar el porcentaje de error que obtenemos en el modelo.

Teniendo como base unos nodos A y B cuyo tiempo de tránsito es de una media de 10 segundos, hemos creado datos aleatorios en lo referente a dia de la semana y hora y para la duración del trayecto hemos generado valores aleatorios entre 5 y 15 segundos. En nuestra simulación hemos realizado estimaciones con 1, 10, 100 y 1000 datos en nuestro modelo y podemos observar que las primeras pruebas devuelven una estimación alejada de la media, pues estas tienen pocos datos que analizar (ver Figuras 5.8a y 5.8b). Sin embargo, a medida entrenamos el modelo, este es capaz de realizar mejores estimaciones hasta el punto de acercarse a la media.

De esta forma hemos podido comprobar que el porcentaje de error es inversamente proporcional a la cantidad de datos con los que hemos entrenado nuestro modelo, llegando a hacer predicciones más precisas cuando la cantidad de datos es lo suficientemente grande.

```
Tiempo estimado con un entrenamiento 9.248122380096127
Tiempo estimado con 10 entrenamientos 9.24432637464085
Tiempo estimado con 100 entrenamientos 9.364226557853119
Tiempo estimado con 1000 entrenamientos 9.85940244926863
```

(a) Resultados de las pruebas



(b) Diagrama de barras referente a las pruebas

Figura 5.8: Pruebas en aplicación independiente

CAPÍTULO 6

Conclusiones y trabajos futuros

En este último capítulo se comenta la forma en la que se han abordado los objetivos del proyecto, si los resultados han sido favorables con respecto a las pruebas y las expectativas y la forma en la que se puede extender este proyecto en el futuro.

Como vimos en el apartado de los objetivos 1.2, el proyecto consistía en una serie de hitos clave que se buscaban cumplir para que este contase como finalizado. Comentaremos estos hitos y cómo los hemos abordado.

- Crear la aplicación móvil: Tenemos una aplicación funcional para sistemas Android capaz de comunicarse con el servidor y mostrar por pantalla las rutas que los usuarios solicitan de acuerdo con sus preferencias.
- Crear el servidor: Tenemos un servidor en Python que implementa una API REST que acepta llamadas externas para guardar datos en el servidor, procesar datos, calcular rutas teniendo en cuenta los modelos existentes y realizar estimaciones de tiempo para pares de nodos.
- Crear una base de datos: Tenemos una base de datos en PostgreSQL con tres tablas que almacenan los datos del usuario no procesados, los procesados y los modelos que tenemos entrenados.
- Entrenar el modelo de Machine Learning: Nuestro servidor es capaz de procesar los datos no procesados de la base de datos cada cierto tiempo de forma autónoma y crear o re-entrenar modelos de Machine Learning con esos datos.
- Crear una consulta capaz de utilizar el modelo para crear una ruta: Dentro del servidor tenemos una función que comprueba nuestro diccionario de modelos a la hora de asignar pesos en un algoritmo de Dijkstra para calcular la mejor ruta posible.

A nivel personal este TFG ha servido para aprender muchas tecnologías nuevas como Android y Python y técnicas como el aprendizaje automática de las cuales no se tenía conocimiento.

A continuación estudiaremos los resultados obtenidos en las pruebas que hemos realizado y las formas en las que podríamos expandir el trabajo en el futuro.

6.1. Resultados

Durante el capítulo 5 hemos visto los diferentes métodos que hemos utilizado para realizar pruebas en nuestro servidor. Sin embargo, no hemos comentado los resultados obtenidos y cómo se reflejan estos para nuestra finalidad.

A la hora de generar rutas el punto más importante para nosotros es que se utilicen las estimaciones más cercanas a la realidad para utilizar los carriles más óptimos a la hora de elegir los nodos que recorrer. Hemos comprobado como al cambiar las necesidades de seguridad o tiempo del usuario la ruta generada cambia enormemente y en el proceso siempre que para un par de nodos existe un modelo de Machine Learning, este es utilizado para estimar su tiempo.

Podemos concluir se han cumplido con los requisitos y expectativas del proyecto debido a que todos los hitos se han cumplido y se han superado las expectativas en cuanto a la estimación de los tiempos.

6.2. Futuras líneas de trabajo

Este proyecto comenzó teniendo como objetivo centrarse en el aprendizaje automático de las rutas de los patinetes eléctricos y se desvió el objetivo principal en hacer una aplicación móvil y hacerlo accesible para el público, sin centrarse tanto en los diferentes modelos que existen para aplicar el aprendizaje automática a nuestro programa.

Ese enfoque también ha producido que no se haya podido trabajar mucho en la aplicación móvil y únicamente contiene las funciones esenciales para el funcionamiento de esta.

Si se continuase con el proyecto en el futuro primero se estudiaría todos los diferentes módulos que existen en Scikit-Learn para la aplicación de Machine Learning para encontrar el más óptimo para nuestra aplicación y, al tener claro qué modelo utilizar, se trabajaría en la aplicación móvil para añadir funciones útiles para el usuario y en tener una interfaz mucho más dinámica y trabajada.

Bibliografía

1. M. López, Radiografía de la movilidad en bici y patinete. *revista.dgt.es*, (https://revista.dgt.es/es/reportajes/2022/03MARZO/0331-Observatorio-radiografia-bici-y-patinete.shtml.) (2022).
2. E. Vivó, Los patinetes eléctricos siguen creciendo en España pese a las críticas. (https://neomotor.epe.es/movilidad/los-patinetes-electricos-siguen-creciendo-en-espana-pese-a-las-criticas-YY1098122#:~:text=En%20Espa%C3%B1a%20ya%20hay%20m%C3%A1s,por%20toda%20la%20geograf%C3%ADa%20nacional.) (2022).
3. Google Maps. (https://www.google.com/maps/about/#!/).
4. R. Panko, The Popularity Of Google Maps: Trends In Navigation Apps In 2018. *the manifest*, (https://themanifest.com/app-development/trends-navigation-apps) (2018).
5. Waze. (https://www.waze.com/es/about/).
6. Apple Maps. (https://www.apple.com/maps/).
7. MapQuest. (http://hello.mapquest.com/).
8. R. Dube, 7 Google Maps Alternatives and Why They're Better. (https://www.groovypost.com/howto/seven-google-maps-alternatives-and-why-theyre-better/) (2021).
9. ScootRoute. (https://scootroute.com/).
10. Android. (https://www.android.com/intl/es_es/).
11. Java. (https://www.java.com/en/download/help/whatis_java.html).
12. Kotlin. (https://kotlinlang.org/).
13. osmdroid. (https://github.com/osmdroid/osmdroid).
14. OpenStreetMap. (https://www.openstreetmap.org/about).
15. Android Studio. (https://developer.android.com/studio).

16. N. Mathur, What is Android Studio and how does it differ from other IDEs? (https://hub.packtpub.com/android-studio-how-does-it-differ-from-other-ides/#:~:text=UI%20development%3A%20The%20most%20significant,editors%2C%20and%20a%20drag%2Dand).

17. Python. (https://www.python.org/about/).

18. Fast APi. (https://fastapi.tiangolo.com/).

19. F. Pedregosa *et al.*, Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **12**, 2825-2830 (2011).

20. osmnx. (https://osmnx.readthedocs.io/en/stable/).

21. Pycharm. (https://www.jetbrains.com/es-es/pycharm/).

22. PostgreSQL. (https://www.postgresql.org/).

23. pgAdmin. (https://www.pgadmin.org/).

24. LaTex. (https://es.wikipedia.org/wiki/LaTeX).

25. Overleaf. (https://www.overleaf.com/about).

26. Scrum. (https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)).

27. Android 6.0 Changes. (https://developer.android.com/about/versions/marshmallow/android-6.0-changes?hl=es-419).

28. Psycopg2. (https://www.psycopg.org/docs/).

29. SGDRegressor. (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html).

30. Pickle. (https://docs.python.org/3/library/pickle.html).

31. Dijkstra. (https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra).

32. NetworkX. (https://networkx.org/).

33. D. D. Pedroza-Perez, J. Toutouh y G. Luque, presentado en Optimization and Learning - 6th International Conference, OLA 2023, Malaga, Spain, May 3-5, 2023, Proceedings, ed. por B. Dorronsoro, F. Chicano, G. Danoy y E. Talbi, vol. 1824, págs. 380-392, (https://doi.org/10.1007/978-3-031-34020-8_29).

34. Postman. (https://www.postman.com/).

Apéndices

APÉNDICE A

Manual de usuario

En este manual se detallan las funciones principales de la aplicación móvil y las instrucciones para su ejecución.

A.1. Selección de destino

Al ejecutar la aplicación la primera pantalla que observamos es un mapa centrado en nosotros, señalados por una flecha blanca, en el cual debemos elegir el destino hacia el cual queremos que se nos genere una ruta. En esta pantalla tenemos la opción de mover con el dedo el mapa para desplazarnos o de pellizcar la pantalla o pulsar los botones inferiores para alejar o acercar la vista sobre el mapa.

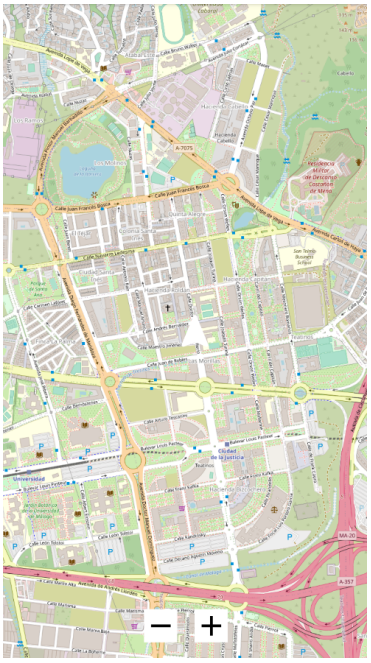


Figura A.1: Pantalla principal de la aplicación móvil

A.1.1. Elementos de la primera pantalla

Esta primera pantalla se compone de cuatro elementos.

El primer elemento es una flecha blanca ubicada en el mapa que indica la ubicación actual del usuario y la dirección en la que está mirando.

Figura A.2: Flecha que indica la ubicación del usuario

El segundo y el tercer elemento son unos botones ubicados en la parte inferior de la pantalla responsables de alejar y acercar la vista del mapa. El botón número 1 sirve para alejar la vista y el botón número 2 sirve para acercar la vista.

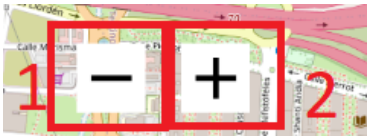


Figura A.3: Botones de Zoom

El último elemento, y más importante, es el mapa. Con este elemento se puede interactuar de varias formas para obtener diferentes resultados. Comenzando por deslizar el mapa con un dedo para moverse a través de él, pinchar con los dedos para acercar o alejar la vista igual que ocurre con los botones o pulsar en algún lugar del mapa, momento en el cual se nos genera una ruta desde nuestra ubicación actual hasta el lugar que hayamos seleccionado.

Figura A.4: Mapa

A.2. Preferencias de ruta

Una vez hemos seleccionado un destino al cual queremos acudir, cambiamos a una segunda pantalla con una ruta para llegar a nuestro destino, información sobre la ruta y opciones para generar una ruta diferente o abandonar esta pantalla.

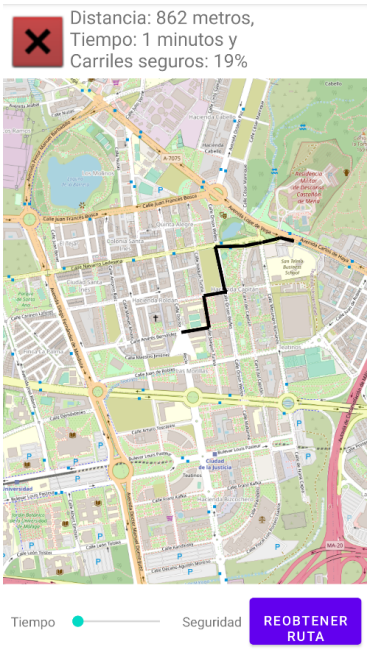


Figura A.5: Segunda pantalla de la aplicación

A.2.1. Elementos de la segunda pantalla

Esta segunda pantalla se compone de siete elementos.

El elemento más importante es el mapa que se abarca el centro de la pantalla y se utiliza tanto para poder ver la ruta y nuestra posición, como para realizar todas las funciones que podíamos realizar en la pantalla anterior incluyendo la de seleccionar una nueva ubicación en el mapa para generar una ruta nueva hacia esa dirección y desechar nuestra ruta actual.

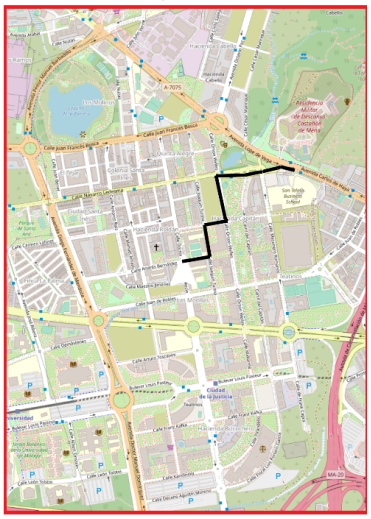


Figura A.6: Mapa con ruta

Igual que en la pantalla anterior, en nuestro mapa disponemos de una flecha blanca que indica la ubicación actual del usuario y la dirección en la que está orientado. Esta flecha se utiliza para poder seguir correctamente el recorrido de la ruta.

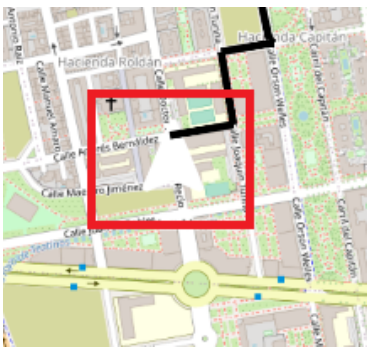


Figura A.7: Flecha que indica la ubicación del usuario en la ruta

Dentro del mapa podemos ver la ruta a seguir para llegar a nuestro destino que está detallada por lineas de color negro que conectan nuestra flecha con la el destino seleccionado.

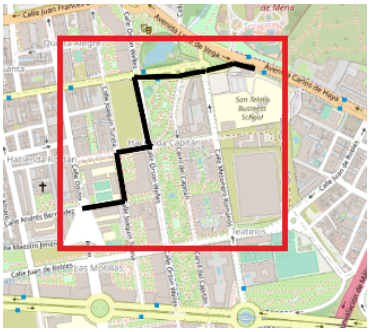


Figura A.8: Ruta para llegar al destino

En el centro de la parte superior de la pantalla, encima del mapa, tenemos los detalles de la ruta generada. Estos detalles indican la distancia hasta llegar al destino en metros, la duración del viaje en minutos y el porcentaje de carriles seguros que existen en el camino.

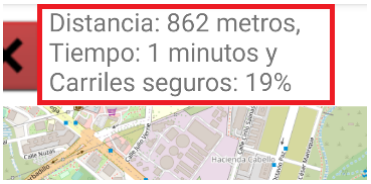


Figura A.9: Detalles de la ruta

En la parte inferior de la pantalla tenemos la parte encargada de la generación de rutas alternativas con diferentes preferencias en cuanto a seguridad. Esto se compone de un slider (Elemento 1) con el cual determinar si preferimos enfocarnos en la seguridad de la ruta o en reducir el tiempo del viaje y un botón (Elemento 2) que se encarga de generar la nueva ruta según estas características.

Figura A.10: Generación de rutas alternativas

En la esquina superior izquierda tenemos un botón rojo con forma de cruz encargada de desechar la ruta actual y volver a la pantalla anterior.

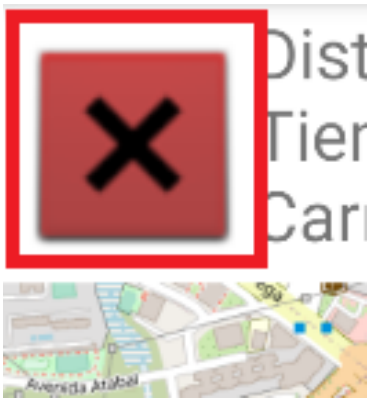


Figura A.11: Botón de retroceso

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA