UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

VISTRAFFIC: HERRAMIENTA PARA ANÁLISIS DEL TRÁFICO VEHICULAR EN SANTIAGO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS NICOLÁS RAMÍREZ MARTÍNEZ

PROFESORA GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
JOSÉ URZÚA REINOSO

SANTIAGO DE CHILE
2022

Resumen

El aumento del flujo vehicular es un fenómeno que se viene observando hace unos años en Chile, por lo que se debe analizar y estudiar, con tal de realizar mejoras en los flujos viales para prevenir congestiones y problemas de transporte en las ciudades.

Existen aplicaciones que permiten a los usuarios conocer la mejor ruta para viajar desde un punto A al punto B tomando los datos del tráfico para asignarle una ruta. Sin embargo, no permiten al usuario analizar el comportamiento de la ciudad con los datos históricos, ya que las aplicaciones usan estos datos en sus funcionalidades pero no los dejan a disposición de los usuarios. Sin analizar los datos históricos del tráfico vehicular de la ciudad no es tan directo realizar buenas reformas viales.

Para ellos se construyó VisTraffic, una aplicación web en la cual se pueden consultar los datos históricos del tráfico en un rango de tiempo que entrega el usuario de la aplicación. La consulta analiza todos los registros incluidos en el rango de tiempo entregado por el usuario y jerarquiza los tacos encontrados. La jerarquía se realiza en función de la frecuencia con que un taco aparece en el conjunto de registros analizados. En otras palabras, los tacos se ordenan según el tiempo que estuvieron presentes en la ciudad, no por su extensión o por su tiempo de retraso.

Para desarrollar esta aplicación se diseñó una arquitectura web que contiene un Frontend, un Backend y una base de datos no relacional. El Backend se encarga de consultar y manipular los datos, además sigue una arquitectura de software inspirada en Clean Architecture, con la intención de hacerla lo más extensible que se pueda. El Frontend tiene como objetivo realizar la consulta al Backend y marcar los tacos y su información contenidos en la respuesta.

El objetivo de esta aplicación es que sirva a usuarios dedicados al mundo del transporte, como personal del MTT, UCI, UOCT o Direcciones Viales, con el fin de conocer en profundidad cómo se comporta la ciudad. Por otro lado también puede ser útil para usuarios no relacionados a este tema, permitiéndoles conocer cómo se comportan las calles cercanas a donde viven o trabajan.

En este documento se describe todo el proceso de desarrollo de la aplicación, abarcando la exploración de otras plataformas, el análisis para entregar una aplicación diferente a lo que ya existe, la implementación de ella y su validación con usuarios.

Fire all of your guns at once
And explode into space.
Like a true nature's child
We were born
Born to be wild

Agradecimientos

Quiero agradecer a mi familia, a mis abuelos y a mi hermano, por darme siempre su apoyo incondicional, en especial mi mamá por darme la oportunidad de seguir una carrera profesional.

A mi polola Daniela le agradezco siempre estar a mi lado, junto con brindarme su apoyo y ayuda ante cualquier situación.

A mis amigos de la universidad, por darme un montón de buenos recuerdos durante los años que estuvimos en la facultad.

A mi amigo de la infancia Chevi, por siempre estar presente y ayudarme a entender lo inentendible.

A mi profesora guía Jocelyn, por su disposición a trabajar conmigo en el momento que más lo necesitaba.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Contexto

Debido al constante crecimiento del parque automotriz en la ciudad de Santiago, cada vez se registran más y mayores estancamientos vehiculares. Entre octubre de 2020 y septiembre de 2021 se comercializaron 390.964 unidades de vehículos cero kilómetros, lo que representa un incremento de 57,3 % comparado con el período móvil anterior (octubre de 2019 a septiembre de 2020) [6]. La Región Metropolitana concentra el 54,1 % de las compras del parque de vehículos motorizados livianos y medianos en lo que va del año, por lo que se puede esperar que los mayores cambios se concentren en esta región [6].

La Unidad Operativa de Control de Tránsito (UOCT) se dedica a optimizar la gestión de tránsito en las principales ciudades del país, a través de la administración y la operación de sistemas inteligentes de transporte y el mejoramiento de los sistemas de información a usuarios, para facilitar las condiciones de desplazamiento de las personas. La UOCT administra y opera los sistemas de control de modo de optimizar permanentemente las condiciones de tránsito en las principales ciudades, así como de otros sistemas inteligentes de apoyo a la gestión de tránsito, como son los circuitos cerrados de televisión, letreros de mensajería variable, estaciones de conteo automático de flujos vehiculares, entre otras herramientas. Actualmente la UOCT cuenta con el control del 95 % de los semáforos de las principales intersecciones de Santiago.

Los operarios de la UOCT tienen a su disposición cámaras en las principales intersecciones de la ciudad, las cuales son monitorizadas para ajustar los tiempos de los semáforos y reducir los tiempos de *taco* en Santiago. Según cifras de TomTom [13], el tiempo extra gastado por desplazarse en hora punta a lo largo de un año es de 4 días y 16 horas. Dado el constante crecimiento del parque automotriz, los operarios van a tener que solucionar un mayor número de congestiones, por lo que se necesita un software que optimice el tiempo en que los operarios conozcan cual es el estado del tránsito en Santiago.

Sumado al problema anterior, la UOCT actualmente no almacena datos del tráfico de la ciudad, por lo que muchas decisiones se van tomando sobre el momento sin la ayuda del análisis de datos históricos del tráfico.

1.2. Problema

El momento en que se genera más tráfico vehicular en Santiago corresponde a la hora punta, cuando la gente comienza a volver a sus hogares. Ante esto, los operadores de la UOCT van ajustando en tiempo real los tiempos de los semáforos para aliviar algunas congestiones.

Las cámaras en la ciudad ascienden aproximadamente a 250, las cuales son monitorizadas por un grupo de entre 6 y 8 operarios del UOCT, con el fin de identificar accidentes y modificar tiempos de semáforos para disminuir las congestiones vehiculares [16]. El problema de esta situación es que los operarios no pueden estar al tanto de todas las cámaras al mismo tiempo, por lo que se ayudan de plataformas como *Google Maps* o *Waze* para identificar el estado del tránsito. Lamentablemente, el uso de estas aplicaciones es realizado a nivel de usuario básico, por lo que el operador debe buscar en la plataforma los estancamientos a solucionar, donde lo ideal sería que la plataforma le entregue directamente esa información, sirviendo como una herramienta para optimizar su trabajo.

En caso de no buscar una mejor solución, se puede dar la situación que los operadores no logren ajustar los semáforos exitosamente, ya que se espera que cada vez aumentan los estancamientos en santiago [16], por lo que el tiempo que le toma al operador conocer el estado del tráfico de la ciudad va a ir en aumento.

Sin embargo, los ajustes que los operadores puedan realizar sólo representan una ayuda para momentos de hora punta y su ejecución no significa una mejora al desplazamiento automovilístico de la ciudad. Estos ajustes podrían tener un mayor impacto si de forma planificada, junto a un análisis del tráfico de Santiago, se fueran optimizando los tiempos de los semáforos de las avenidas o intersecciones que presentan problemas.

Toda esta información se obtuvo a partir de reuniones con Richard Mora, Gestor de Proyectos de la Unidad de Ciudades Inteligentes (UCI) del Ministerio de Transportes y Telecomunicaciones. La UOCT y UCI son dos unidades dentro del ministerio que trabajan de forma conjunta para mejorar la movilidad del país.

A partir de lo expuesto anteriormente podemos apreciar que existe un problema, el cual va más allá del trabajo que hacen los operadores de la UOCT, que es la falta de análisis y estudio de datos del tráfico de Santiago. Actualmente existen formas de poder acceder a estos datos, por lo que es importante tomarlos, procesarlos y poner información importante respecto de cómo se comporta la ciudad a disposición de personas encargadas del tema, como podrían ser miembros del Ministerio de Transporte, direcciones del tránsito de municipalidades, personas que trabajan en planificación urbana como también ciudadanos interesados en conocer las características del tránsito de su ciudad.

La plataforma propuesta busca solucionar este problema recolectando datos con intervalos de minutos del tráfico en Santiago, extrayendo información relevante de cómo se comporta la ciudad y hacerla disponible, todo esto con el objetivo de que sea de ayuda para la toma de decisiones viales en Santiago.

1.3. Objetivos

Objetivo General

El objetivo principal es concebir una plataforma que facilite el análisis del tráfico vehicular histórico de Santiago, con el fin de ser información relevante en la toma de decisiones.

Objetivos Específicos

1. Mostrar visualizaciones que entreguen información del tráfico histórico de Santiago.
2. Almacenar constantemente datos del tráfico de Santiago.
3. Tener una interfaz intuitiva para interactuar con gráficos, mapas y/o visualizaciones.
4. Diseñar y realizar validación de la aplicación con usuarios

1.4. Solución

Con el fin de generar una plataforma que levante información importante para la toma de decisiones a partir de los datos del tráfico de Santiago, se propone una aplicación web que tenga la capacidad de visualizar geográficamente los peores tacos de la ciudad de Santiago, pudiendo realizar consultas en rangos de tiempo sobre los datos históricos del tráfico. Para ello se utilizarán los datos de TomTom junto con el SDK que provee. Esta aplicación tendrá un Backend encargado de consultar y procesar los datos y un Frontend encargado de mostrar los datos procesados.

En cuanto a la base de datos se propone que debe ser de tipo no relacional dada la naturaleza de los datos. Los datos recopilados son series de tiempo, por lo que no tenemos relaciones entre entidades y la cantidad de entradas crece constantemente, por lo que se almacenarán en una base de datos MongoDB. Los datos del tráfico vienen desde TomTom, consultando su Traffic API. A través de su API se puede acceder a los estancamientos que se desarrollan en la ciudad en tiempo real, de los cuales se puede almacenar los siguientes datos: Geometría (ubicación geográfica de donde ocurre el `taco`), tiempo de retraso, hora de inicio, comienzo, dirección, magnitud del estancamiento, entre otros.

1.5. Metodología

La metodología usada para llevar a cabo este proceso fue seguir un orden específico de implementación de los sistemas que componen la solución. Lo primero a realizar fue el análisis de los proveedores de datos y seleccionar uno de ellos. Luego, como es natural, el primer proceso fue crear una base de datos y comenzar a poblar de datos. Con esto, listo el siguiente

sistema a construir es el Backend, el cual se encarga de consultar los datos. Finalmente se construyó un Frontend que toma el input del usuario, el cual representa una consulta, lo envía al Backend y marca en el mapa la respuesta de este.

1.6. Estructura del Documento

En los siguientes capítulos se describe todo el proceso del trabajo de memoria. En el Capítulo 2 Estado del Arte se describe el marco teórico, que habla de las principales tecnologías de desarrollo web y otras librerías utilizadas para construir la solución del problema, se hablará de los actuales servicios de datos de tráfico y mapas y de las aplicaciones existentes que hacen uso de estos datos.

En el Capítulo 3 Análisis y Diseño se analizan los servicios de TomTom, el cual será el proveedor de datos, a partir de este análisis se realiza un diseño de la solución y se describen sus principales componentes.

El Capítulo 4 Implementación describe en detalle el proceso de programación de la solución, destacando partes importantes del código, cómo se implementa la arquitectura propuesta para el Backend, cómo se construyó el Frontend y el proceso de recolección y consulta de datos.

Finalmente, en el Capítulo 5 Validación se muestra el diseño de un conjunto de actividades acompañadas de una encuesta que busca medir la funcionalidad de la solución junto con la usabilidad que posee. En el Capítulo 6 Conclusión se analiza la aplicación creada y se describen ciertos casos de borde que puede tener la solución. Este último capítulo es acompañado de trabajos futuros y posibles mejoras al software.

Capítulo 2

Estado del Arte

En este capítulo se entregará información de los algoritmos, librerías y frameworks a utilizar en la solución propuesta en la memoria. Luego se realiza un pequeño análisis de algunas aplicaciones webs relacionadas con el tráfico vehicular y se describe un poco de los datos explorados en internet relacionados al tráfico vehicular de Santiago.

2.1. Marco Teórico

Para el desarrollo de la memoria se utilizaron diversos recursos, como una librería para comunicarse con la base de datos, un framework que permite desarrollar aplicaciones web y un algoritmo de clasificación. A continuación se describen los principales recursos utilizados en la memoria.

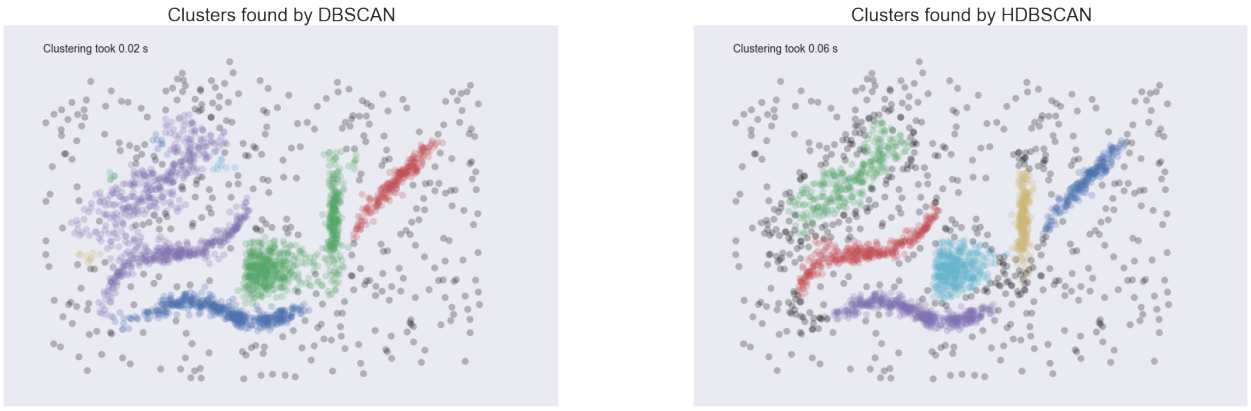
2.1.1. Pandas - Dataframe

La Librería Pandas [14] [15] es un software libre y una extensión de la librería numpy, ofrece manipulación y análisis de datos para el lenguaje de programación Python. Cuenta con estructuras de datos y operaciones para manipular tablas numéricas y series temporales.

Una de estas estructuras son los Dataframes, la cual cuenta con indexación integrada. Esta estructura cuenta con dos dimensiones en las cuales se puede guardar distintos tipos de datos como caracteres, enteros, decimales, entre otros. Es similar a una hoja de cálculo o una tabla de una base de datos relacional.

El uso de Dataframes hace más simple el manipular grandes cantidades de datos, además presenta varios métodos que permiten modificar esta estructura de forma análoga a como se manipulan tablas de bases de datos relacionales.

Para la memoria los Dataframes serán utilizados para manipular los puntos sobre los que se ejecutará el algoritmo de agrupamiento, ya que el método que ejecuta el algoritmo recibe



(a) Grupos encontrados por DBSCAN (b) Grupos encontrados por HDBSCAN

Figura 2.1: Comparación entre DBSCAN y HDBSCAN [1]

un Dataframe que almacena los datos.

2.1.2. HDBSCAN

HDBSCAN [7] es un algoritmo de agrupamiento basado en densidad, el cual es capaz de detectar en qué áreas existen concentraciones de puntos y donde están separados por áreas vacías o con escasos puntos. Los puntos que no pertenecen a un grupo se etiquetan en un grupo que representa el ruido en la agrupación.

Es un algoritmo del tipo no supervisado, ya que no requiere información de las formas de los grupos, sino que detecta formas arbitrarias basándose en la ubicación espacial y en la distancia a un número de vecinos especificado.

El algoritmo de agrupación DBSCAN utiliza una distancia específica para separar los grupos densos del ruido más disperso. A este algoritmo se le debe entregar una `distancia de búsqueda`, la cual es ocupada como métrica para identificar los grupos. Cuando los grupos presentan densidades similares este es el algoritmo más rápido para identificarlos. De aquí nace HDBSCAN, el cual no requiere de una distancia, lo cual puede ser visto como un `autoajuste`, probando toda una variedad de distancias para separar grupos de densidades variables del ruido más disperso.

El algoritmo HDBSCAN tiene una complejidad temporal $O(n^2)$, donde n es la cantidad de puntos a agrupar.

En la Figura 2.1 podemos ver que HDBSCAN encuentra los grupos (marcados por distintos colores) más densos de manera más directa. Sobre un conjunto de puntos generados al azar en el resultado obtenido por DBSCAN se aprecia que el grupo morado es grande y poco denso, esto se debe a que fue formado gracias al valor de la `distancia de búsqueda`. Por otro lado, el resultado de HDBSCAN separa el grupo morado encontrado por DBSCAN en dos grupos distintos, esto se debe a que HDBSCAN iteró por diversos valores de la variable `distancia de búsqueda`, obteniendo los grupos más densos sobre los puntos generados.

La respuesta del algoritmo es asignar a cada punto entregado el grupo al que pertenece, los cuales son enumerados de 1 a k (donde k es el total de grupos encontrados), mientras que los puntos que no pertenecen a un grupo son marcados con un -1.

2.1.3. Flask

Flask [3] es un framework minimalista escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código. Fue diseñado para trabajar bajo un patrón MVC y contiene funcionalidades como autenticación de usuarios, manejo de migraciones, entre otras. Estas funcionalidades deben ser agregadas, se conocen como extensiones de Flask, las cuales deben ser agregadas por los desarrolladores en la medida que sean necesarias, es por eso que se dice que es un framework minimalista.

Éste fue el framework elegido para desarrollar el backend de la solución propuesta, dado que con el framework base y la extensión flask-injector se puede llevar a cabo el diseño propuesto de un backend con inversión de dependencias.

2.1.4. Pymongo

Pymongo [4] es una librería de Python que permite conectarse a una base de datos no relacional MongoDB y realizar consultas sobre sus datos. Es la forma recomendada de trabajar con una base de datos MongoDB si se está escribiendo la aplicación con el lenguaje Python. Contiene todas las operaciones que se pueden realizar sobre una base de datos MongoDB.

Ya que la solución propuesta en esta memoria ocupa una base de datos no relacional MongoDB, el código fuente encargado de comunicarse con la base de datos debe incluir la librería pymongo para realizar esta comunicación correctamente.

2.1.5. Cron

Cron [2] es un administrador de tareas de Linux que permite ejecutar comandos en un momento determinado, por ejemplo, cada minuto, día, semana o mes. Si queremos trabajar con cron, podemos hacerlo a través del comando crontab. Un crontab es a la vez un archivo de texto donde se listan todas las tareas que deben ejecutarse y el momento en que deben hacerlo.

2.2. Datos y Mapas

Existen plataformas como *TomTom Traffic Index* [13] que entregan información acerca del tráfico de diversas ciudades del mundo. Sin embargo, la información que presentan no ayuda en la toma de decisiones sobre el tránsito, ya que son datos de carácter general y

sirven para comparar distintas ciudades. Entrega datos que hablan de la ciudad pero no son lo suficientemente específicos como para determinar qué calle de una comuna es la que presenta más tráfico o accidentes. El sitio realiza un ranking anual del estado del tráfico en los países en que la plataforma se encuentra presenta. El ranking tiene un orden descendente, donde el primer puesto lo tiene la ciudad que más tiempo promedio pierde anualmente en estancamientos una persona. El primer lugar lo tiene Estambul, con 142 horas perdidas en el año, mientras que chile ocupa el puesto 26 con 89 horas. El ranking tiene un total de 404 ciudades participantes.

También hay aplicaciones como *Google Maps* [10], *Here* [12] y *TomTom* [17] que entregan información y mapas de la ciudad a través de sus API's usando distintos modelos de cobro. Por un lado el servicio de *Google Maps* para desarrolladores no permite acceder a los datos del tránsito en tiempo real, aunque sí maneja esta información y la usa para dar respuestas más exactas en sus otros servicios como por ejemplo el cálculo de una ruta. *TomTom* y *Here* sí permiten acceder a los datos del tráfico en tiempo real, ambas cuentan con buenos ejemplos y documentación. De estas dos APIs se ha escogido trabajar con *TomTom*, ya que por las reuniones con la UCI se supo que *TomTom* estuvo trabajando en un piloto junto a ellos, aunque no se tuvo más información de hasta dónde llegó ese piloto ni de qué se trataba específicamente. Es importante destacar que *TomTom* se encuentra muy activo en Chile, como podemos apreciar en su cobertura de mercado [18].

TomTom es una empresa que nació desarrollando GPS para automóviles, que luego se inclinó por el desarrollo de datos y mapas. La empresa cuenta con una división orientada hacia desarrolladores llamada TomTom Developer, que cuenta con APIs de diversos datos, tales como búsqueda, generador de rutas, información del tráfico, entre otros. Al tener una cuenta se otorga una cantidad de 2500 consultas gratis al día, superando esa cantidad se comienza a cobrar 1 dólar cada 2000 consultas y dado que el proyecto no escala a tantas consultas diarias no representa un problema.

En cuanto a los mapas TomTom tiene a disposición un SDK gratuito que sirve para mostrar los mapas en una aplicación web y permite también realizar consultas de forma programática a las distintas APIs que TomTom posee.

2.3. Aplicaciones

En cuanto a las aplicaciones Google Maps y TomTom disponen de aplicaciones destinadas al usuario que necesita calcular la mejor ruta de un punto A al B, sin embargo este tipo de aplicaciones no busca estudiar ni analizar el tráfico de la ciudad, si no que calcular la mejor forma de desplazarse según las condiciones que se están dando en el instante que se consulta. Además permite calcular el tiempo de traslado en un viaje a realizar en un tiempo futuro cercano, como podemos ver en la Figura 2.2, para lo cual servicios como Google Maps debe realizar un análisis histórico de los datos del tráfico y predecir un rango de tiempo de desplazamiento para esa fecha y hora consultada, sin embargo esos datos no quedan disponibles para el usuario, solamente son usados en el cómputo de la consulta.

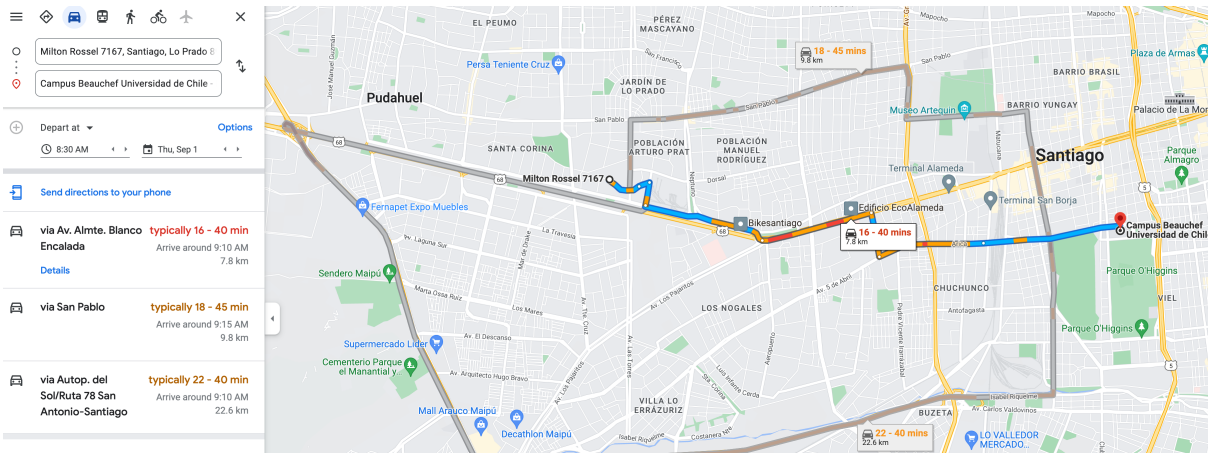


Figura 2.2: Ejemplo de uso de Google Maps para calcular una ruta en un tiempo futuro

Se buscaron en internet plataformas que permitan consultar datos históricos del tráfico pero no se encontró ninguna herramienta de uso masivo que tuviera esta funcionalidad.

2.4. Búsqueda de datos históricos del tráfico

Se realizó una búsqueda de información histórica del tráfico vehicular, sin embargo no se encontraron bases de datos del tráfico en ningún formato, ya sea como una instancia de una base de datos o un archivo de hoja de cálculo. Lo que está muy disponible son mapas de Chile, divisiones territoriales y líneas de calles o carreteras.

Lo más cercano que se encontró fueron datos de los siniestros vehiculares en formato hoja de cálculo ocurridos en Chile desde 1972 al 2021. Algunos de los datos que incluye son la región donde ocurrió, la causa, la fecha, el lugar, cantidad de participantes, entre otros. Estas tablas de datos se encuentran en la Comisión Nacional de Seguridad de Tránsito [8]. En la Tabla 2.1 se puede ver un extracto de la hoja de cálculo `Evolución de siniestros de tránsito Chile (de 1972 a 2021)`.

Lamentablemente estos datos por sí solos no entregan información útil para analizar el tráfico y poder realizar ajustes sobre él. Sin embargo, en el caso de contar con los datos del tráfico y los datos de los siniestros, estos se vuelven mucho más interesantes para el análisis del tráfico vehicular, ya que se pueden encontrar relaciones entre el estado del tráfico en el momento y sector en que se originó el accidente, lo que lleva a tener un mejor análisis de la ciudad, pudiendo llegar a relacionar estados del tráfico con tipos de accidentes. Sin embargo, para tener un análisis mucho más rico, los datos de los siniestros deberían ser específicos de cada accidente, ya que los datos actuales son más estadísticos y no dan una descripción de cada siniestro.

Tabla 2.1: Evolución de siniestros de tránsito Chile (de 1972 a 2021)

Año	**Siniestros**	**Fallecidos**	**Total lesionados**	**Total víctimas**	**Tasa motorización**	**Vehículos cada 100 habitantes**	**Parque vehicular**	**Población**	**Fallecidos cada 100 siniestros**	**Siniestros por cada fallecido**
2017	94.879	1.483	62.171	63.654	3,5	28,3	5.190.704	18.373.917	1,56	63,98
2018	89.311	1.507	57.939	59.446	3,4	29,6	5.498.895	18.552.218	1,69	59,26
2019	89.983	1.617	57.749	59.366	3,3	29,9	5.718.409	19.107.216	1,80	55,65
2020	64.707	1.485	42.103	43.588	3,5	28,7	5.591.145	19.458.310	2,29	43,57
2021	80.751	1.688	51.928	53.616	3,2	31,0	6.102.351	19.678.363	2,09	47,84

Capítulo 3

Análisis y Diseño

En la sección 3.1 se muestran las distintas funcionalidades que ofrece el SDK de TomTom, dentro de las cuales sólo algunas serán utilizadas para la solución realizada en esta memoria. En la sección 3.2 se analizan los datos que ofrece TomTom, qué funcionalidades del SDK serán utilizadas y cómo se utilizarán los mismos datos utilizados en TomTom Traffic Index para aportar un nuevo enfoque geográfico en VisTraffic. En la sección 3.3 se describe la forma que tendrá la solución, cuáles y cómo serán sus componentes y la forma de comunicarse entre ellas.

3.1. Exploración SDK de TomTom

Se han realizado diversas pruebas al SDK de TomTom con el objetivo de aprender a realizar llamadas programáticas a la API y a representar las respuestas en el mapa de Santiago. En la Figura 3.1 se puede apreciar esta exploración, representando en el mapa las respuestas de la Traffic API y la Routing API. Aunque este avance fue exploratorio para relacionarse con el SDK de TomTom, algunos desarrollos de esta exploración fueron utilizados durante el desarrollo de la memoria.

Para la exploración se optó por un diseño sencillo, que cuenta con una barra lateral dividida en dos secciones: el cálculo de la ruta más rápida en la parte superior y una lista de los incidentes en tiempo real en la parte inferior, los cuales se pueden ordenar respecto a su tiempo de espera o longitud del estancamiento.

Acompañando a esta barra lateral se encuentra un menú de marcadores, lo que permite agregar un marcador al mapa que despliega un mensaje al ser seleccionado.

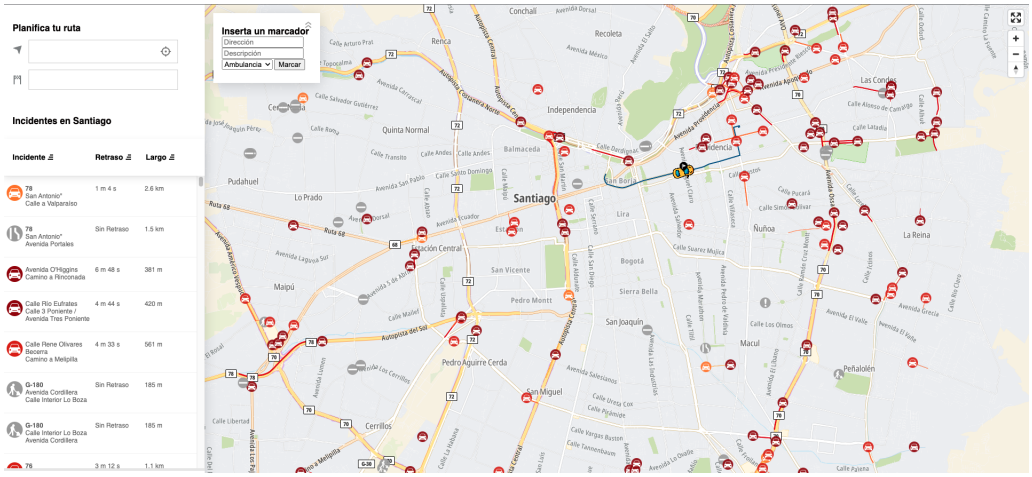


Figura 3.1: Maqueta para probar funciones del SDK de TomTom.

3.1.1. Cálculo de ruta

Se implementó la funcionalidad de cálculo de la ruta más rápida usando el servicio Routing API. La ruta es calculada desde un punto A a un punto B, ingresando la dirección de cada punto en un searchBox incluido en el SDK, el cual tiene diferentes configuraciones como sugerir direcciones mediante un auto-completar y poder limitar estas sugerencias sólo a Chile. Al ser consultada la ruta se ajustan los márgenes del mapa para centrar la ruta, se pinta el camino de color azul sobre el mapa y se entrega un pop-up con datos sobre la ruta. En la Figura 3.2 podemos observar las características descritas.

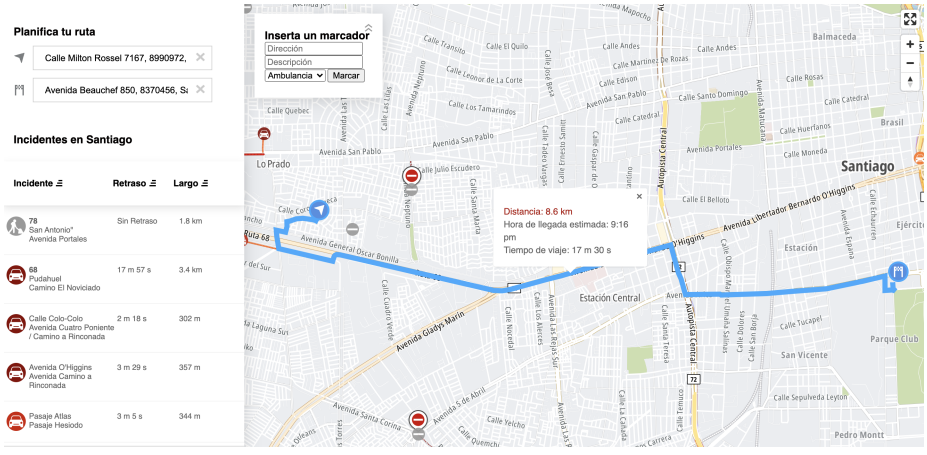


Figura 3.2: Cálculo de ruta mas rápida

3.1.2. Incidentes en el mapa

En la barra lateral hay una lista con los accidentes en la ciudad de Santiago en tiempo real, estos accidentes se pueden ordenar según largo del estancamiento o por la duración de éstos. En el mapa son marcados con distintos símbolos los cuales representan el tipo de incidente. La Traffic API de TomTom reconoce los siguientes tipos: Desconocido, Choque,

Neblina, Condiciones Peligrosas, Lluvia, Hielo, Estancamiento (Taco), Vía Cerrada, Calle Cerrada, Trabajos en la vía, Inundación y Vehículo Averiado.

El estancamiento tiene distintos niveles de magnitud y son representados con el mismo símbolo pero con distinto color de fondo. Los niveles de magnitud son *Tráfico Lento* con fondo amarillo, *Tráfico en Cola* con fondo naranjo y *Tráfico Parado* con fondo rojo. Todos los demás tipos de incidentes tienen un símbolo y fondo gris. Todo esto se puede apreciar en la Figura 3.1, dónde se puede ver distintos tipos de incidentes.

Al seleccionar uno de estos símbolos se entregarán más detalles acerca del incidente en un pop-up, lo que se puede apreciar en la Figura 3.3.

Figura 3.3: Información de un incidente, en este caso, estancamiento de alta magnitud

Esto resultó útil para el desarrollo de la plataforma, ya que fue necesario marcar puntos en el mapa.

3.1.3. Marcadores en el mapa

Junto a la barra lateral se encuentra un menú de marcadores, el cual permite ingresar en una dirección un marcador acompañado de una descripción y un icono que mejor represente la situación. El marcador al ser seleccionado en el mapa despliega un pop-up con la descripción ingresada, lo cual se puede observar en la Figura 3.4.

Figura 3.4: Marcador de semáforo averiado

3.1.4. Animación en el mapa

Dentro de la exploración al SDK también se trabajó en la representación de objetos que se muevan en el mapa. Para esto se ocupó el icono de un auto que recorre una ruta fija, el cual después recorre la misma ruta en sentido contrario, siguiendo la idea de un loop por sobre la ruta marcada en el mapa como se puede ver en la Figura 3.5.

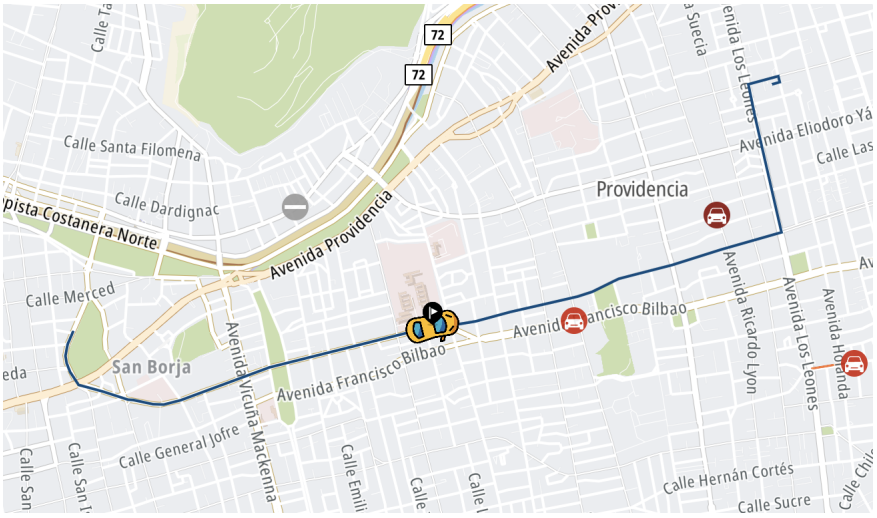
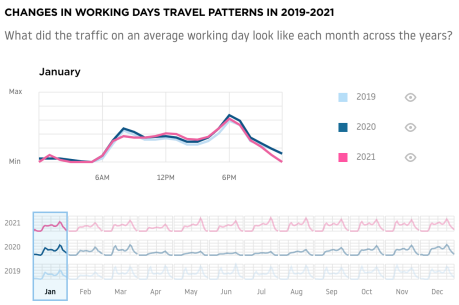


Figura 3.5: Marcador de semáforo averiado

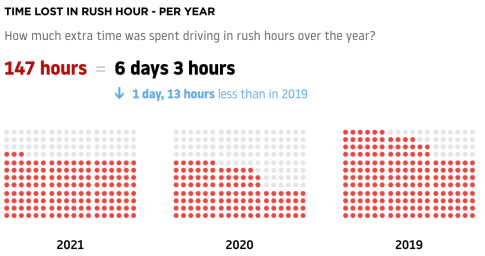
3.2. Análisis

Al comparar lo que brinda TomTom con las otras plataformas mencionadas en el capítulo de Estado del Arte, notamos que es el único servicio que entrega información de los incidentes ocurridos en la ciudad en tiempo real a través de una API, por lo que existe una buena oportunidad de poner a disposición esta información, ya que otros servicios de internet lo muestran en tiempo real, pero no existe una forma directa (como a través de una API) de conseguir estos datos. El simple hecho de almacenar esta información ya presenta una oportunidad interesante para estudiar cómo se comporta el tráfico en la ciudad de Santiago.

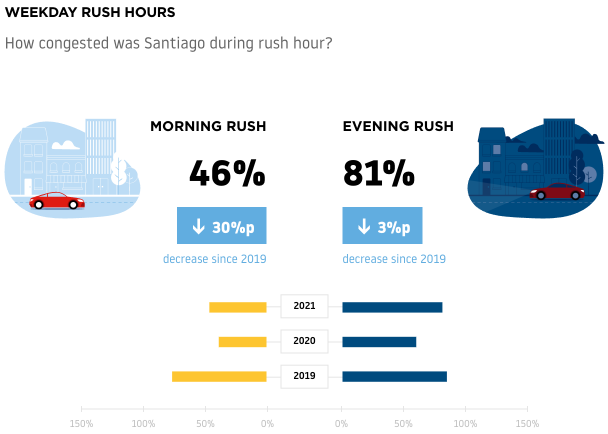
Si bien TomTom Traffic Index usa los datos de Traffic API, los representa de forma estadística a través de gráficos, esquemas e infografías. Algunos ejemplos de los datos que representan son: comparar el horario puntas de la mañana con el de la tarde, el número de horas anuales perdidas por desplazarse en horario punta, gráfico de líneas que compara el tráfico de un mes a través de los años. En la Figura 3.6 podemos observar algunos de los datos entregados por TomTom Traffic Index para la ciudad de Santiago.



(a) Cambios en los patrones de viajes en días hábiles en 2019-2021



(b) Tiempo perdido en hora punta - por año



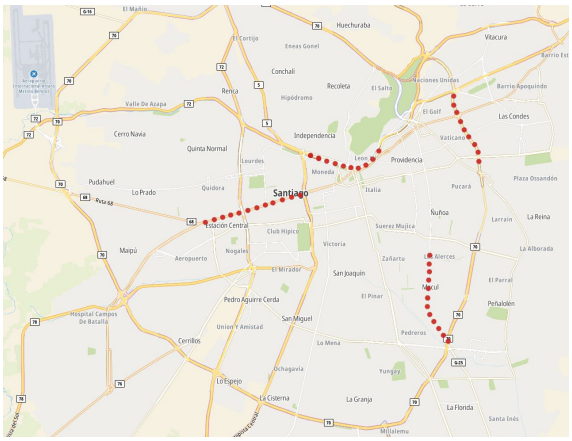
(c) Horas punta en días hábiles

Figura 3.6: Gráficos de los datos de Santiago presentados en TomTom Traffic Index

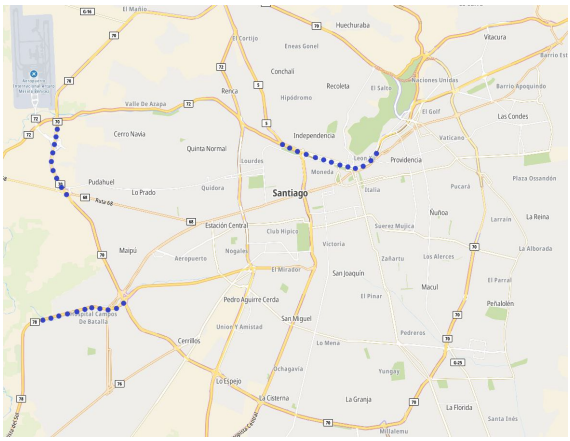
Dado esto, el objetivo es poner a disposición los datos de Traffic API, específicamente los entregados por el endpoint Incident Details, pero de una forma distinta a como lo hace TomTom Traffic Index. Ya que cada incidente contiene los datos geográficos del lugar en

que ocurre, se decidió representar estos datos con un sentido más geográfico, visualizando la locación geográfica de los incidentes en el mapa de Santiago. Los incidentes además serán acotados solamente a estancamientos vehiculares (o tacos).

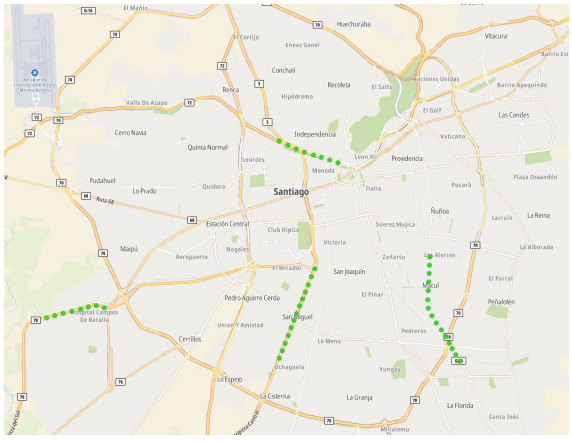
Lo primero que se debe hacer es comenzar a almacenar los datos, para esto se utilizará una base de datos no relacional, ya que por la naturaleza de los datos no hay una relación entidad a la que se pueda sacar ventaja en una base de datos relacional. Los datos serán consultados cada 10 minutos, tiempo que permite no tener datos tan similares entre cada consulta a la API. Además es un margen de tiempo suficiente para captar los tacos de magnitud importante, ya que los tacos que duren menos de 10 minutos y no sean captados en los datos realmente no presentan problemas comparados a los tacos que aparezcan en dos registros adyacentes. Este tiempo también permite no incurrir en cobros por parte de TomTom, ya que la cantidad de consultas diarias está dentro del rango que se entrega de forma gratuita.



(a) Registro 1 (Tacos representados con puntos rojos)



(b) Registro 2 (Tacos representados con puntos azules)



(c) Registro 3 (Tacos representados con puntos verdes)

Figura 3.7: Registros consecutivos con 10 minutos de diferencia

Para mostrar información de locación geográfica de los tacos hay que diseñar el tipo de consulta a realizar sobre los datos. Como se restringieron los datos a solamente tacos, el objetivo de la consulta y la visualización será determinar los peores tacos, los más problemáticos,

en un rango de tiempo. Para esto necesitamos definir una métrica de qué es un peor taco con tal de poder entregarlos ordenados. Esta métrica será la frecuencia con que un taco aparece en cada registro de la base de datos, entendiendo que un registro es una entrada a la base de datos y están separados por 10 minutos entre sí. Sin embargo a través de los registros un taco puede ir evolucionando e ir cambiando de forma, por lo que para identificar un taco se acumularan los puntos de todos los registros que se encuentran en el rango de tiempo en un mismo plano, sobre el cual se ejecutará un algoritmo de agrupamiento para identificar los tacos.

Para entender mejor esta métrica se tiene un ejemplo en imágenes. Los datos de las imágenes no son reales, solo son para ejemplificar cómo se define un taco y cómo se ordenan entre ellos.

Lo primero que tenemos son las imágenes de 3 registros consecutivos, registros que tienen 10 minutos de diferencia entre ellos. Estas se pueden observar en la Figura 3.7.

La superposición de los puntos en el mismo plano se ve de la siguiente forma. En la Figura 3.8 podemos observar primero la superposición del registro 1 y registro 2.

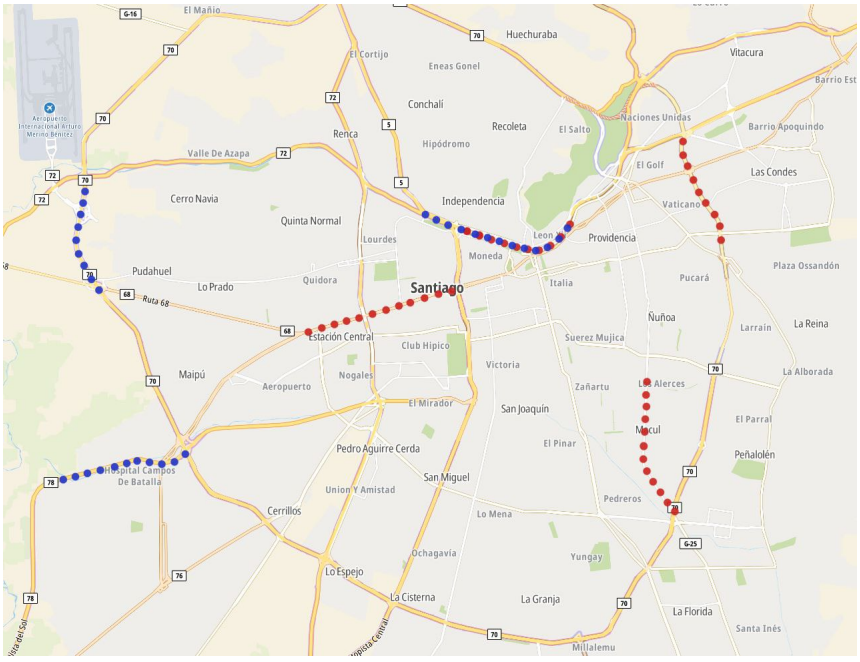


Figura 3.8: Superposición de registro 1 y registro 2

Luego, a esa superposición se agrega el siguiente registro, lo que se puede ver en la Figura 3.9.

Si estos fueran todos los registros dentro del rango de tiempo a consultar el siguiente paso es ejecutar un algoritmo de agrupación sobre el plano resultante de las superposiciones, lo cual entregaría como resultado los tacos que tuvieron mayor frecuencia a través de los registros incluidos en el rango de tiempo. En la Figura 3.10 podemos observar un posible resultado de ejecutar un algoritmo de agrupamiento sobre el plano final de la superposición.

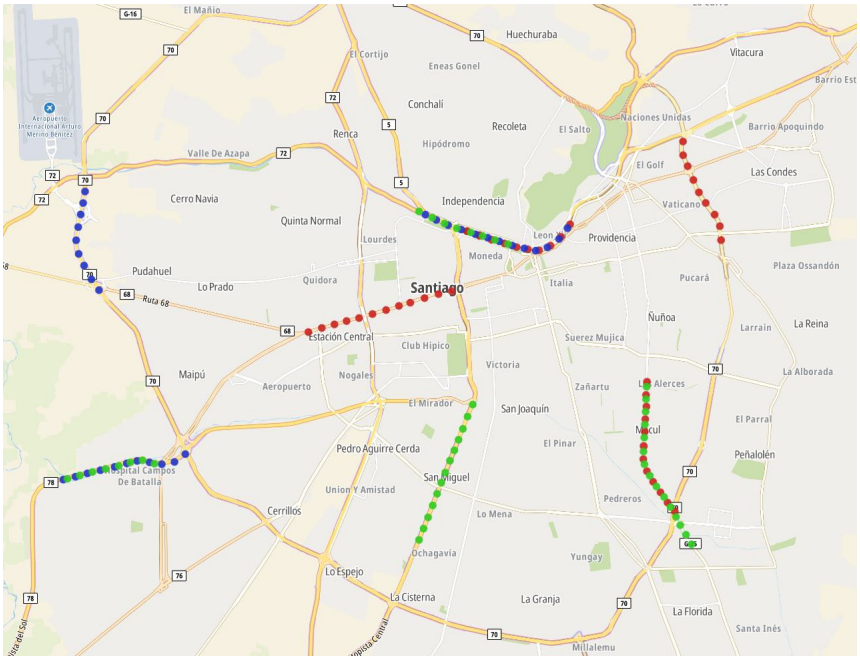


Figura 3.9: Superposición de registro 1, registro 2 y registro 3

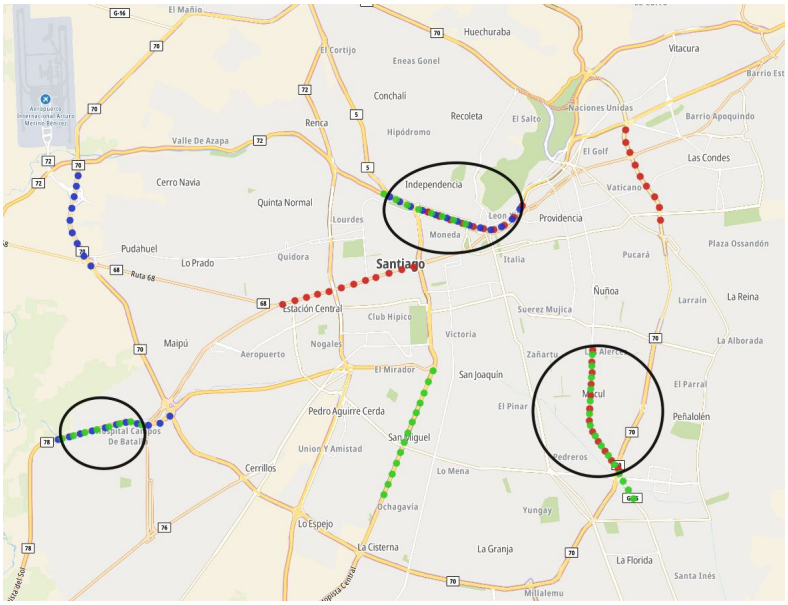


Figura 3.10: Resultado de ejecutar algoritmo de agrupamiento sobre el plano final de la superposición

En cuanto al algoritmo de agrupamiento se analizaron varias opciones, como k-means, DBSCAN, OPTICS, entre otros. Se decidió utilizar HDBSCAN ya que es un algoritmo basado en densidad, donde los datos se agrupan por áreas de altas concentraciones de puntos de datos, es decir, el algoritmo encuentra los lugares que son densos en puntos de datos y los llama grupos. Además, los algoritmos de agrupamiento de este tipo no toman en cuenta los valores atípicos en los grupos, por lo que estos datos se toman como ruido.

3.3. Diseño

Como el objetivo es poner a disposición información sobre el tráfico de la ciudad, se propone utilizar una arquitectura web, ya que una aplicación web es de fácil acceso y utilización, ya que no se necesita instalar ni descargar ningún tipo de software. En la Figura 3.11 podemos ver un diagrama de la arquitectura propuesta.

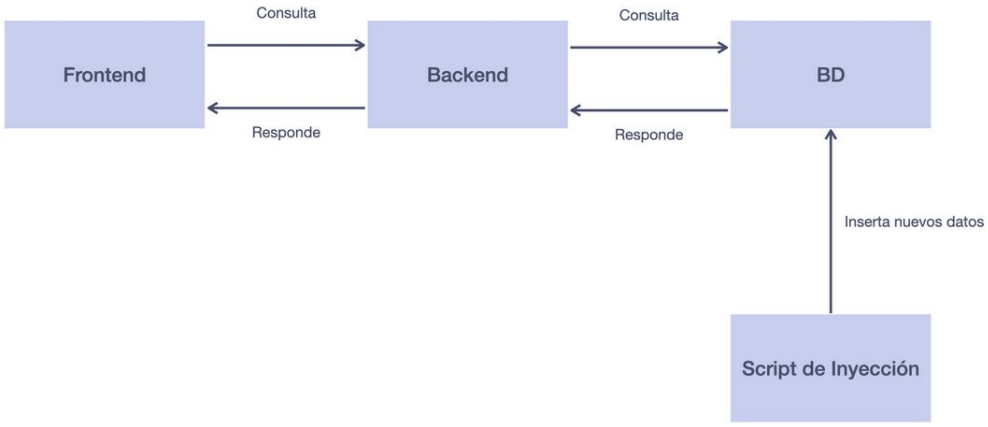


Figura 3.11: Arquitectura Web

El objetivo del Frontend es enviar los parámetros de consulta sobre los datos, como por ejemplo el rango de tiempo en que se consultarán los registros. Estos datos llegan al Backend, que se encarga de consultar la base de datos a partir de los parámetros recibidos para luego ejecutar el algoritmo de agrupamiento sobre el resultado de la consulta. La base de datos es de tipo no relacional y su principal función es almacenar los datos del tráfico. El script de inyección provee de nuevos datos a la base de datos cada 10 minutos.

3.3.1. Backend

El Backend es el sistema encargado de consultar los datos, procesarlos y responder los tacos para ser marcados en el Frontend. El lenguaje propuesto para construir este sistema es Python, ya que dentro de este lenguaje encontramos un módulo con la implementación de HDBSCAN, un módulo para trabajar los Datos como Dataframes y un framework web como Flask que simplifica conectarse al Backend por medio del protocolo HTTP.

Con el objetivo de poder incorporar nuevas consultas o nuevas funcionalidades de forma más simple se diseñó una estructura de software para el Backend que sigue la idea de Clean Architecture. La Figura 3.12 muestra un esquema de la arquitectura propuesta.

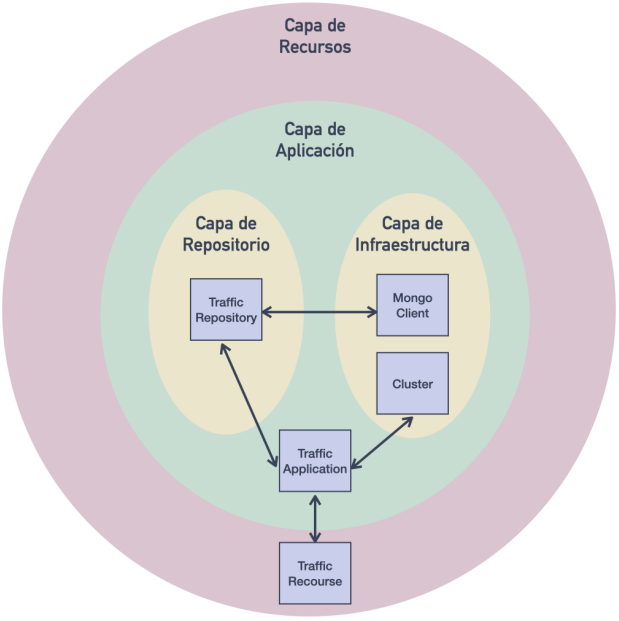


Figura 3.12: Arquitectura de Software del Backend

Para que este sistema pueda ser implementado se utilizará una extensión del framework Flask llamado flask-injector, el cual brinda la capacidad de implementar inyección de dependencias al sistema, lo que permite que cada capa de la arquitectura quede lo más encapsulada posible y solo pueda comunicarse con capas internas. Este diseño sigue muy bien los principios SOLID, ya que como vemos en la Figura 3.12 se cumple:

- **Single-responsability**: Cada clase del sistema tiene sólo una responsabilidad. Esto también viene acompañado de la responsabilidad de la clase que está asociada con la capa a la que pertenece.
- **Open-closed**: Para agregar nuevas funcionalidades basta con extender el sistema más que modificar lo establecido, es decir, en caso de nuevos tipos de consultas basta con crear nuevas clases en sus capas correspondientes sin tener que ir a modificar las existentes.
- **Interfaces segregation**: Cada clase del sistema debe implementar su propia interfaz.
- **Dependency inversion**: Cada clase recibe en su constructor un objeto instanciado de las capas interiores con las que se comunica.

Como vemos en la Figura 3.12, los cuadros violetas y las flechas representan el flujo en el Backend el cual comienza por su capa más externa, representada por la clase TrafficResource, encargada de recibir la llamada HTTP junto con los datos de los parámetros de la consulta. Esta capa se comunica con la Capa de Aplicación a través de la clase TrafficAplication, la cual recibe los datos, los manipula y se comunica con la Capa de Repositorio, donde la clase TrafficRepository se encarga de comunicarse con la Capa de Infraestructura y consultar a la base de datos a través del MongoClient incluido en el módulo pymongo. Luego de efectuar la

consulta a la base de datos TrafficRepository entrega su respuesta a TrafficAplication quien ahora se comunica con la Capa de Infraestructura a través de la clase Cluster para manipular la respuesta de la base de datos y aplicar el algoritmo de agrupamiento sobre los datos. Cuando TrafficAplication recibe la respuesta del agrupamiento, da formato a los datos y los entrega a Capa de Recursos para que la respuesta llegue al Frontend a través del protocolo HTTP.

3.3.2. Base de datos

La base de datos del sistema es no relacional, esto se diseñó así ya que los datos recopilados son series de tiempo, por lo que no tenemos relaciones entre entidades y la cantidad de entradas crece constantemente. Además, el tener una base de datos de este tipo brinda la libertad de poder incorporar nuevos campos de datos a los registros sin interferir con los ya ingresados, lo que entrega más flexibilidad al sistema, ya que para realizar nuevas o mejores consultas se podrían necesitar nuevos campos de información. Los datos recibidos por Traffic API son una lista de incidentes, donde cada incidente es un objeto JSON que describe las propiedades del incidente como también su geometría.

Se propone implementar una base de datos MongoDB, donde cada entrada consta de un id, un objeto Datetime con la fecha y hora del momento en que se ingresa la entrada a la base de datos y una lista de objetos JSON que describen cada incidente. El diagrama de la Figura 3.13 representa la estructura de la base de datos.

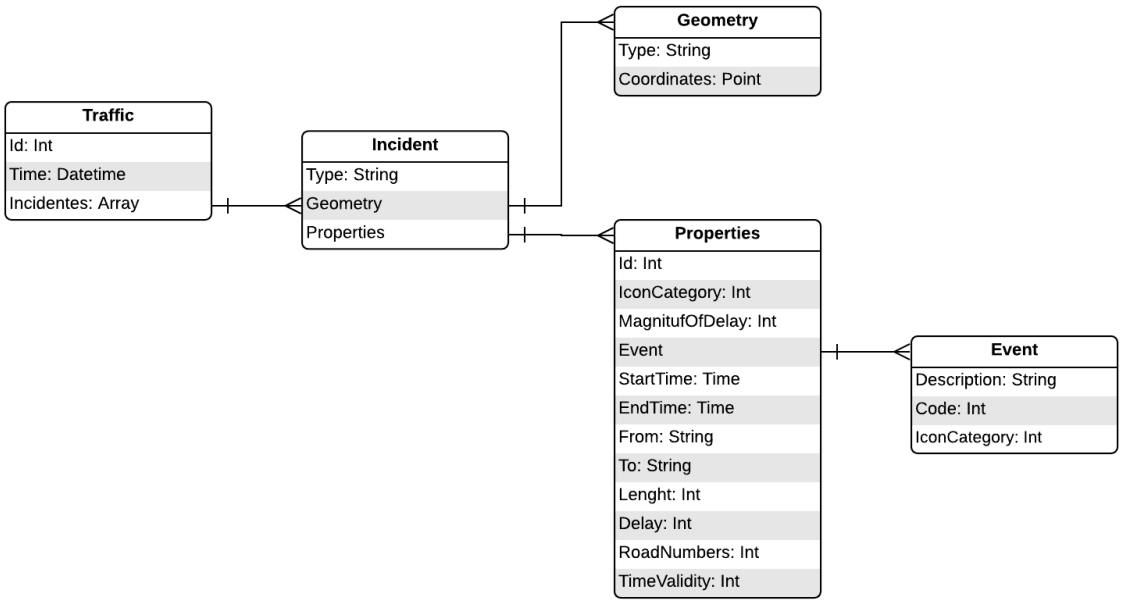


Figura 3.13: Estructura de la base de datos

En cuanto a los parámetros para consultar a la base de datos se diseñaron dos tipos de consultas, ambas incluyen filtros para solo tomar los incidentes que son estancamientos, filtrar sobre la comuna donde ocurren los incidentes, fijar un largo de estancamiento mínimo y un tiempo de espera mínimo, es decir, no llegarán como respuesta estancamientos con tiempo de espera menor al tiempo de espera mínimo ni estancamientos con largo menor al mínimo. Donde difieren ambas consultas es con respecto al rango de tiempo, una consulta toma una fecha de inicio y una fecha de fin, tomando todos los incidentes que se encuentren en ese rango. Por otro lado, la segunda consulta también toma una fecha de inicio y fin pero también recibe rangos de hora, por lo que solo toma los accidentes que se encuentran dentro del rango de fechas y en los rangos de hora especificados.

En resumen, se tendrán dos tipos de consultas, una que incluye un rango de horario y necesita los siguientes parámetros:

- Fecha inicial
- Fecha final
- Hora inicial
- Hora final
- Comuna
- Largo mínimo
- Tiempo de espera mínimo

Y una consulta sin rango horario, cuyos parámetros deben ser los siguientes:

- Fecha inicial
- Fecha final
- Comuna
- Largo mínimo
- Tiempo de espera mínimo

3.3.3. Agrupamiento

Luego de consultar los datos y obtener la colección de accidentes se itera sobre ellos con el fin de acceder a la geometría de cada uno. La geometría de un incidente es una lista de puntos, donde un punto es un par (x,y) que hace referencia a latitud y longitud en el mapa. De cada incidente se extraen sus puntos y son llevados a una lista que contiene todos los puntos de todos los incidentes.

Además de obtener esta lista, también se guardan otras listas para tener una relación entre cada punto y los datos del incidente al que pertenece, en otras palabras, para cada punto se guarda la fecha y hora, largo y retraso del estancamiento al que el punto pertenece.

El algoritmo de agrupamiento HDBSCAN se aplica sobre la lista que contiene todos los puntos de todos los incidentes, obteniendo como resultado un agrupamiento de todos estos puntos. HDBSCAN tiene la capacidad de ordenar los grupos que encuentra según la densidad de puntos que cada grupo tiene, por lo que la respuesta del algoritmo viene con los grupos (o tacos) ordenados desde el más al menos denso.

3.3.4. Frontend

El Frontend tiene como objetivo principal entregar los parámetros al Backend para que se realice la consulta a los datos y marcar en el mapa los tacos que el Backend entrega como respuesta. En cuanto a su diseño se propone el que aparece en la Figura 3.14.

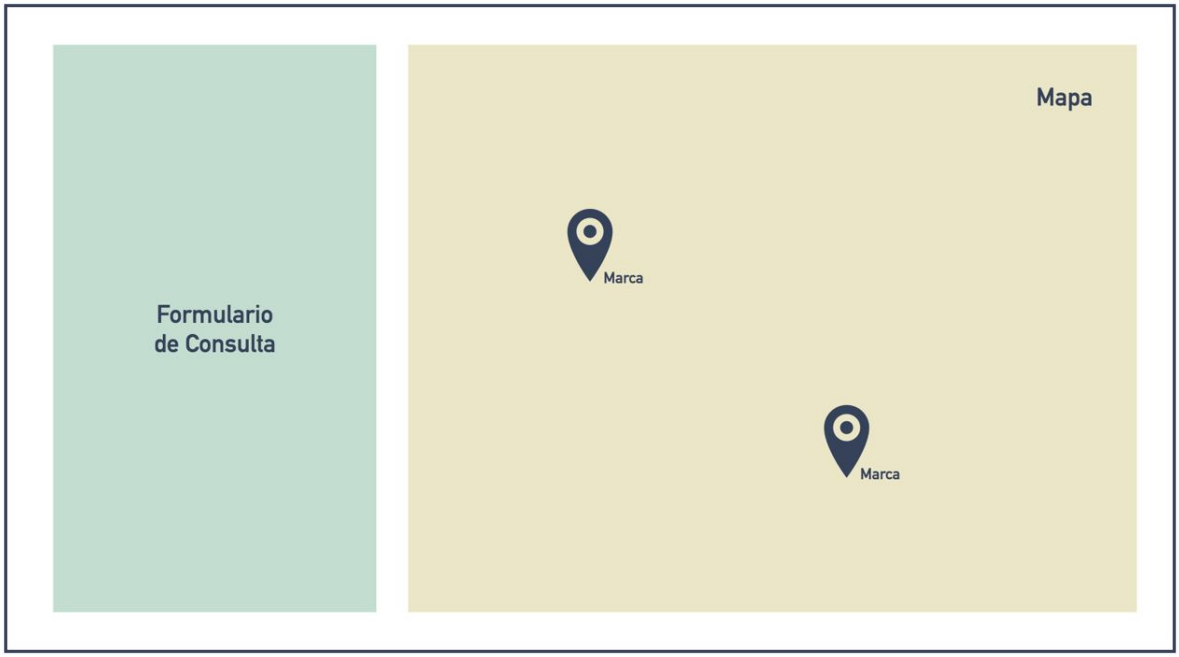


Figura 3.14: Diseño Frontend

Al lado izquierdo se incluye una barra lateral en la que debe ir un formulario, en el cual el usuario puede ingresar a través de campos de input los parámetros de la consulta. Al lado derecho se observa el mapa, en el cual se marcan los tacos que entrega el Backend como respuesta a la consulta.

3.3.5. Alimentación de Datos

El endpoint con el que se consulta a la Traffic API de TomTom tiene la siguiente forma:

Parámetros Requeridos:

- **URL Base**: Url de base que identifica la API a ser consultada
- **Version**: Versión de la API a la que se está llamando
- **Bbox**: Las esquinas del área geográfica sobre la que se hará la consulta
- **Llave**: Llave privada de cada usuario para comunicarse con la API

Parámetros Opcionales:

- **Campos**: Define los campos de cada incidente incluido en la respuesta
- **Lenguaje**: Define el lenguaje en que se espera la respuesta de la API
- **Filtro de categoría**: Filtra los incidentes de la respuesta según su tipo.

Cada llamada a la Traffic API responde con una lista de incidentes encontrados en el área sobre la que se consultó.

3.4. Resumen

En este capítulo se mostró el trabajo de exploración del SDK de TomTom y se analizó la forma en que se pondrá a disposición la información de Traffic API, optando por una representación geográfica de los peores tacos en Santiago. Finalmente en la sección de Diseño se describió el sistema a implementar con sus componentes principales.

Capítulo 4

Implementación

En la sección 4.1 se describe la implementación del sistema encargado de consultar y manipular los datos, obteniendo la lista jerarquizada de tacos a marcar en el mapa. La sección está subdividida en la arquitectura de capas con que se implementó el Backend. En la sección 4.2 se describe el sistema que se encarga de realizar la consulta al Backend y visualizar la respuesta de este. Finalmente en la sección 4.3 se describe el esquema de la base de datos, como se consiguen y su almacenamiento.

La implementación se puede encontrar en el siguiente repositorio de Github `https://github.com/Grynn01/vistraffic`

4.1. Backend

El Backend fue construido usando el framework Flask y el lenguaje de programación Python. El Backend es ejecutado en un computador MacBook Air M1 2020, con un procesador Apple M1 y 8GB de memoria RAM.

Como se dijo en la sección de Análisis y Diseño, el Backend ha sido construido siguiendo una arquitectura de software del tipo Clean Architecture u Onion Architecture, por lo que describiremos el código de cada una de sus capas.

4.1.1. Capa de Recurso

La capa de Recurso es la encargada de recibir el método HTTP POST. En esta implementación la capa de recurso está formada por la clase `TrafficResource`. A través del método POST se reciben los datos del formulario y se transforman en una variable de tipo dictionary. Luego, esta variable llamada `query_data` se entrega a un DTO.

DTO

DTO es un acrónimo de `Data Transfer Object`. En este caso el objetivo del DTO es transformar una variable de tipo `diccionario` en una variable del tipo `TrafficQuery`, permitiendo un manejo mucho más simple de los datos que llegan desde el formulario del Frontend. Además, la clase `TrafficQuery` posee un método llamado `format` encargado de dar formato correcto de los datos, transformándolos al tipo que corresponda para cada uno de ellos. Podemos ver la implementación del DTO se encuentra en el Código 4.1

```
@dataclass
class TrafficQuery:
    start_date: datetime
    finish_date: datetime
    top_jams: int
    commune: str
    length: int
    delay: int
    start_time: int = None
    finish_time: int = None
    server_delay: int = 4

    def format(self):
        self.start_date = utils.adjust_to_server_datetime(
            parser.isoparse(self.start_date)
        )
        self.finish_date = utils.adjust_to_server_datetime(
            parser.isoparse(self.finish_date)
        )
        self.top_jams = int(self.top_jams)
        self.length = int(self.length)
        self.delay = int(self.delay)
        if self.start_time:
            self.start_time = int(self.start_time.split(":")[0]) + self.
    server_delay
        if self.finish_time:
            self.finish_time = int(self.finish_time.split(":")[0]) + self.
    server_delay
```

Código 4.1: TrafficQuery (DTO)

Luego de entregar la variable dictionary al DTO y aplicar su método format se obtiene una variable del tipo TrafficQuery, la cual es entregada a la Capa de Aplicación para que continúe con el proceso.

Al obtener una respuesta desde la Capa de Aplicación se verifica que la respuesta sea distinta de `None`, para finalmente armar la respuesta usando el método `make_reponse` de Flask y que estos datos lleguen al Frontend a ser marcados en el mapa. Esta lógica se observa en el Código 4.2

```python
def post(self):
    query_data = request.form.to_dict()
    if not query_data:
        return {
            "error_type": "MISSING_PARAMETERS",
            "error_message": "La consulta no contiene todos los parametros requeridos",
        }
    try:
        query_data_dto = TrafficQuery(**query_data)
        query_data_dto.format()
        response_data = self.traffic_app.get_jams(query_data_dto)
    except TrafficApplicationException as e:
        return {
            "error_type": str(e.error_type),
            "error_message": str(e.message),
        }, 500

    if not response_data:
        return {
            "error_type": "QUERY_ERROR",
            "error_message": "No se pudo realizar la consulta sobre nuestra base de datos",
        }, 500

    response = make_response(response_data)
    response.headers["Access-Control-Allow-Origin"] = "*"
    response.headers["Access-Control-Allow-Credentials"] = True

    return response
```

Código 4.2: Método post de TrafficResource

Para seguir de mejor manera la arquitectura de software planteada el proyecto ha sido construido en base a inyección de dependencias, por lo que la Capa de Recursos recibe en directamente una instancia de la clase `TrafficAplication`, que es la clase que representa a la Capa de Aplicación. Esto se aprecia en el Código 4.3

```python
class TrafficResource(Resource):
    @inject
    def __init__(self, traffic_app: TrafficApplicationInterface):
        self.traffic_app = traffic_app
```

Código 4.3: Constructor de TrafficResource

4.1.2. Capa de Aplicación

El objetivo de la Capa de Aplicación es recibir la variable `query_data` del tipo `TrafficQuery` y usarla para computar la respuesta, realizando llamados a la Capa de Recurso y a la Capa de Infraestructura. En este caso la Capa de Aplicación está representada por la clase TrafficAplication. En el Código 4.4 está el constructor de esta clase.

```python
class TrafficApplication(TrafficApplicationInterface):
    @inject
    def __init__(
        self, traffic_repo: TrafficRepositoryInterface, cluster_client:
    ClusterInterface
    ):
        self.traffic_repo = traffic_repo
        self.cluster_client = cluster_client
```

Código 4.4: Constructor de TrafficApplication

Al igual que la Capa de Recurso la Capa de Aplicación recibe una instancia de la Capa de Repositorio, representada por la clase `TrafficRepository` y una instancia de la clase `Cluster`, la cual pertenece a la Capa de Infraestructura.

La lógica principal de la clase `TrafficAplication` radica en su método `get_jams`, el cual tiene como objetivo consultar la base de datos, filtrar según la información que contiene la `query_data` y almacenar esa información en un conjunto de variables.

Lo primero que realiza el método `get_jams` es revisar la información que trae `query_data` para determinar qué método de `TrafficRepository` consultar. Desde cualquiera que sea el método llamado llegarán 4 variables:

1. `points_to_cluster`: Lista de elementos de tipo `Point`, los cuales representan coordenadas (x,y) en el mapa.
2. `points_incident_length`: Lista de enteros que representan el largo del `taco` en metros.
3. `points_incident_delay`: Lista de enteros que representan el retraso del `taco` en segundos.
4. `points_incident_datetime`: Lista de elementos de tipo `datetime` que contienen la fecha y hora a la que se observa el `taco`.

Todas estas variables comparten índice, es decir, sea un índice `i` se tiene que para el punto `points_to_cluster[i]`, el incidente al cual pertenece tiene largo `points_incident_length[i]`, tiene un delay `points_incident_delay[i]` y la fecha y hora del incidente al que pertenece es `points_incident_datetime[i]`.

Las listas `points_incident_length`, `points_incident_delay` y `points_incident_datetime` son insertadas en una variable llamada `points_info_dict` de tipo `dictionary`. Este diccionario se entrega y la lista `points_to_cluster` son entregados a la instancia de la clase `Cluster`, donde se realiza un agrupamiento de los puntos, donde cada grupo formado es identificando como un `taco`. La respuesta de la clase Cluster es a través de su método `clusterize`, el cual entrega un `Dataframe`, que es transformado a formato JSON y retornado a la Capa de Recurso.

4.1.3. Capa de Repositorio

La Capa de Repositorio está encargada de consultar a la base de datos. Esta capa está representada por la clase TrafficRepository. Su constructor recibe una instancia de la clase MongoConnection, que pertenece a la Capa de Infraestructura, la cual contiene el método `get_traffic_collection` encargado de conectarse y consultar la base de datos. Esto se observa en el Código 4.5

```
class TrafficRepository(TrafficRepositoryInterface):
    @inject
    def __init__(self, mongo_client: MongoClientInterface) -> None:
        self.mongo_client = mongo_client
```

Código 4.5: Constructor de TrafficRepository

Esta clase contiene varios métodos, donde cada uno representa un tipo de consulta. Cada uno de ellos configura un diccionario con los datos de la consulta. Los métodos son los siguientes:

1. `query_jams_by_date`: Consulta los datos del tráfico entre 2 fechas sobre una comuna.
2. `query_jams_by_date_no_commune`: Consulta los datos del tráfico entre 2 fechas sobre toda la ciudad de Santiago.
3. `query_jams_by_date_and_range_time`: Consulta los datos del tráfico entre 2 fechas, filtrando solo las incidentes ocurridos en un rango de horas del día, sobre una comuna.
4. `query_jams_by_date_and_range_time_no_commune`: Consulta los datos del tráfico entre 2 fechas, filtrando solo las incidentes ocurridos en un rango de horas del día, sobre toda la ciudad de Santiago.

En el Código 4.6 podemos ver la implementación de `query_jams_by_date` como ejemplo de uno de estos métodos:

```
def query_jams_by_date(self, start_date, finish_date, commune, length,
    delay):
    pipeline = [
        {"$unwind": {"path": "$incidents"}},
        {
            "$match": {
                "incidents.properties.iconCategory": 6,
                "incidents.properties.comuna": commune,
                "incidents.properties.length": {"$gte": length},
                "incidents.properties.delay": {"$gte": delay},
                "date": {"$gte": start_date, "$lt": finish_date},
            }
        },
        {
            "$group": {
                "_id": "$_id",
                "date": {"$first": "$date"},
                "incidents": {"$push": "$incidents"},
            }
```

```
        },
    ]

    return self._query_bd(pipeline)
```

Código 4.6: Método query_jams_by_date de TrafficRepository

Todos los métodos de `TrafficRepository` llaman al método privado `_query_db`, encargado de realizar la consulta a la base de datos y armar las listas con los puntos y sus datos.

La lógica del método `_query_db`, encontrada en el Código 4.7, es conectarse a la base de datos, realizar la consulta e iterar sobre los resultados. Cada `snap` de la base de datos es una fotografía a los incidentes de Santiago en un instante de tiempo, por lo que cada snap contiene una lista donde cada elemento de ella es un diccionario que describe un incidente. Para cada snap se recorre su lista de incidentes. Cada uno de estos incidentes contiene una lista de puntos (x,y) que marcan su ubicación en el mapa. Cada uno de estos puntos se agregan a la lista `points_to_cluster[i]`, siendo `i` un índice arbitrario. Para cada uno de estos puntos se agrega un elemento a `points_incident_length[i]`, `points_incident_delay[i]` y `points_incident_datetime[i]`, asegurándose que las 4 listas tengan la misma cantidad de elementos, lo que permite que para cada punto ubicado en un índice de `points_to_cluster` se pueda acceder a sus datos consultando las otras listas con el mismo índice.

```
def _query_db(self, pipeline):
    points_to_cluster = []
    points_incident_length = []
    points_incident_delay = []
    points_incident_datetime = []
    Point = make_dataclass("Point", [("x", float), ("y", float)])
    snaps = self.mongo_client.get_traffic_collection("traffic").aggregate(
        pipeline, allowDiskUse=True
    )
    for snap in snaps:
        incidents = snap.get("incidents")
        incidents_datetime = snap.get("date")
        for incident in incidents:
            point_list = incident.get("geometry").get("coordinates")
            incident_length = incident.get("properties").get("length")
            incident_delay = incident.get("properties").get("delay")
            for point in point_list:
                points_to_cluster.append(Point(point[0], point[1]))
                points_incident_length.append(incident_length)
                points_incident_delay.append(incident_delay)
                points_incident_datetime.append(incidents_datetime)

    return (
        points_to_cluster,
        points_incident_length,
        points_incident_delay,
        points_incident_datetime,
    )
```

Código 4.7: Método query_bd de TrafficRepository

4.1.4. Capa de Infraestructura

La Capa de Infraestructura es la encargada de proveer servicios externos a las capas de Aplicación o de Repositorio. En esta capa se encuentran dos servicios, el cliente para conectarse a la base de datos nombrado anteriormente en la Capa de Repositorio y la clase Cluster encargada de realizar el agrupamiento sobre los puntos (x,y) que pertenecen a la consulta.

En esta sección describiremos la clase Cluster, ya que el otro servicio MongoConnection ya fue descrito en la Capa de Repositorio.

La clase Cluster es inyectada en la capa de Aplicación, en la clase TrafficAplication, donde es usada para agrupar los puntos que llegan como respuesta de la consulta a la base de datos mediante TrafficRepository.

El método encargado del agrupamiento es llamado `clusterize`, el cual recibe una lista de puntos `points_to_cluster`, un diccionario que contiene las listas `points_incident_length`, `points_incident_delay` y `points_incident_datetime[i]` con los datos de estos puntos y un entero correspondiente a la cantidad de `tacos` que se quieren identificar(`top_jams`). Los puntos y la información sobre ellos son convertidos en Dataframes para poder trabajar con ellos más fácilmente.

Sobre el Dataframe que contiene a los puntos se aplica el algoritmo HDBSCAN, el cual corresponde a un algoritmo de agrupamiento que permite identificar los conjuntos de puntos que corresponderá a un mismo `taco` debido a su cercanía. El agrupamiento se realiza mediante el método `fit`. Esto entregará una lista del mismo tamaño de `points_to_cluster`, con una etiqueta asociada a cada punto indicando a qué grupo (o `taco`) pertenece. Esta lista es convertida en un Dataframe. Toda esta lógica se encuentra en el Código 4.8.

```
df = pd.DataFrame(points_to_cluster)
clusterer = hdbscan.HDBSCAN(min_cluster_size=30)
clusterer.fit(df)
labels = pd.Series(clusterer.labels_, index=df.index, name="cluster")
labels_df = pd.DataFrame(labels)
```

Código 4.8: Agrupamiento de puntos con HDBSCAN

Se obtiene la cantidad de puntos por grupo, lo que permite conocer los `tacos` más frecuentes debido a que poseen una mayor densidad de puntos. Luego se seleccionan los `n` grupos más densos, donde `n` corresponde a la variable `top_jams`, lo cual se encuentra en el Código 4.9.

```
points_per_cluster = (
    labels_df.pipe(lambda x: x[x.cluster >= 0])
    .assign(point_id=lambda x: x.index)
    .merge(df, left_index=True, right_index=True)
)
top_clusters = labels[labels >= 0].value_counts().head(top_jams).index
```

Código 4.9: Puntos por grupo

Para cada uno de los grupos seleccionados anteriormente, se calcula el centroide entre todos sus puntos, para así tener un punto que represente al grupo. Esta lista de puntos representantes se almacenan en la variable `top_center_points`. La lógica se representa en el Código 4.10.

```
for cluster in top_clusters:
    top_center_points.append(
        self._calculate_centroid(points_per_cluster, cluster)
    )
```

Código 4.10: Cálculo del centroide de un grupo

Se define un Dataframe `top_clusters_df` con los grupos más densos y el punto que los representa, lo que se puede ver en el Código 4.11.

```
top_clusters_df = pd.DataFrame(top_clusters, columns=["cluster"]).assign(
        points=list(zip(top_center_points_x, top_center_points_y))
    )
```

Código 4.11: Dataframe top_clusters_df que contiene los tacos y su punto representante

Por otro lado se modifica el Dataframe `points_per_cluster` filtrando a aquellos puntos que no pertenecen a los grupos de mayor densidad. Esta operación se encuentra en el Código 4.12.

```
top_clusters_point_info_df = (
    points_per_cluster[points_per_cluster.cluster.isin(top_clusters)]
    .merge(points_info, left_on="point_id", right_index=True)
    .merge(top_clusters_df, left_on="cluster", right_on="cluster", how="
   left")
)
```

Código 4.12: Filtrado de puntos que no pertenecen a los grupos de mayor densidad

Para cada grupo de `top_clusters_point_info_df` se busca en sus puntos el momento en que el taco presentó el mayor retraso en tiempo, tomando este momento como el punto más crítico del taco, asociando al grupo la hora, extensión y retraso de este momento.

Finalmente, se retorna un Dataframe que contiene para cada grupo su centroide (el punto que lo representa), el retraso, la extensión y la fecha y hora de su momento crítico.

4.1.5. API

La API solo cuenta con un método HTTP POST, el cual tiene como endpoint la ruta /traffic/, que es respondida por el método `post` en la clase TrafficResource. Este llamado debe ir acompañado por un body con los siguientes campos obligatorios:

1. `start_date`: Desde qué día se quiere consultar los datos.
2. `finish_date`: Hasta qué día se quiere consultar los datos.
3. `start_time`: Hora en que comienza el rango horario a consultar.

4. `finish_time`: Hora en que termina el rango de horario a consultar.
5. `commune`: Nombre de la comuna sobre la que se consultará. El valor `ALL` consulta sobre todas las comunas.
6. `top_jams`: Cantidad de tacos a marcar en el mapa.

Y los siguientes campos opcionales:

1. `length`: Filtra la consulta solo considerando los tacos con extensión igual o mayor al valor entregado. El valor debe ser entregado en metros.
2. `delay`: Filtra la consulta solo considerando los tacos con retraso igual o mayor al valor entregado. El valor debe ser entregado en minutos.

En el Código 4.13 se presenta un ejemplo de body que contiene la totalidad de los parámetros:

```
{'start_date': '2022-06-27',
'finish_date': '2022-06-29',
'start_time': '18:00',
'finish_time': '20:00',
'commune': 'LAS CONDES',
'top_jams': '5',
'length': '0',
'delay': '0'}
```

Código 4.13: Ejemplo de body de consulta a la API

Ante una consulta válida el formato de la respuesta se encuentra en el Código 4.14.

```
{
   "map_config":{
      "center":{
         "lat":-33.40921291546832,
         "lng":-70.54031320284037
      },
      "zoom":12.921112461845077
   },
   "points_incident_delay":{
      "1103":252,
      "36":607,
      "373":796,
      "906":1289,
      "999":329
   },
   "points_incident_length":{
      "1103":657.1522218771,
      "36":557.5804342108,
      "373":1096.4613834203,
      "906":2456.801156526,
      "999":1128.666005986
   },
   "points":{
```

```
        "1103":[-70.5442898214, -33.4061291545],
        "36":[-70.5532275268, -33.4180463632],
        "373":[-70.5557862811, -33.4080221965],
        "906":[-70.5371035015, -33.3856813503],
        "999":[-70.5862266873, -33.404634844]
    },
    "points_incident_datetime":{
        "1103":1656457203492,
        "36":1656456004083,
        "373":1656460202738,
        "906":1656455403761,
        "999":1656457203492
    }
}
```

Código 4.14: Formato de respuesta ante una consulta válida

Esta respuesta es recibida por el frontend, donde `map_config` trae los cambios al zoom y a la posición del centro del mapa, `points` es un diccionario que contiene como llave el `label` del grupo (o `taco`) y su valor es el punto (x,y) que representa su posición geográfica, `points_incident_delay` contiene el retraso de cada taco, `points_incident_length` contiene la extensión de cada taco, `points_incident_datetime` contiene un unix-timestamp que representa el momento crítico de ese taco.

4.2. Frontend

El frontend está conformado principalmente por dos archivos, un archivo html que ordena la disposición de los componentes y su estilo css, junto a un archivo javascript que contiene un conjunto de funciones encargadas de realizar las marcas en el mapa y controlan las interacciones con el formulario.

La disposición de la página sigue lo establecido en el capítulo de Análisis y Diseño, obteniendo la implementación de la Figura 4.1.

Donde en la parte izquierda se tiene una barra lateral que contiene un formulario para realizar consultas sobre los datos y marcar la respuesta en el mapa.

El estado base del formulario es el observado en la Figura 4.2a.

Para poder agregar filtros sobre la extensión y el retraso de los tacos se debe hacer click en el texto `opciones avanzadas`, desplegándose dos nuevos campos numéricos de entrada de información, como se ve en la Figura 4.2b.

Esto se logra usando la librería Jquery, la cual permite manipular el DOM del HTML. El DOM que se genera al cargar la aplicación web tiene los `inputs` de Largo Mínimo y Retraso Mínimo escondidos, por lo que se asigna una función al texto `Opciones avanzadas` que al ser seleccionado cambia el estado de los campos de `input` de ocultos a disponibles. Estas manipulaciones se encuentran en el Código 4.15.

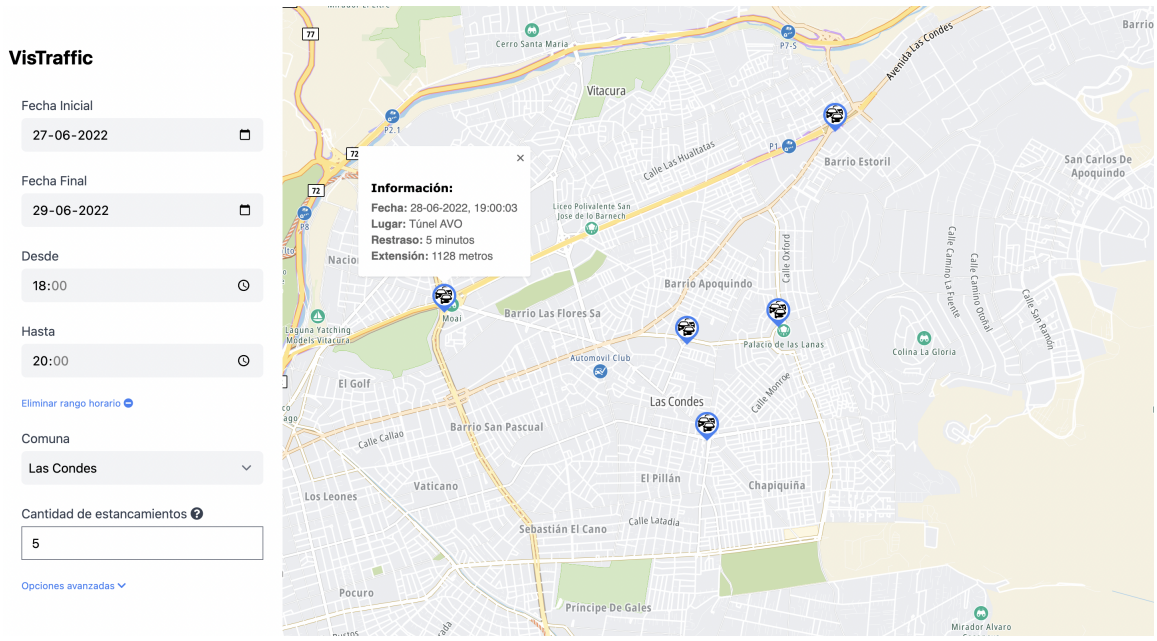


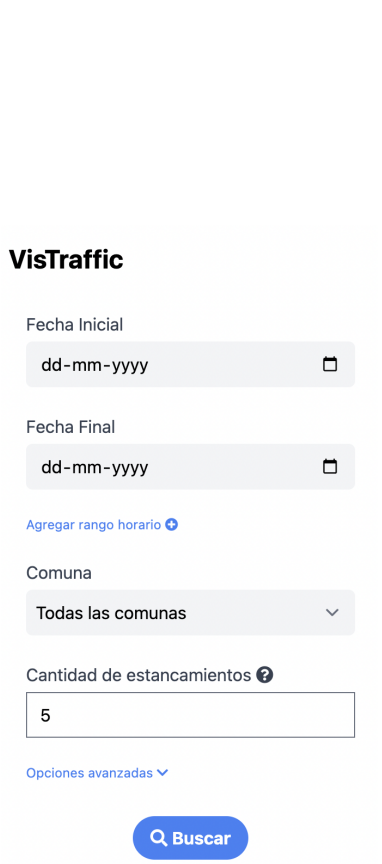
Figura 4.1: Implementación Frontend

```
1
2  $('#advanced_options').click(function() {
3      if(advanced_options){
4          $('#minimum_delay').hide();
5          $('#minimum_length').hide();
6          advanced_options = false;
7          $('#advanced_options').html("Opciones avanzadas <i class=\"fa fa-
       chevron-down\"></i>")
8      }
9      else{
10         $('#minimum_delay').show();
11         $('#minimum_length').show();
12         advanced_options = true;
13         $('#advanced_options').html("Opciones avanzadas <i class=\"fa fa-
       chevron-up\"></i>")
14     }
15 });
```

Código 4.15: Modificaciones para disponer de opciones avanzadas en el formulario

En el caso de necesitar agregar un rango horario a la consulta se debe hacer click en el texto `Agregar rango horario`, desplegándose dos campos de entrada de horarios como se observa en la Figura 4.2c.

Este cambio también se realiza con una función de la librería Jquery, la forma en que se realiza es análoga a la forma en que se muestran las opciones avanzadas. Los inputs de hora que tienen sus minutos fijos a cero, ya que el diseño de la consulta a la base de datos es por rangos de hora que no consideran minutos. El código javascript 4.16 se encarga de fijarlos.



(a) Formulario en estado base

(b) Formulario con opciones avanzadas

(c) Formulario con Rango Horario

Figura 4.2: Variaciones del formulario lateral

```

const timeInput_init = document.getElementById('range_time_init');
timeInput_init.addEventListener('input', (e) => {
    let hour = e.target.value.split(':')[0]
    e.target.value = `${hour}:00`
})
```

Código 4.16: Jquery para formulario con rango de horarios

Para realizar la consulta basta con oprimir el botón `Buscar`, el cual tiene asociada la función `query_for_points`, cuyo trabajo es levantar el mensaje de espera mediante la librería de javascript sweetalert2. Dentro de este módulo de espera se realiza una llamada asíncrona al Backend mediante la función fetch del módulo Fetch API de Javascript. La llamada usa el método POST de HTTP a la dirección `http://<host>:<port>/traffic/`, incluyendo en el body de la llamada los datos entregados al formulario, esto es realizado en el Código 4.17. La pantalla mostrará lo reflejado en la Figura 4.3 mientras se realiza la consulta.

```
var data = new FormData(document.getElementById("traffic_form"));
fetch("http://127.0.0.1:5000/traffic/", {method: 'POST', body: data})
```

Código 4.17: Fetch API

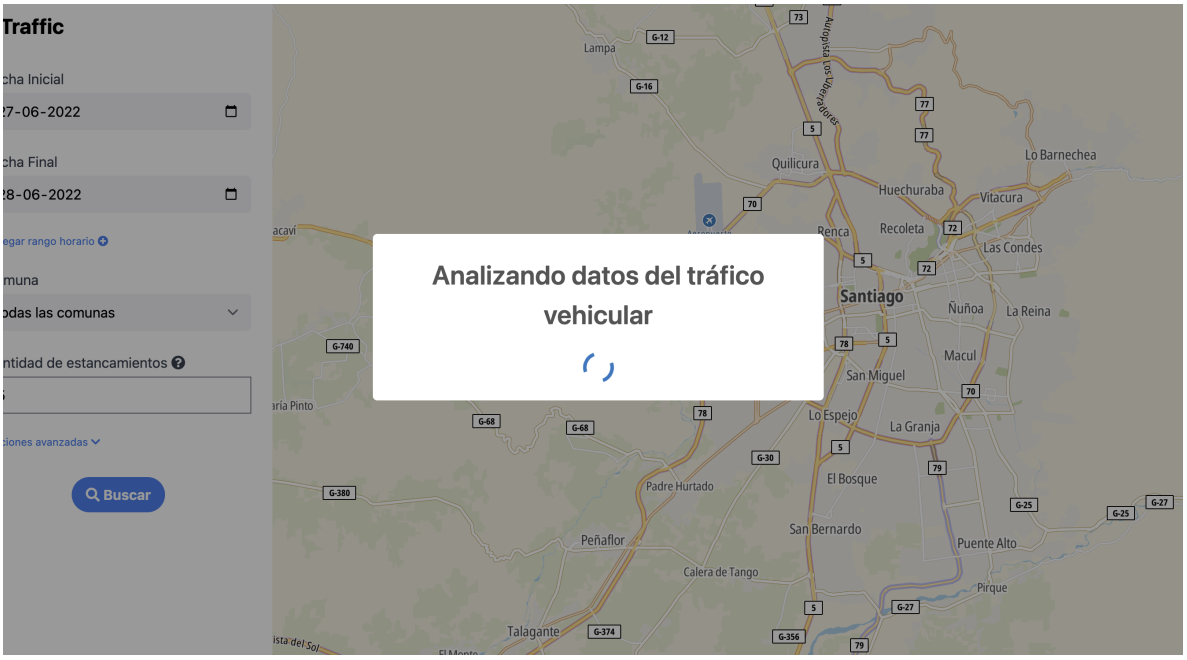


Figura 4.3: Pantalla de carga mientras se realiza la consulta

Esta llamada es acompañada por la función `then`, la cual ejecuta la función que recibe como argumento cuando llega la respuesta desde el backend. Al llegar la respuesta asigna en variables los datos necesarios para agregar las marcas de los tacos al mapa. Luego de tomar los valores de la respuesta del backend ajusta el centro y el zoom del mapa, arma la lista `points_to_query_address` la cual tiene los puntos que representan a cada taco y consulta por sus direcciones a la API Reverse Geocode de TomTom, recibiendo como respuesta los datos de la dirección de cada coordenada (x,y). El SDK de TomTom provee un método para llamar a la API Reverse Geocode, llamada que se realiza de forma asíncrona acompañada de una función `then` que se ejecuta cuando la respuesta por parte de TomTom es recibida. Esta función `then` ejecuta la función que se le entrega como argumento, la cual tiene como objetivo recorrer todas las listas entregadas por el Backend índice a índice, obteniendo la coordenada (x,y) que representa la dirección geográfica del taco, su dirección, su fecha y hora del momento más crítico del taco, su extensión en metros y el tiempo de retraso en minutos. Esta lógica se encuentra en el Código 4.18

```
then(result => {
    result = result.value
    point_list = result.points
    map_config = result.map_config
    points_delay = result.point_incident_delay
    points_length = result.point_incident_length
    points_datetime = result.points_incident_datetime
    map.setZoom(map_config.zoom)
    map.setCenter(map_config.center)
    points_to_query_address = []
    for (point in point_list) {
        points_to_query_address.push({ "position": point_list[point] })
    }
    tt.services.reverseGeocode({
        batchMode: 'sync',
```

```
        key: 'vWTlTqKJ02v7vwz6TojysaLIsY5WVLa8',
        batchItems: points_to_query_address,
    }).then(result => {
        addresses = result.batchItems
        i = 0
        for (point in point_list) {
            point_address = addresses[i].addresses[0].address.
   streetNameAndNumber === undefined ? addresses[i].addresses[0].address.
   streetName : addresses[i].addresses[0].address.streetNameAndNumber
            var fixed_time = new Date(points_datetime[point]);
            marker_array.push(createMarker(point_list[point], '#3b83f6',
   generate_popup_text_info(points_delay[point], points_length[point],
   fixed_time, point_address)));
            i += 1
        }
        marker_array[0].togglePopup()
    });
});
```

Código 4.18: Función que marca los tacos en el mapa

Todos estos valores se entregan a la función `create_marker`, cuya implementación podemos ver en 4.19, la cual crea un elemento HTML al insertar en el mapa mediante las funciones de javascript que interactúan con el DOM. Luego el elemento es insertado en el mapa usando las clases `Marker` y `Popup` incluidas en el SDK de TomTom. El resultado de esto se puede ver en la Figura 4.4.

Figura 4.4: Marcador en mapa

```
function createMarker(position, color, popupText) {
    var markerElement = document.createElement('div');
    markerElement.className = 'marker';

    var markerContentElement = document.createElement('div');
    markerContentElement.className = 'marker-content';
    markerContentElement.style.backgroundColor = color;
    markerElement.appendChild(markerContentElement);

    var iconElement = document.createElement('div');
```

```
    iconElement.className = 'marker-icon';
    iconElement.style.backgroundImage =
        'url("../assets/images/custom-markers/jammm.png")';
    iconElement.style.backgroundSize = 'cover';
    markerContentElement.appendChild(iconElement);

    var popup = new tt.Popup({ offset: 30 }).setHTML('<div class="tt-pop-
    up-container">' +
        '<div class="pop-up-content">' + popupText + '</div>' + '</div>');
    // add marker to map
    marker = new tt.Marker({ element: markerElement, anchor: 'bottom' })
        .setLngLat(position)
        .setPopup(popup)
        .addTo(map);

    return marker
}
```

Código 4.19: Implementación de createMarker

4.3. Alimentación de Datos

Se cuenta con una cuenta en Amazon Web Services (AWS), el cual es un proveedor de servicios en la nube, que permite disponer de almacenamiento, recursos de computación, aplicaciones móviles, bases de datos, entre otros servicios. En esta cuenta se usa el servicio EC2 que permite tener servidores virtuales, donde hay una instancia que funciona con Ubuntu Server. La instancia del servidor es del tipo T2.micro, la cual cuenta 1GB de RAM y un procesador Intel Xeon Scalable de hasta 3,3 GHz de un núcleo. Dentro de esta instancia se encuentra levantada una base de datos MongoDB junto al script de inyección, el cual fue escrito en Python y es activado a través de Cron (administrador de tareas de Linux) cada 10 minutos para consultar y almacenar nuevos datos del tráfico.

La alimentación de la base de datos se realiza mediante un script de python llamado `data_inyector.py` el cual necesita estar acompañado de un archivo llamado comunasChile.geojson que contiene todos los perímetros de las comunas de Chile en forma de polígonos, lo cual permite consultar a qué comuna pertenece una coordenada (x,y). El script `data_inyector.py` está conformado por 3 clases `TomTomIncidentRequest`, `ConnectMongo` y `DataInyector`.

La clase `ConnectMongo` solo tiene un método estático que establece una conexión con la base de datos. Por otro lado la clase `TomTomIncidentRequest` tiene los siguientes métodos:

1. `call_api`: Realiza la llamada a la Incidents API de TomTom y retorna su respuesta.
2. `get_comuna`: Consulta la comuna a la que pertenece una coodernada geográfica (x,y).
3. `complete_comuna`: Agrega a cada incidente la comuna en la que se ubica.

La implementación de esta clase se encuentra en el Código 4.20.

```python
class TomTomIncidentRequest:
    def __init__(self, api_key):
        self.city_coordenates = "
    -70.805086,-33.538223,-70.525680,-33.364616"
        self.api_key = api_key
        self.url = "https://api.tomtom.com/traffic/services/5/
    incidentDetails"
        with open("/home/ubuntu/cron/comunasChile.geojson") as f:
            self.geojson = json.load(f)

    def call_api(self):
        payload = {
            "bbox": self.city_coordenates,
            "language": "es-ES",
            "fields": "{incidents{type,geometry{...},properties{...}}}",
            "key": self.api_key,
        }
        req = requests.get(self.url, params=payload)
        return req

    def get_comuna(self, raw_point):
        point = Point(raw_point)
        for feature in self.geojson["features"]:
            polygon = shape(feature["geometry"])
            if polygon.contains(point):
                return feature["properties"]["COMUNA"]

    def complete_comunas(self, incidents):
        for incident in incidents:
            incident["properties"]["comuna"] = self.get_comuna(
                incident["geometry"]["coordinates"][0]
            )
        return incidents
```

Código 4.20: TomTom Incident Request

Finalmente, la clase `DataInjector`, cuya implementación se encuentra en el Código 4.21 ocupa las dos clases anteriormente descritas para insertar nuevos datos en la instancia de MongoDB. En su constructor instancia un objeto de la clase `TomTomIncidentRequest` y un objeto de la clase `ConnectMongo`, instancias utilizadas por su único método llamado `inject`. Este método se encarga de consultar la Incidents API de TomTom, recorrer la lista de accidentes que responde la API y agregar a cada uno la comuna donde ocurren para finalmente agregar una nueva entrada a la base de datos que contiene la fecha y hora de la consulta junto con la lista de accidentes.

```python
class DataInjector:
    def __init__(self, api_key):
        self.db_connection = ConnectMongo.get_connection()
        self.api_connection = TomTomIncidentRequest(api_key)

    def inject(self):
        db = self.db_connection.traffic_db
        traffic = db.traffic
        raw_incidents = self.api_connection.call_api()
        incidents = self.api_connection.complete_comunas(
            raw_incidents.json().get("incidents")
```

```
        )
        new_traffic_document = {
            "date": datetime.datetime.utcnow(),
            "incidents": incidents,
        }
        traffic.insert_one(new_traffic_document)
```

Código 4.21: Data Injector

El método principal del script crea una instancia de `DataInjector` y ejecuta su método `inject`. El script es ejecutado mediante un cron job, el cual ejecuta el `data_injector.py` cada 10 minutos, cuyo ejecución se define en el Código 4.22.

```
*/10 * * * * /home/ubuntu/venv/bin/python /home/ubuntu/cron/data_inyector.
    py
```

Código 4.22: Comando que ejecuta el cronjob

En la base de datos se construyeron 2 índices compuestos (usan más de un campo) para mejorar el proceso de consulta, estos índices ayudan a realizar la operación `$match` de forma más rápida. El índice by_dates usa los campos incidents.properties.iconCategory y date para mejorar la velocidad de consulta cuando se pregunta por los tacos en un rango de tiempo en toda la ciudad de Santiago. El segundo índice es by_dates_and_commune, el cual usa los campos incidents.properties.iconCategory, incidents.properties.comuna y dates para mejorar las consultas en un rango de tiempo sobre una comuna. Al estar el operador `$match` al principio del pipeline de la consulta estos índices se ejecutan automáticamente.

4.4. Resumen

En este capítulo se describió la implementación del Backend, siguiendo una estructura de software de capas, lo cual permite que el software sea más extensible para futuros desarrollos. Se describe también el flujo que sigue el sistema desde que llega una consulta hasta entregar su respuesta. En la sección de Frontend se mostró la interfaz con que el usuario interactúa con VisTraffic y como los tacos son marcados en el mapa. Finalmente en la sección de Alimentación de Datos se mostró como se capturan y guardan los datos del tráfico vehicular.

Capítulo 5

Validación

5.1. Metodología

El objetivo de la validación será evaluar la funcionalidad y la usabilidad de la aplicación. Para ello se ha confeccionado una encuesta en Google Forms con 4 actividades, donde las actividades 1, 2 y 3 tienen foco en la evaluación de la funcionalidad de la plataforma, mientras que la actividad 4 tiene el foco en evaluar la usabilidad de la aplicación.

5.1.1. Funcionalidad

Para realizar la evaluación de la funcionalidad se confeccionaron 3 actividades, en las cuales se busca que el usuario de la aplicación explore todos los distintos tipos de consultas. En la evaluación no se interactúa con las opciones avanzadas (Retraso Mínimo y Largo Mínimo) ya que para un usuario común no agrega valor a lo que puede consultar. Cabe destacar que para usuarios dedicados al tema, es decir, que trabajan en algo relacionado al transporte y toman decisiones al respecto, las opciones avanzadas podrían ser de ayuda para construir consultas que entreguen información enfocada en este tipo de usuario.

Actividad 1

La actividad 1 tiene las siguientes instrucciones: En esta actividad tu objetivo es identificar los 3 peores tacos en la comuna de Santiago durante el mes de Junio.

El objetivo de esta actividad es que el usuario sea capaz de consultar los datos en un rango de tiempo amplio (todo un mes), filtrar la consulta a sólo una comuna y seleccionar cuantos estancamientos (o tacos) quiere marcar en pantalla.

Luego de realizada la actividad el usuario evaluado debe responder dos preguntas:

1. ¿Cuál es el lugar del taco más extenso?

2. ¿Cuál es el lugar del taco que presenta mayor retraso?

Estas preguntas se realizan con el fin de comparar la información recibida por cada usuario. Además, se busca que el usuario se relacione con la información entregada por cada taco (Fecha, Lugar, Retraso, Extensión), en este caso se busca que el usuario haga una relación entre Lugar-Extensión y Lugar-Retraso.

Las tacos marcados al realizar la Actividad 1 se encuentran en la Tabla 5.1.

Tabla 5.1: Tacos de la Actividad 1

	Fecha	**Lugar**	**Retraso**	**Extensión**
Taco 1	18-06-2022, 12:20:02	Calle Franklin 1034	33 minutos	1314 metros
Taco 2	01-06-2022, 12:50:02	Calle Unión Latino Americana 191	27 minutos	1538 metros
Taco 3	09-06-2022, 12:30:02	Avenida Santa Rosa 25	14 minutos	1308 metros

Actividad 2

La actividad 2 tiene las siguientes instrucciones: En esta actividad tu objetivo es identificar el peor taco del día 07 de Junio en la ciudad de Santiago.

El objetivo de esta actividad es que el usuario vea que puede consultar por un día y que la consulta se realice sobre toda la ciudad, no solo sobre una comuna.

Luego de realizada la actividad el usuario evaluado debe responder dos preguntas:

1. ¿En qué lugar se ubica?
2. ¿A qué hora sucedió?

En esta actividad el usuario debe enfocarse en la Fecha para responder las preguntas.

Las tacos marcados al realizar la Actividad 2 se encuentran en la Tabla 5.2.

Tabla 5.2: Tacos de la Actividad 2

	Fecha	**Lugar**	**Retraso**	**Extensión**
Taco 1	07-06-2022, 07:40:02	Autopista Central 1781	17 minutos	16062 metros
Taco 2	07-06-2022, 18:40:03	Avenida Héroes de La Concepción	4 minutos	1011 metros

Actividad 3

La actividad 3 tiene las siguientes instrucciones: En esta actividad tu objetivo es identificar los 5 peores tacos entre el 13 y el 17 de junio en la ciudad de Santiago durante el horario punta de la tarde (18:00 - 20:00)

El objetivo de esta actividad es que el usuario agregue un filtro de rango horario a su consulta y explore un nuevo rango de fechas en su consulta, realizando una consulta semanal.

Luego de realizada la actividad el usuario evaluado debe responder dos preguntas:

1. ¿Reconoces alguno de los tacos marcados en el mapa?
2. Si tu respuesta fue no, ¿Por qué no los reconoces?

Como ya se había evaluado todos los campos de datos de un taco en esta actividad solo se pregunta si reconoce uno de los tacos marcados en el mapa. Como estamos filtrando por horario punta se espera que sean estancamientos populares de la ciudad.

5.1.2. Usabilidad

Para realizar la evaluación de la usabilidad de la aplicación se decidió utilizar el Sistema de Escala de Usabilidad, creado por John Brooke en 1986.

Actividad 4

La actividad 4 contiene las preguntas del Sistema de Escala de Usabilidad adaptadas a la aplicación, quedando de la siguiente forma:

1. Si necesitaras de su información, ¿usarías VisTraffic frecuentemente?
2. VisTraffic es difícil de utilizar
3. VisTraffic es fácil de utilizar
4. Creo que se necesita ayuda técnica para utilizar VisTraffic
5. Creo que las funcionalidades VisTraffic están bien integradas
6. Creo que hay muchas inconsistencias en VisTraffic
7. Creo que las personas pueden aprender a usar VisTraffic rápidamente
8. Creo que VisTraffic es incómodo de utilizar
9. Me sentí cómodo utilizando VisTraffic
10. Debo aprender muchas cosas antes de poder utilizar VisTraffic

Cada una de estas preguntas debe ser contestada con un puntaje de 1 a 5, donde 1 es *Muy en desacuerdo* y 5 es *Muy de acuerdo*. Esto nos permite saber que tan buena o mala es la experiencia del usuario ocupando la aplicación.

5.2. Resultados

5.2.1. Usuarios

Los participantes de la evaluación de VisTraffic son todos residentes de la ciudad de Santiago, sus edades están en el rango de 25-50 años, todos se desplazan por la ciudad durante los días de la semana ya sea por estudios o trabajos. Fueron elegidos por conveniencia, siendo conocidos o amigos de quien realiza este trabajo. De los participantes 2 se movilizan en transporte público, 2 en automóvil y uno en motocicleta. Los participantes fueron reclutados personalmente, cumpliendo que ninguna de estas personas había tenido contacto con la aplicación ni en su diseño ni implementación. La pruebas se realizaron de manera presencial, ejecutando el Backend y Frontend en un computador MacBook Air M1 2020, las cuales fueron realizadas entre los días 10 y 12 de julio del 2022.

5.2.2. Funcionalidad

En cuanto a la funcionalidad, la aplicación se comportó correctamente, entregando siempre información consistente a las consultas de los usuarios. La evaluación fue aplicada a 5 personas.

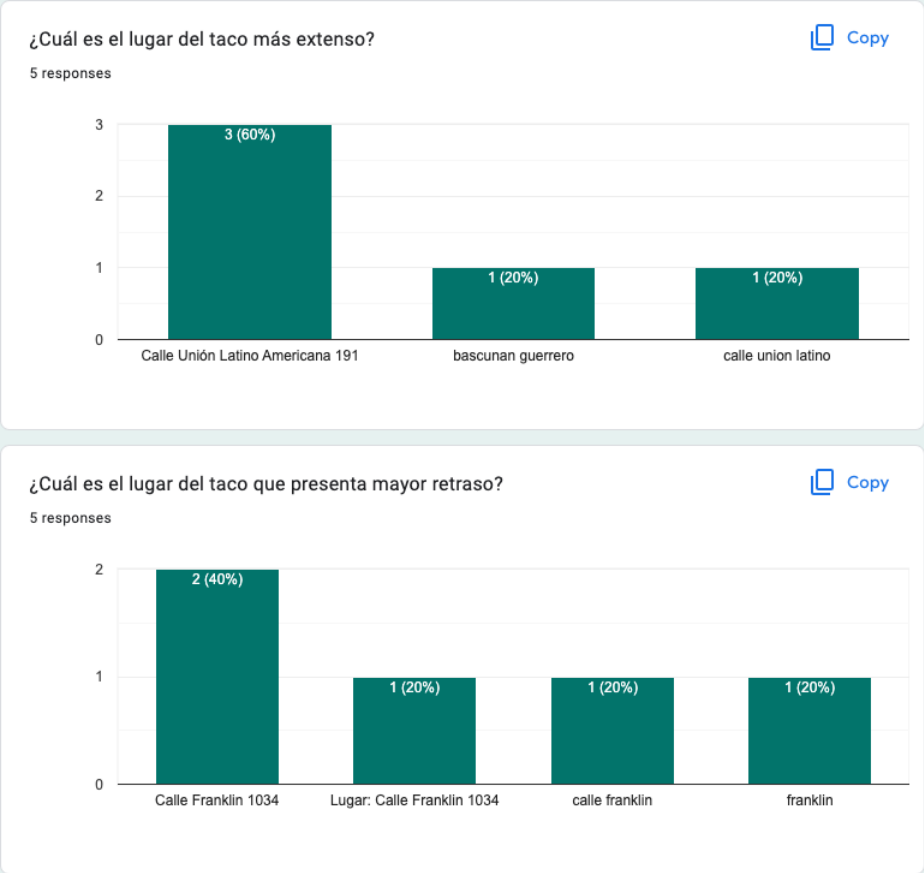


Figura 5.1: Resultados Actividad 1

Actividad 1

Los resultados de la Actividad 1 se muestran en la Figura 5.1.

Podemos observar que en la pregunta ¿Cuál es el lugar del taco más extenso? solo una respuesta es diferente y se debe a que el usuario buscó en el mapa la calle más cercana a la marca en el mapa. Lo bueno es que en la segunda pregunta notó que la ubicación (lugar) estaba incluida en la información del taco.

En cuanto a la pregunta ¿Cuál es el lugar del taco que presenta mayor retraso? se tienen 4 respuestas distintas, sin embargo podemos notar que son la misma respuesta ya que lo que cambia es que algunos copiaron la ubicación del pop-up del taco mientras que otros usuarios escribieron su respuesta.

En resumen, 4 de los 5 encuestados respondieron correctamente la Actividad 1, mientras que uno de ellos solo contestó bien la segunda pregunta de la actividad.

Actividad 2

Los resultados de la Actividad 2 se muestran en la Figura 5.2.

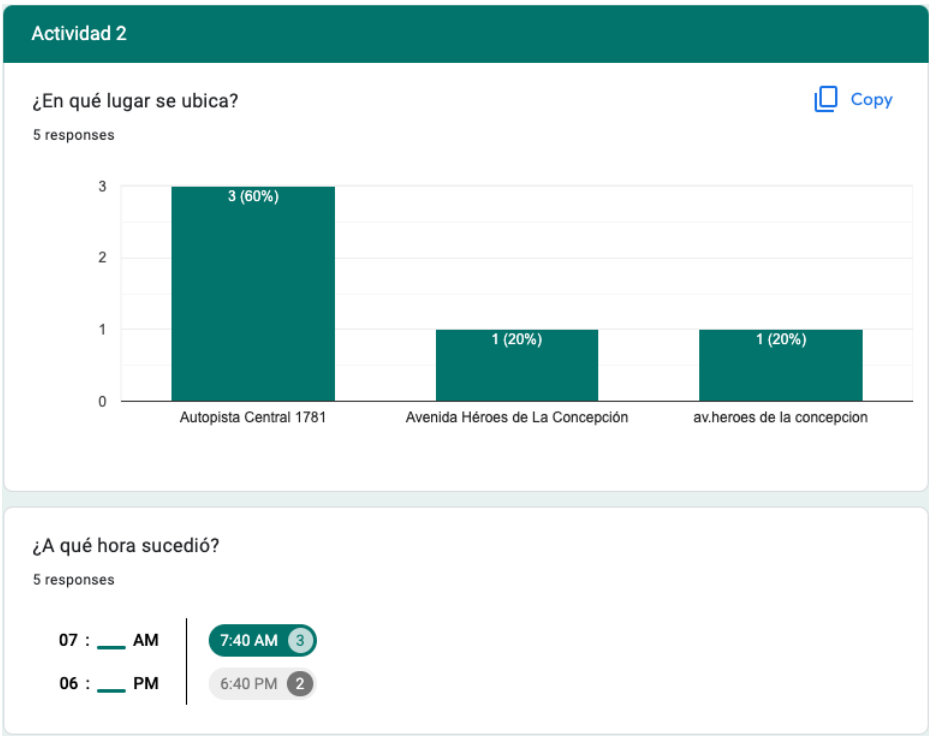


Figura 5.2: Resultados Actividad 2

Los resultados de la consulta de la actividad pueden ser cualquiera de los siguientes:

Esta consulta tiene 2 posibles respuestas, ya que el algoritmo de agrupamiento ordena los tacos según su densidad y existen 2 tacos equivalentes. Como los usuarios marcaban en el

formulario que querían marcar un solo taco (por las instrucciones de la actividad) podía salir cualquiera de las dos opciones válidas, el taco de la Autopista Central o el taco de Héroes de la Concepción. Si se realiza la misma consulta pero marcando 2 tacos en el mapa estos dos son los resultados entregados por la aplicación, por lo que ambas respuestas son correctas.

Todos los usuarios respondieron una de las dos respuestas posibles, por lo que todos realizaron correctamente la Actividad 2.

Actividad 3

Los resultados de la Actividad 3 se muestran en la Figura 5.3.

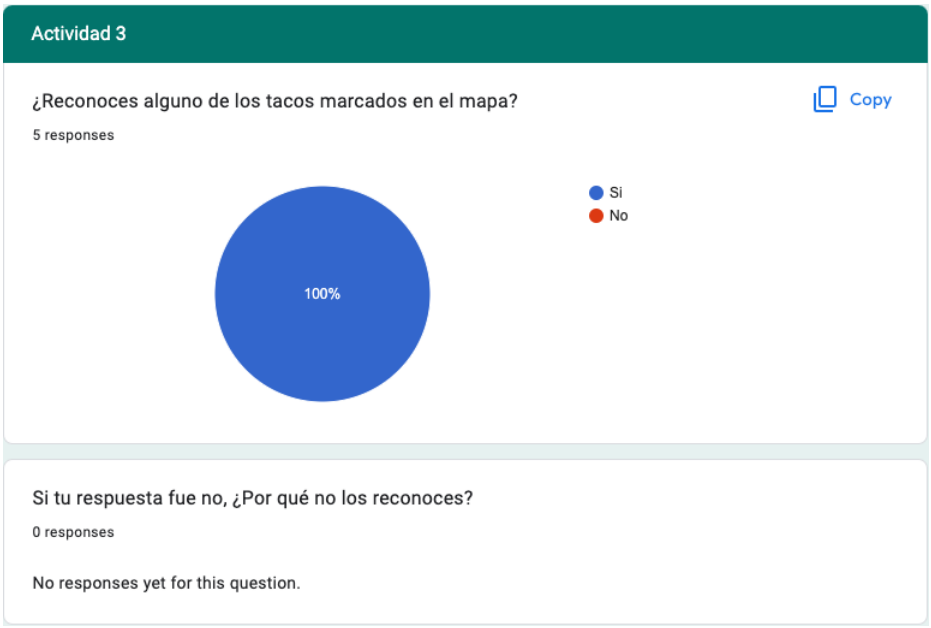


Figura 5.3: Resultados Actividad 3

Con esta actividad se pudo comprobar que los usuarios ya estaban relacionándose bien con el sistema, ya que se realiza una consulta similar a las anteriores pero agregando un rango de horas. Como la consulta se realiza solo en el horario punta se marcan sectores de la ciudad conocidos por presentar estancamientos a esta hora por lo que todos los usuarios reconocieron al menos uno de los tacos marcados.

5.2.3. Usabilidad

Para medir la usabilidad hay que realizar un cálculo con los puntajes entregados en las preguntas de SUS. El cálculo se realiza sumando todos los puntajes de las preguntas impares a lo que llamaremos puntaje_impar, con esto calcularemos el valor de X, donde:

$$X = \text{puntaje\_impar} - 5$$

Por otro lado calculamos la suma de puntaje de las preguntas pares, lo que llamaremos puntaje_par, que utilizaremos para calcular el valor de Y, dónde:

$$Y = 25 - \text{puntaje\_par}$$

Finalmente, el puntaje de usabilidad a partir de la evaluación de una persona se calcula como:

$$Puntaje = (X + Y) \times 2{,}5$$

En la Tabla 5.3 podemos observar una tabla con los resultados donde cada columna representa una pregunta y cada fila es una persona evaluando la aplicación. Junto a esto se encuentra la Tabla 5.4 en la que están calculados los puntajes de cada persona siguiendo la regla descrita anteriormente, acompañada finalmente por el promedio de estos valores.

Tabla 5.3: Respuestas del cuestionario de usabilidad

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**
Persona 1	5	1	5	1	5	1	5	1	5	1
Persona 2	3	1	5	1	5	1	5	1	5	1
Persona 3	5	2	4	3	5	1	5	2	5	2
Persona 4	5	1	5	1	5	1	4	1	5	1
Persona 5	4	3	3	1	5	1	5	1	4	2

Tabla 5.4: Puntaje por persona

	Puntaje
Persona 1	100
Persona 2	95
Persona 3	85
Persona 4	97.5
Persona 5	82.5
Promedio	92

Este puntaje se puede interpretar usando el diagrama de la Figura 5.4.

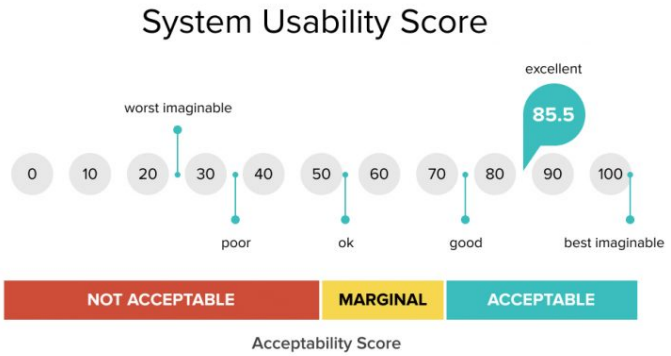


Figura 5.4: Puntaje de Usabilidad para Sistemas [5]

Como se observa del diagrama, desde 80 puntos es una usabilidad aceptable. El promedio que se obtuvo de la evaluación de VisTraffic es de 92, por lo que la usabilidad de la aplicación es excelente.

5.2.4. Comentarios

Luego de terminar las 4 actividades el usuario podía escribir comentarios, sugerencias o dudas de VisTraffic. Se destacan los siguientes comentarios:

1. Puede existir una confusión entre la comuna y la ciudad de Santiago, por lo que sería mejor denominarlo Región Metropolitana.
2. El cuadro de cantidad de estancamientos no es tan explicativo en su función.
3. Mejorar la imagen y el diseño del cuadro de información del taco (hacerlo más atractivo para la vista).

La mayoría de los comentarios están relacionados con mejoras en la interfaz que se comunica con el usuario, lo cual es importante a tener en cuenta ya que puede ayudar a mejorar el puntaje de usabilidad de la aplicación.

Capítulo 6

Conclusiones

La aplicación VisTraffic permite consultar los datos de los tacos de la ciudad y recibir una representación geográfica de donde se encuentran, ordenados por una métrica que definimos en base a la frecuencia con que un taco aparece en cada registro de la base de datos. Si bien esta métrica podría no ser óptima, las pruebas sobre la aplicación muestran que al menos los usuarios reconocen los tacos marcados en el mapa, por lo que sí se están identificando los tacos más relevantes de la ciudad.

Un detalle importante a observar es que el tiempo de retraso de los tacos es un tiempo calculado por TomTom, por lo que este tiempo puede tener imprecisiones. Por otro lado, pero aún hablando del tiempo de retraso, podría parecer extraño o erróneo que entre los peores tacos haya alguno con un tiempo de retraso pequeño, como por ejemplo 4 minutos, lo cual puede ser correcto al ser un taco pequeño que se mantuvo por horas. Si un taco pequeño se mantiene por horas ese taco tiene una alta frecuencia para VisTraffic, ya que aparece en varios registros continuos, por lo que estaría dentro de los peores tacos.

Al día de la entrega de esta memoria la base de datos tiene alrededor de 36 mil registros de Santiago, desde diciembre hasta la fecha, pesando aproximadamente 4GB. Si bien VisTraffic necesita varias mejoras para llevarse a producción y ser una herramienta de uso masiva, la base de datos ya significa un aporte importante para la ciudadanía, ya que personas relacionadas al mundo del transporte podrían realizar consultas a estos datos, diferentes a las que realiza VisTraffic, que entreguen información relevante para analizar y mejorar el flujo vehicular de la capital.

El funcionamiento de la aplicación depende de dos servicios de TomTom, uno es el servicio Maps Display API, el cual es el encargado de mostrar el mapa y lo que uno agrega en él. El otro servicio es Search API, que se utiliza para consultar las direcciones de los puntos que representan a un taco. Por otro lado, la base de datos se alimenta de Traffic API, por lo que en caso de que la API presente interrupciones en su servicio, la base de datos no podría ingresar información de esos momentos.

Existen tacos en la ciudad que pueden empezar en una comuna y terminar en otra, este problema se enfrentó dos veces en el desarrollo de la aplicación, la primera vez fue al momento de agregar un nuevo registro a la base de datos. El campo `comuna` es agregado utilizando

un archivo geojson que contiene los polígonos de las comunas, se toma el primer punto de la lista de puntos que representan al taco y se consulta a qué polígono pertenece el punto, por lo que un taco que empieza en una comuna y termina en otra se reconoce su ubicación por la comuna donde comienza. La segunda vez fue al momento de calcular donde irá el marcador del taco en el mapa, para este caso se calculó el centroide del grupo calculado por el algoritmo de agrupación, por lo que en algunos casos un taco podría estar marcado afuera de la comuna a la que pertenece.

En cuanto a las tecnologías seleccionadas en el diseño de VisTraffic todas funcionaron de acuerdo a lo esperado, la librería del algoritmo de agrupamiento contaba con buena documentación, flask-injector permitió de forma sencilla agregar al inyección de dependencias en el Backend y pymongo permitió consultar la base de datos de forma muy similar a como se harían consultas directas a la base de datos, ocupando `python` en vez de su lenguaje de consultas.

Otro software que sirvió mucho para el desarrollo del trabajo fue MongoDB Compass, que es una herramienta interactiva para consultar, optimizar y analizar los datos de una base de datos MongoDB. En ella se consultaron los datos, se construyeron y probaron los `pipelines` de las consultas `aggregate` y se construyeron los índices que optimizan el proceso `$match` de estas consultas.

En cuanto a los objetivos de la memoria podemos decir que el objetivo principal se cumplió parcialmente, ya que se logró construir una aplicación que entrega visualizaciones de los datos históricos del tráfico en Santiago, pero no se pudo determinar si estos datos son relevantes para la toma de decisiones viales. Los objetivos específicos sí se cumplieron, ya que se muestran los peores tacos en el mapa analizando el rango de tiempo consultado, los datos están constantemente siendo almacenados y la prueba de usabilidad avala que la aplicación es intuitiva en su uso.

6.1. Trabajo Futuro

Hay varios trabajos futuros para VisTraffic en pos de que sea una herramienta de uso masivo. Lo primero que se debe mencionar es que actualmente VisTraffic tiene tiempos de consulta muy largos, la consulta de los tacos del mes de junio en la comuna de Santiago toma entre 2 y 3 minutos, desde que se da click al botón buscar hasta que se marcan los tacos en el mapa. Este tiempo se divide en el proceso de consultar la base de datos y en aplicar el algoritmo de agrupamiento sobre la respuesta de la base de datos. Por lo tanto, para mejorar los tiempos de consulta hay que trabajar en optimizaciones en la consulta a la base de datos, como también optimizar el algoritmo de agrupamiento. Para la base de datos se podría analizar el crear más tablas con los datos, por ejemplo, separando cada mes en una tabla y ver cómo cambian los tiempos de consulta. En cuanto al algoritmo de agrupamiento al tener una complejidad temporal de $O(n^2)$, donde n es la cantidad de puntos, se podría realizar una reducción en la representación de un taco, pasando de una lista de puntos a solo un punto que represente al taco, el cual podría ser el centroide de la lista. Por otra parte también se podría implementar el algoritmo de agrupamiento en forma paralela, con tal de que disminuya su tiempo de ejecución.

Otra mejora a la identificación de los tacos es ocupar un algoritmo de agrupación con sesgo, ya que al tener una base de datos con miles de registros se pueden realizar pre-cálculos sobre los datos que permitan al algoritmo simplificar el como reconoce un taco y que el proceso tome menos tiempo de ejecución.

MongoDB también incluye un tipo de consultas geoespaciales, las cuales son `geoIntersect`, que selecciona las geometrías que intersectan con una geometría entregada a la consulta; `geoWithin`, que selecciona las geometrías que están dentro la geometría entregada a la consulta y `near` que selecciona las geometrías cercanas a un punto. Puede ser interesante y se deja como trabajo futuro realizar las consultas de VisTraffic utilizando esta característica de MongoDB para comparar los tiempos de consulta con el desarrollo actual de la aplicación.

El servidor donde se encuentra la base de datos es bastante lento, ya que es el servidor AWS EC2 T2.micro, mientras que el Backend es ejecutado en un MacBook Air M1 2020. Sería interesante como trabajo futuro probar que la los tiempos que toma la aplicación en marcar los tacos si el sistema estuviera montado en equipos de mejores características.

Es interesante realizar pruebas esta aplicación a más usuarios, sobre todo a usuarios relacionados al tema de transporte, los cuales pueden ser trabajadores del Ministerio de Transporte y Telecomunicaciones, trabajadores del UOCT, trabajadores municipales de la dirección de tránsito, entre otros. Esto puede ser de gran ayuda, tanto para planificar nuevas consultas como para mejorar la experiencia de usuario y usabilidad de la plataforma.

Bibliografía

[1] Comparing python clustering algorithms. https://hdbscan.readthedocs.io/en/latest/comparing_clustering_algorithms.html, 2022.

[2] Cron. https://www.man7.org/linux/man-pages/man5/crontab.5.html, 2022.

[3] Flask, a flexible and popular web development framework. https://palletsprojects.com/p/flask/, 2022.

[4] Pymongo. https://pymongo.readthedocs.io/en/stable/, 2022.

[5] 10up. System usability score. https://10up.com/uploads/2018/11/sus-score-1-768x427.jpg, 2018.

[6] Asociación Nacional Automotriz Chile (ANAC). Informe mercado automotor anac. *Estudios de Mercado ANAC*, Septiembre, 2021.

[7] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] CONASET. Evolución de siniestros de tránsito chile (1972 – 2021). https://www.conaset.cl/programa/observatorio-datos-estadistica/biblioteca-observatorio/estadisticas-generales/, 2021.

[9] Instituto Sistemas Complejos de Ingeniería. Parque de autos se acelera y rozará los 5,5 millones de unidades este año. 2021.

[10] Google. Google maps platform. https://developers.google.com/maps, 2021.

[11] Marco Gutierrez. Parque de autos se acelera y rozará los 5,5 millones de unidades este año. *El Mercurio*, 2021.

[12] Here. Here maps for developers. https://developer.here.com/?cid=www.here.com-topnav-menu, 2021.

[13] TomTom Traffic Index. Santiago traffic report. https://www.tomtom.com/en_gb/traffic-index/santiago-traffic/, 2021.

[14] pandas dev. pandas: powerful python data analysis toolkit. https://pandas.pydata.org/, 2022.

[15] pandas dev. pandas: powerful python data analysis toolkit. https://github.com/pandas-dev/pandas, 2022.

[16] Tamara Rojas. Prevén que congestión vehicular seguirá subiendo en rm: velocidad llega a 22 km/h en puntos críticos. *Radio Bío-Bío*, 2021.

[17] TomTom. Tomtom for developers. https://developer.tomtom.com/, 2021.

[18] TomTom. Tomtom market coverage. https://developer.tomtom.com/search-api/search-api-documentation/search-api-market-coverage, 2021.

[19] TomTom. Tomtom traffic api. https://developer.tomtom.com/traffic-api/traffic-api-documentation-traffic-incidents/incident-details, 2021.