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

PLANIFICADOR DE COMPRAS COLABORATIVAS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

GIANLUCA CESARE CARBONI MENÉNDEZ

PROFESOR GUÍA:
FEDERICO OLMEDO BERÓN

MIEMBROS DE LA COMISIÓN:
JOSÉ PINO URTUBIA
ANDRÉS MUÑOZ ORDENES

SANTIAGO DE CHILE
2021

Resumen

Desde los comienzos del año 2020, la propagación del virus SARS-CoV-2 (causante de la enfermedad COVID-19) ha afectado la situación cotidiana de toda la población. Este virus puede causar desde síntomas leves hasta enfermedades graves que pueden causar incluso la muerte de personas infectadas. Es por esto que para reducir las tasas de contagio, se recomienda a la población no salir de manera tan frecuente de su hogar. Sin embargo, existen actividades esenciales como salir a realizar compras del hogar que no pueden ser reemplazadas fácilmente. En Chile, si bien las personas pueden realizar esta actividad incluso en tiempos de cuarentena a través de la emisión de permisos de movilidad especiales, esta impone inevitablemente un riesgo para la persona debido a la posibilidad de contagio.

Surge así la necesidad de reducir el número de salidas que las personas deben realizar para comprar productos de primera necesidad. Si bien existen alternativas para no tener que salir de casa, como lo son los servicios de delivery pagos, estas conllevan ciertas restricciones como por ejemplo el costo de envío o sobrecarga de los precios respecto a la tienda física. Otro enfoque que se puede seguir para abordar este problema es generar mecanismos que fomenten la colaboración entre los integrantes de una comunidad que vivan en conjunto como los vecindarios.

Es así como el trabajo de esta memoria consiste en el desarrollo e implementación de un prototipo de una *app* móvil que ayude a la coordinación de compras colaborativas entre los distintos miembros de una comunidad. Adicionalmente, para los casos donde el miembro encargado de realizar las compras no sea capaz de realizar la totalidad de las compras debido a sus restricciones (monetarias, peso total, entre otras), la *app* se encarga de seleccionar automáticamente los productos más prioritarios a través de un algoritmo de optimización que busca obtener el mayor beneficio posible para la comunidad, y que además cumpla con las restricciones del comprador.

Para validar la solución desarrollada, realizamos 3 tipos de evaluaciones. Primero evaluaciones de usabilidad, lo que es un aspecto crucial debido a que la solución será utilizada por el público general. Segundo, desarrollamos un modelo matemático para estimar el número de salidas a comprar que un miembro se "ahorraría" al utilizar la solución desarrollada. Finalmente, determinamos qué tan alineada está la selección de productos automática con las selecciones que realizarían los potenciales usuarios. Las 3 evaluaciones arrojaron resultados satisfactorios, permitiendo concluir que el prototipo desarrollado cumple los objetivos planteados. No obstante, estas también entregaron *feedback* sobre algunas potenciales mejoras, que por limitaciones de tiempo, dejamos como trabajo futuro.

Agradecimientos

Quiero agradecer al profesor Olmedo por darme el tema de esta memoria y ayudarme en todo el proceso. También a otros profesores del DCC como Aidan por responder a tantas preguntas que les hice sobre la carrera en sí. A Sandra y Ángelica por ayudarme con varios procesos de esta carrera. Y, finalmente, a mis amigos y familia por su apoyo incondicional a lo largo de todos estos años.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes

El COVID-19 (nombrado por sus siglas en inglés *COronaVirus Disease 2019*) es la enfermedad causada por SARS-CoV-2, el virus descubierto más recientemente de la familia de coronavirus. Esta corresponde a una enfermedad infecciosa que puede causar síntomas leves como rinorrea, cansancio, tos, fiebre, entre otros, y que para algunas personas puede llegar a ser mortal, principalmente para las personas mayores o aquellas que sufran ciertas enfermedades [1]. Actualmente, debido a la propagación de este virus, hemos estado viviendo una pandemia global hace más de un año, donde los diferentes países han tomado distintas medidas para combatir el virus.

A nivel mundial, la OMS ha recomendado al público distintas medidas de protección básica como el usar mascarillas, lavarse las manos frecuentemente, y mantener el distanciamiento social [2]. Muchas de las medidas impuestas por gobiernos nacionales, incluyendo el chileno, involucran las declaraciones de estados de cuarentena. El objetivo de estas es separar y restringir el movimiento de personas que posiblemente estuvieron expuestas al virus, para así poder frenar su propagación.

Debido a estas restricciones, la gente ya no puede realizar actividades regulares como reuniones sociales, las cuales pueden ser reemplazadas por otras alternativas como videollamadas. Sin embargo, la compra de productos de primera necesidad no puede ser reemplazada fácilmente y debe seguir haciéndose aún en tiempos de cuarentena. Para ello, el gobierno de Chile emite permisos temporales individuales que permiten el desplazamiento de las personas hacia los distintos puntos de venta.

Aún así, el uso de estos permisos temporales trae ciertas limitaciones. Por empezar, hay un límite semanal en la cantidad de permisos que se pueden solicitar. Peor aún, estos permisos incluyen todo tipo de compras de primera necesidad, lo que puede causar que se agoten antes de lo previsto. Por ejemplo, si hay un límite de 2 permisos semanales (como se impuso en Chile durante el primer *peak* de la pandemia), y una persona solicitó un permiso el día lunes para ir al supermercado, y otro el día miércoles para ir a comprar una estufa, se queda sin

permisos disponibles hasta la semana siguiente. Si durante ese periodo de tiempo debiera salir a la farmacia por un motivo de salud, simplemente no podría. Esta situación se ve aún más agravada por el hecho que los permisos temporales tienen una duración acotada, generalmente de 2 horas, lo que muchas veces impide hacer más de una compra con el mismo permiso.

Otro problema relevante respecto a las compras de primera necesidad es que inevitablemente imponen un riesgo para la persona que hace la compra. Esto puede ser crítico para personas de edad avanzada o que padezcan de otros factores de riesgo. Por ejemplo, se sabe que para la población no vacunada, el virus es al menos 15 veces más letal para personas mayores de 60 años de edad que para aquellas menores de 30 años de edad [3].

Debido a todo esto, existe una *necesidad urgente de reducir el número de salidas* que la gente debe hacer de su casa, por ejemplo a supermercados o farmacias, para abastecerse de productos de primera necesidad. Sin embargo, las soluciones basadas en servicios de delivery pagos como, por ejemplo, Cornershop, llevan consigo ciertas limitaciones que impiden resolver esta necesidad en su totalidad.

Dentro de estas limitaciones se encuentran el hecho de pagar costos de envío, el tiempo considerable que toma realizar estos pedidos, y además la interacción física que se debe realizar con el repartidor, lo que puede resultar en un contagio del cliente. Para mayor información de estas limitaciones, puede revisar la Sección 2.5.

En su lugar, adoptamos una solución en términos de *economía colaborativa* que permite ahorrar dinero, es más amigable con el medio ambiente y resulta muy eficiente si cuenta con una buena adopción.

Para lograr construir una solución con estas características, esta memoria trata sobre la construcción de un prototipo de un *sistema planificador de compras colaborativas* que sea capaz de ayudar a distintos grupos de personas que vivan en comunidad (familias, vecindarios, entre otros).

Lo que buscamos con esta solución es que los miembros de este grupo puedan ayudarse entre ellos y realizar compras al resto del grupo, para que así estos no tengan que salir de casa. Además, con esta solución no buscamos crear solo un canal de comunicación, sino también que esta puede seleccionar de manera automática un subconjunto de los productos solicitados más prioritarios a partir de las posibles restricciones que puedan tener los compradores (dinero, peso, etc...).

Debido a los diversos factores que pueden darle mayor o menor importancia un producto solicitado por una persona, hacer que esta solución pueda efectivamente seleccionar subconjuntos de manera informada corresponde a uno de los mayores desafíos de esta memoria.

A continuación, detallamos los objetivos generales y específicos de esta memoria.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de esta memoria corresponde al desarrollo de un sistema prototipo basado en el modelo de economía colaborativa que apoye a la coordinación de compras de primera necesidad entre miembros de una comunidad. En concreto, cuando una persona vaya a realizar sus compras, los restantes miembros de la comunidad (de ahora en adelante llamados "usuarios") pueden encargarle sus productos para que ellos no tengan que salir de su casa. Una vez recolectados todos los encargos, la tarea fundamental del sistema será seleccionar de manera informada y razonable qué productos comprar a cada usuario, respetando así las restricciones propias del encargado de hacer la compra, como por ejemplo, peso que puede cargar o monto total que puede cancelar.

1.2.2. Objetivos Específicos

1. *Diseño del algoritmo de selección de productos:* A partir de las restricciones que el comprador pueda presentar (por ejemplo, peso y costo total de los productos), el sistema debe ser capaz de seleccionar automáticamente un subconjunto de productos dentro de todos los encargos realizados por los usuarios. Este subconjunto no sólo debe ser compatible con las restricciones del comprador, sino que debe ser escogido siguiendo otros criterios razonables, por ejemplo, dando preferencia a los productos de mayor necesidad o a los productos de aquellos usuarios que más participan de la *app*.

2. *Desarrollo del sistema:* Debemos desarrollar tanto la interfaz del sistema (*frontend*) como la lógica y el almacenamiento de datos (*backend*). Respecto a la interfaz, como es un sistema que será utilizado por el público en general y se espera que sea particularmente útil para adultos mayores, la interacción con el usuario debe ser lo más intuitiva posible, por lo que el diseño de esta debe estar concentrada en la usabilidad.

3. *Evaluación del sistema:* Para determinar que el uso del sistema sea lo suficientemente intuitivo, debemos realizar estudios al sistema a partir de público objetivo. Como en este caso los usuarios varían tanto en experiencia como en edad, el desarrollo se debe enfocar fuertemente en puntos de usabilidad como eficiencia, que sea fácil de aprender y recordar, entre otros. Adicionalmente, debemos comprobar que la solución efectivamente es capaz de reducir el número de salidas a comprar que realizan los usuarios. Finalmente, necesitamos realizar evaluaciones de rendimiento para comprobar que efectivamente el algoritmo de selección se encuentra en línea con lo que las personas consideran que es "justo" y "razonable".

1.3. Metodología

En primer lugar, desarrollamos detalladamente el modelo matemático del problema descrito, determinando así los factores que afectarán al momento de seleccionar los productos que el comprador le llevará a cada usuario. Realizado esto, identificamos la naturaleza del problema de optimización subyacente para después investigar algoritmos que resuelvan el problema propuesto por el modelo. Los algoritmos que investigamos fueron tanto de solución exacta como aproximada.

Afortunadamente encontramos un algoritmo que soluciona el problema de optimización de manera exacta, en un tiempo despreciable para el tipo de instancias que la *app* va a generar, y además resulta ser más consistente que otras soluciones aproximadas investigadas. Gracias a esto, decidimos por la elección de esta alternativa.

Seguido de esto, proseguimos con el desarrollo e implementación del sistema en sí, el cual en un principio discutimos si debería corresponder a una *app* web o móvil. Debido a la portabilidad y a la facilidad de uso que generalmente conllevan las *apps* móviles, decidimos implementar esta última alternativa.

Lo primero en lo que trabajamos es la interfaz de usuario, donde le presentamos a usuarios candidatos bocetos diseñados para representar a la *app*. Específicamente, representan las distintas ventanas que los usuarios visualizarán dentro de la solución final, y sirve para dar cuenta de los posibles errores de usabilidad generales que podrían haber en el diseño inicial. Una vez terminados, los presentamos a los usuarios potenciales, los cuales ofrecieron *feedback* en su mayoría positivo, por lo que continuamos inmediatamente con el almacenamiento y estructura de datos.

A partir de los bocetos presentados a los usuarios, diseñamos una estructura prototipo de base de datos, la cual sufre de diversos cambios a lo largo del desarrollo de la *app*. Para almacenar estos datos, optamos por alternativas de servicios *cloud* con el objetivo de realizar el levantamiento de la base lo más rápido posible. Después de comprobar que el almacenamiento funcionara correctamente utilizando un conjunto de datos de prueba, proseguimos con el desarrollo de la lógica de la *app*.

La programación de la lógica del sistema fue la etapa de mayor duración debido a la gran toma de decisiones que realizamos a lo largo de este proceso como por ejemplo, cómo el usuario comprador terminará las compras de una tienda, cómo realizará los cobros correspondientes, entre otras. Adicionalmente, trabajamos paralelamente en el *frontend* que corresponde a la elección de colores, tamaño de ventanas, espaciado de mensajes, entre otros elementos correspondientes a lo que el usuario visualiza al momento de usar la *app*.

Finalmente, una vez que terminado el desarrollo, realizamos las evaluaciones finales. Con estos resultados, comprobamos no solo si la solución en la que se trabajó es capaz de resolver el problema planteado, sino que también evaluamos si el sistema es lo suficientemente comprensible para poder ser usado por todo el público general, y si efectivamente el algoritmo de selección realiza selecciones “justas”.

1.4. Descripción general de la solución

El prototipo que desarrollamos le permite a personas integrantes de una comunidad (como por ejemplo, un vecindario) ingresar como usuarios dentro de la *app*, donde pertenecerán a un grupo en común. Una persona (el comprador) antes de salir de compras puede anunciar dentro de la *app* que saldrá a comprar. Este debe especificar a que tienda irá a comprar (Jumbo, Lider, Cruz Verde, entre otras), ya que por ejemplo un usuario no podría pedirle productos farmacéuticos si es que el comprador va al supermercado. Además, el comprador puede incluir ciertas restricciones en su anuncio, como por ejemplo cuánto dinero máximo está dispuesto a prestar para pagar los productos del resto de los usuarios, cuánto es el peso máximo extra que puede llevar y cuántos productos puede llevar además de los suyos.

Una vez que el comprador publica el anuncio de su salida dentro de su grupo, los restantes miembros del grupo pueden revisarlo y saber a qué tiendas se dirigirá. Mientras el anuncio permanezca abierto, pueden seleccionar los productos que deseen encargar, visitando dentro de la *app* el catálogo de productos de las tiendas que el comprador va a visitar.

Cuando el comprador llega a la primera tienda, este debe "cerrar" el anuncio dentro del grupo. Al hacer esto, la *app* selecciona un subconjunto de encargos realizados por el resto de los usuarios, tomando en cuenta las restricciones impuestas por el comprador, el cual se muestra en una lista junto al precio tentativo de estos productos y el total a pagar. Si hubo "encargos" que quedaron afuera, estos pasan a una lista de espera. Así, si al momento de comprar hay algún producto que no se encuentra disponible, el comprador puede usar la *app* para eliminar el producto del subconjunto originalmente seleccionado, y la *app* recalculará el nuevo subconjunto a comprar, sin descartar el resto de los productos que estaban desde antes. Dicho de otra manera, debe determinar el reemplazo de los productos no disponibles.

Finalmente, todos los productos comprados pasan a una lista de cobros que además de mostrar cada producto comprado, calcula también el total de dinero que el comprador le debe cobrar a cada usuario a partir del precio final que pagó por los productos (el cual, en caso de ser necesario, puede actualizarse durante el proceso de compra).

Capítulo 2

Marco Teórico

En este capítulo explicamos los elementos considerados para componer la solución, para la completa comprensión del trabajo realizado. Primero definimos las tecnologías a considerar (como por ejemplo, los lenguajes a utilizar), luego compartimos las arquitecturas de diseño y almacenamiento pensadas, después las nociones básicas de programación lineal junto a su resolución. Finalmente hablamos sobre las soluciones existentes que intentan reducir el número de salidas a comprar productos de primera necesidad.

2.1. Tecnologías consideradas

Para poder desarrollar aplicaciones tanto *web* como móviles, tenemos en consideración 2 lenguajes de programación principales: Python [4] o Java [5]. Ambos lenguajes han sido usados en el pasado por el memorista, y generalmente ambos están considerados dentro de los mejores lenguajes que un ingeniero de software debiera saber [6]. Los 2 incluyen herramientas y *frameworks* [7] que pueden servir para resolver el problema.

Python es generalmente considerado un lenguaje más sencillo de aprender debido a la similitud que tiene su sintaxis con el idioma inglés. Contiene herramientas para desarrollar aplicaciones web (como por ejemplo el *framework* Django [8]) que buscan ser fácil de usar para los usuarios. Para el desarrollo de *apps* móviles, uno de los *frameworks* más populares que se utilizan en la industria corresponde a Kivy [9].

Kivy corresponde a una librería de fuente abierta creada específicamente para el desarrollo sencillo de aplicaciones equipadas con interfaces dedicadas a mejorar la experiencia de usuario. Las aplicaciones desarrolladas tienen la particularidad de que pueden utilizarse no solo en un tipo de sistema en específico, sino que son capaces de ser utilizadas en múltiples plataformas, desde computadores personales con sistemas operativos Windows o Linux, hasta dispositivos portátiles con iOS o Android.

Además de ser una librería flexible al ser capaz de desarrollar *apps* multiplataformas, Kivy fue desarrollado específicamente para soluciones *multitouch*, es decir, software capaz de

reconocer el "tacto" del usuario (como lo son la totalidad de las *apps* móviles hoy en día). Adicionalmente, esta librería tiene la particularidad de usar su propio lenguaje descriptivo diseñado exclusivamente para el diseño de interfaces de usuario llamado Kivy Language [10], similar a cómo funciona los lenguajes HTML y CSS para el diseño y elaboración de páginas web.

El hecho de que puedan desarrollarse *apps* multiplataformas con esta librería, facilita también el *testing* debido a que pueden realizarse pruebas en un sistema operativo de escritorio como Windows, y así estas *apps* deberían funcionar de la misma manera en el resto de los sistemas operativos.

Por otro lado, Java es un lenguaje especializado en el desarrollo *orientado a objetos* [11] que también contiene *frameworks* relacionados al desarrollo de *apps* móviles. De hecho, uno de los entornos de desarrollo (o *IDEs* [12]) más utilizados corresponde a Android Studio [13], el cual además es el entorno oficial para la plataforma Android.

Este entorno de desarrollo además de estar basado en IntelliJ [14] (uno de los mejores entornos de desarrollo de Java) contiene módulos y funciones enfocadas en el desarrollo de *apps* para Android, lo que permite un desarrollo más específico y especializado para las plataformas de este sistema operativo.

Dentro de las *features* que posee este entorno, se pueden destacar dos. La primera es la capacidad de emular dispositivos de Android, lo que facilita y hace que el *testing* de nuestro prototipo en distintas plataformas tenga una mayor consistencia. Asimismo, es posible también emular otros comportamientos más allá del *software* que se desarrolla como por ejemplo la navegación dentro del sistema operativo de un celular, o la capacidad de recibir alarmas y/o notificaciones.

La segunda *feature* destacable es la compatibilidad con otros servicios como Google Cloud Platform [15]. Esto permite una integración sencilla con los servicios de esta plataforma, facilitando así el desarrollo de otros elementos que nuestro prototipo pueda necesitar como por ejemplo el levantamiento de una base de datos.

2.2. Patrones de diseño: Modelo Vista Controlador

Para el desarrollo general de sistemas basados en la experiencia de usuarios, una de las alternativas más comunes es regirse por el Modelo Vista Controlador (MVC) [16]. Este corresponde a un patrón de desarrollo de *software* que busca separar el almacenamiento de datos de la lógica de negocio, o en este caso, la lógica de la *app*.

En concreto, lo que busca hacer este patrón es segmentar el *software* desarrollado en 3 componentes esenciales: el *Modelo* que representa a los datos usados por la aplicación, la *Vista* o interfaz de usuario, y el *Controlador* que actúa como el intermediario de los otros dos, gestionando el flujo de información necesario junto a las transformaciones para adaptar los datos a las necesidades de cada uno.

- *Modelo*: además de corresponder a la información misma, gestiona todos los accesos a dicha información, incluyendo tanto consultas como actualizaciones. Envía a la *Vista* la parte de información que en cada momento se le solicita para que sea mostrada (generalmente esta solicitud la realiza el mismo usuario). Todas las peticiones de datos llegan a esta componente gracias al *Controlador*.
- *Controlador*: Responde y reacciona directamente a las acciones que realiza el usuario, y envía peticiones al *Modelo* cuando se realiza alguna solicitud sobre la información. También puede enviar órdenes a la *Vista* si se solicita algún cambio en la forma que el usuario visualiza la información.
- *Vista*: Presenta al *Modelo* en un formato adecuado para que el usuario sea capaz de interactuar, por lo tanto requiere la información que este envía como salida.

El flujo de este patrón funciona generalmente de la siguiente forma. El usuario interactúa con el *software* a través de un elemento de la *Vista*, como por ejemplo un botón. El *Controlador* recibe la notificación de la acción realizada por el usuario y gestiona el evento a través de un manejador de eventos o *callback*. Luego, el *Controlador* accede al *Modelo* actualizándolo o posiblemente modificándolo (si se agrega edita o elimina un dato). Finalmente, el *Controlador* delega a la *Vista* para que muestre los datos de salida del *Modelo* de manera adecuada. Puede observar cómo funciona este flujo en la siguiente figura:

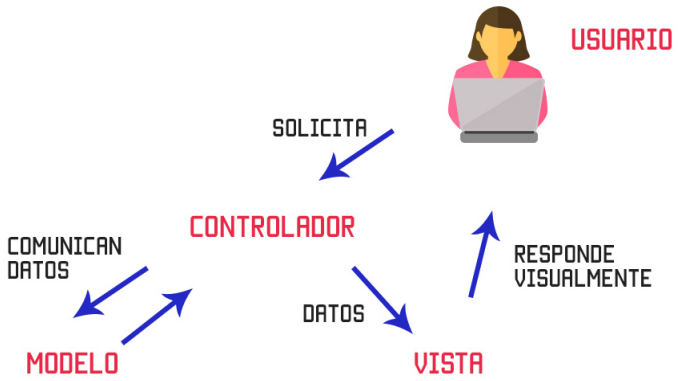


Figura 2.1: Patrón de diseño Modelo Vista Controlador junto a las interacciones entre sus componentes.[1]

2.3. Almacenamiento de datos

Sobre el almacenamiento de los datos, existen varias opciones que se pueden dividir en dos tipos. El primer tipo corresponde a montar un servidor de prueba en un ordenador u otro dispositivo. Esto va a permitir a distintos clientes del sistema conectarse a la *app* a través de la dirección *IP* de esta máquina. A la vez, esta máquina deberá manejar estas conexiones usando algún software dedicado a servidores *HTTP* como Apache [17].

[1]Fuente: `https://codigofacilito.com/articulos/mvc-model-view-controller-explicado`

Apache HTTP Server corresponde a un *software* de servidor gratuito, que le permite a los usuarios servir contenido en la web. En particular, este es uno de los servicios web más populares, ya que este software ejecuta alrededor de un 46 % de los sitios web en el mundo. Esto hace que sea una solución confiable.

Una alternativa más específica sería utilizar Tomcat [18]. Al igual que Apache, también corresponde a un servidor *HTTP*, con la diferencia que esta alternativa se especializa en la comunicación con aplicaciones Java. Esta sería la alternativa ideal si se decidiera usar este lenguaje junto a la *IDE* de Android Studio mencionada en la Sección 2.1.

La otra opción corresponde a la configuración de servicios *cloud*. Estos corresponden a servicios a través de la nube ofrecidos por varias empresas (AWS [19], IBMCloud [20], entre otras), dentro de los que se encuentra el almacenamiento de bases de datos. El costo de este servicio depende de la empresa que ofrece el servicio, el tipo de datos que se quiere almacenar, el tamaño de estos, etc.... Generalmente, ofrecen sus servicios basados en planes que tienen un límite de acciones y/o capacidad de almacenamiento, los cuales mientras a mayor precio, mayor son estas cantidades.

Para el caso de IBMCloud, uno de los servicios más utilizados de esta plataforma corresponde a Db2 [21]. Lo que hace este servicio es otorgar un almacén de datos relacional de alto rendimiento diseñado para empresas. Los usuarios de la plataforma así, pueden almacenar un tamaño variable de datos relacionales los cuales son encriptados por la misma nube.

El costo de este servicio depende de 2 factores. El primero corresponde a la ubicación donde se quiera almacenar los datos, ya que Db2 permite localizar su base en distintas regiones. Es por esto que existen algunas regiones más costosas que otras, pero también existen algunas que no cobran demás. Por otro lado está el tamaño de los datos junto a la cantidad de conexiones simultáneas. Dependiendo del plan que el usuario elija, estos factores aumentan el costo de la mantención. Por ejemplo, los clientes del plan Standard deben pagar 0,1564 dólares por dato cada hora, y además tienen un máximo de 20 GB de almacenamiento dentro de la nube.

Para el tipo de base de datos a utilizar, se estudian 2 opciones principales. La primera corresponde a que esté definida con el lenguaje SQL [22]. La estructura de los datos definidos en este lenguaje está definida en tablas donde cada columna almacena valores singulares. A pesar de que esto hace que la estructura de los datos a utilizar tenga que ser más uniforme y limitada, esto hace que se almacenan de una manera más consistente y además sea más sencillo poder realizar consultas complejas a estos datos. Respecto a la implementación, existen en la actualidad varios motores de base de datos desarrollados en este lenguaje que pueden servir para componer el sistema, como por ejemplo MySQL [23] o PostgreSQL [24].

Por otro lado, también se encuentra la alternativa de almacenar datos en una base NoSQL [25], donde a diferencia de la anterior, esta es capaz de almacenar datos más complejos como por ejemplo imágenes y videos, no solo texto. Adicionalmente, el tamaño de estos datos no tiene por qué encontrarse restringido por el tipo y cantidad de columnas que podría imponer un sistema SQL. Dentro de los motores NoSQL más utilizados en la actualidad se encuentran MongoDB [26] y CouchDB [27].

2.4. Problema de la mochila multi-dimensional

Uno de los objetivos principales del sistema descrito en este informe es que este sea capaz de seleccionar el subconjunto de productos que se deben comprar a partir de las restricciones de los compradores. Esto corresponde a un problema el cual podemos modelar matemáticamente. Después de investigar y estudiar el problema en concreto, se da cuenta de que corresponde a un problema de programación lineal [28].

Los problemas de programación lineal consisten en modelos compuestos de una función objetivo que se busca maximizar o minimizar, sobre un conjunto de variables que se encuentren sujetas a una serie de restricciones en la forma de un sistema de ecuaciones o inecuaciones. Tanto la función objetivo como las restricciones corresponden a ecuaciones lineales.

La Figura 2.2 muestra un ejemplo de estos problemas, donde se busca maximizar una función compuesta de variables x_1 y x_2, las cuales a la vez, se encuentran restringidas por un total de 3 restricciones.

$$\text{Maximizar } 3x_1 + x_2$$

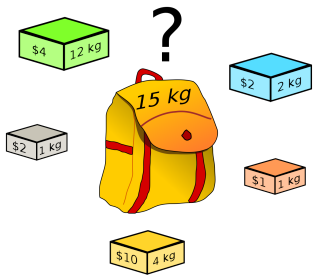
$$s.a \begin{cases} 9x_1 + x_2 \leq 18 \\ 3x_1 + x_2 \leq 12 \\ x_1, x_2 \geq 0 \end{cases}$$

Figura 2.2: Ejemplo de modelo de programación lineal.

Estos modelos se utilizan en casi todas las áreas de toma de decisiones, como por ejemplo la ingeniería de diseño. Estos modelos además son *deterministas*, es decir, que todos los parámetros utilizados se suponen conocidos con certeza, evitando así tener que pensar en probabilidades. Además, como estas funciones lineales no son ni estrictamente convexas ni estrictamente cóncavas, sus soluciones no son necesariamente únicas.

Un ejemplo de programación lineal que codifica el problema de la selección automática de productos es el *problema de la mochila* [29]. En este, se intenta encontrar el subconjunto de mayor valor dentro de un conjunto de objetos de variado valor y peso, donde además este subconjunto no puede sobrepasar el peso máximo que puede almacenar la mochila, definiendo así la restricción que debe cumplir. Así, los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo. Además, este problema tiene la opción de contar con otras restricciones (como el volumen ocupado por los objetos), haciendo que el problema tenga múltiples restricciones [30].

Este ejemplo, junto a otros problemas de programación lineal entera, son de complejidad *NP-hard* [31]. Estos problemas son al menos tan complejos como los problemas *NP* [32], que corresponde a la clase de problemas de búsqueda y optimización donde se quiere saber si existe una solución única o alguna mejor que las conocidas. Se pueden resolver de manera exacta a través de algoritmos voraces [33] o de forma aproximada a través de algoritmos de aproximación [34].



$$\text{Maximizar } 4x_1 + 2x_2 + 2x_3 + 2x_4 + 10x_5$$

$$s.a \begin{cases} 12x_1 + 2x_2 + x_3 + x_4 + 4x_5 \leq 15 \\ x_1, x_2, x_3, x_4 \geq 0 \\ x_1, x_2, x_3, x_4 \in \mathbb{Z} \end{cases}$$

Figura 2.3: Problema de la mochila simple.

Los algoritmos voraces primero eligen el elemento de mayor valor ponderado. Se comprueba que la solución es factible con este elemento, y en caso de que sea así, se continúa con el elemento siguiente y así se sigue iterando hasta llegar a la solución óptima, lo que se alcanza cuando se haya iterado por todos los elementos o se hayan modificado todas las variables sin haber modificado del valor de la función objetivo.

Si bien es un método útil para resolver el problema, puede tomar bastante tiempo dependiendo del tamaño del modelo. Esto se debe a que la complejidad de estos algoritmos es generalmente cuadrática, lo que quiere decir que el tiempo de ejecución no aumenta proporcionalmente, sino que aún más. Aún así, utilizar este método es sencillo ya que tanto Java como Python poseen librerías que implementan este método, como por ejemplo las librerías MIP [35] o NAG [36].

La otra opción corresponde a resolver el problema de manera aproximada a través del uso de algoritmos de aproximación. Estos generalmente se utilizan cuando no se conoce ningún algoritmo eficiente que resuelva un problema dado en tiempo polinomial. Las soluciones entregadas por estos algoritmos son cercanas a las óptimas y sus tiempos de ejecución son en general más eficientes que los de los algoritmos voraces.

Al igual que para la clase de algoritmos exacto, ambos lenguajes también contienen librerías para usar algoritmos aproximados, donde se destaca el uso de algoritmos genéticos de la librería pyeasyga [37] de Python.

2.5. Trabajo relacionado

A continuación, describimos las soluciones ya existentes que abordan el problema planteado, es decir, *reducir el número de salidas* para realizar las compras del hogar, y explicamos por qué proveen solo una solución parcial.

[1]Fuente: `https://es.wikipedia.org/wiki/Problema_de_la_mochila`

Servicios de compras de primera necesidad

Hoy en día existen muchos servicios que permiten a clientes realizar compras de distintos productos sin salir de casa, incluyendo los de primera necesidad. Entre estos se destacan *apps* de delivery como Cornershop o Rappi, entre otras. El uso de estos servicios ha aumentado tanto a nivel mundial como nacional debido a la pandemia [38]. Sin embargo, estos servicios conllevan un número de desventajas que limitan su adopción.

Primero, su uso involucra pagar un costo de envío asociado a la compra, que depende generalmente del precio total de los productos solicitados y/o de la suscripción que el cliente tenga con el servicio. Más aún, dependiendo del comercio seleccionado para la compra, los productos no tienen siempre el mismo costo que en la tienda, sino que pueden llevar un recargo (nada despreciable) de hasta un 15 % [39]. Segundo, el reparto de los productos puede tomar un tiempo considerable, o incluso no concretarse debido a la alta demanda que se puede producir en ciertos horarios. Y por último, los repartidores suelen interactuar de forma física con varias personas (proveedores y clientes), resultando más expuestos al contagio, por lo que existe un riesgo adicional de que puedan trasmitir la enfermedad a los clientes.

Para profundizar los últimos dos puntos, analizamos los sucesos que han ocurrido este último año relacionados al uso de estas *apps* de delivery. Sobre la duración de los encargos, el uso de estos servicios ha aumentado más de la mitad comparado con la situación antes de la pandemia [40]. Esto se refleja no solo en el aumento de compras de primera necesidad, sino también en el delivery de locales establecidos como restaurantes o pastelerías. Este aumento de demanda puede provocar que los trabajadores de estos servicios demoren más en completar sus pedidos o que incluso deban ser cancelados.

En cuanto a las conductas de sanitización que deben seguir los repartidores, las principales compañías de servicios delivery establecieron medidas que sus trabajadores deben seguir y además ciertas ayudas para que puedan cumplir con estas medidas, como por ejemplo la entrega de alcohol gel y paños desinfectantes [41]. Sin embargo, ha habido evidencia de ocasiones en que la situación de estos trabajadores no les permiten acceso al baño ni elementos de protección [42], lo que aumenta sus probabilidades de contagio, sobre todo si deben trabajar más de lo normal debido a la alta demanda por la pandemia.

Servicios colaborativos

Otro enfoque para abordar este problema es el basado en *economía colaborativa*. Este corresponde a un modelo donde los productos (o en este caso los servicios) son considerados más que bienes económicos, bienes de intercambio, lo que involucra la interacción y colaboración entre varios clientes. Aplicado al caso de las compras de productos de primera necesidad, corresponde a "coordinarse" con otras personas como, por ejemplo, vecinos o personas que viven en el mismo hogar, para realizar las compras de manera colectiva. De esta manera no es necesario que salgan *todos* los vecinos, sino que es suficiente con que *solo uno* haga la compra (la persona designada a comprar puede, por ejemplo, rotarse de compra a compra). Para ello se pueden utilizar *apps* como Listonic [43] que permiten crear listas de supermercado de

manera colaborativa, donde personas pueden compartir listas para que otros usuarios puedan consultarlas y editarlas.

Para no tener que hacer la listas de forma manual, otras soluciones utilizan mecanismos de inteligencia artificial. Un ejemplo de estas es Pantrify [44], que permite generar listas de productos a partir de un recuento de lo que el usuario tiene actualmente guardado en su nevera, despensa o congelador.

Estas dos soluciones basadas en economía colaborativa permiten resolver el problema de la "confección" de una lista de compras, pero dejan otros problemas importantes sin resolver. Primero, no establecen ninguna política *de quién debería encargarse de la siguiente compra*, ni tampoco ningún mecanismo que incentive a los diferentes usuarios a que tomen dicha iniciativa. Segundo y principal, no considera las *limitaciones que un usuario tendrá al momento de hacerle las compras* a los demás usuarios. Seguramente podrá pagar y cargar un número limitado de productos adicionales a los propios. En este caso, *¿cómo elegir qué productos traerle a cada usuario?* Esta decisión debe tomarse de manera informada, siguiendo criterios razonables. Por ejemplo, se esperaría que un producto de "mayor" necesidad tenga prioridad sobre uno de "menor" necesidad; que sea preferible llevarle la compra a un usuario de edad avanzada que a uno joven, sin ningún factor de riesgo; o que los productos de un usuario que ya haya hecho la compra para todos los demás tengan mayor prioridad que aquellos de otro usuario que nunca haya hecho la compra para los demás. Ninguna de las soluciones existentes basadas en economía colaborativa abordan estos desafíos.

Otro sistema basado en economía colaborativa, pero que busca solucionar un problema distinto es KarPool [45]. Esta *app* le permite a usuarios compartir espacios de su vehículo privado para realizar aventones, a cambio de compartir costos asociados al transporte como por ejemplo la gasolina. Esto es conveniente para todos, debido a que el usuario compartiendo el vehículo no tiene que gastar tanto, y los pasajeros pueden llegar a su destino de forma más rápida. Además, esto contribuye a minimizar el problema del alto flujo en la vía pública, ya que se aprovecha al máximo el espacio de los vehículos particulares si una persona decide compartir espacio. Paralelamente, esto reduce la huella de carbono que producimos, lo cual es uno de los principales motivadores para que los usuarios se adhieran a la *app* y la utilicen.

Finalmente, otro ejemplo de economía colaborativa que aborda otro problema es Airbnb [46], donde los usuarios pueden ofrecer sus habitaciones o pisos que no están usando, para que otros usuarios puedan alquilarlos por un tiempo determinado. Este sistema es ocupado bastante por personas que van de viaje (sobretodo a ciudades grandes), debido a que este sistema reemplaza el alojamiento en otros lugares más tradicionales, pero que generalmente son más caros (ej: hoteles).

Capítulo 3

Diseño de la solución

En este capítulo explicamos en profundidad el diseño del prototipo junto a su justificación. Primero definimos los requisitos primordiales que se deben cumplir y detallamos la complejidad de estos. Posteriormente explicamos el diseño del prototipo de alto nivel, donde además justificamos como este cumple con los requisitos definidos. Finalmente, entramos en mayor detalle y describimos el diseño de 3 componentes importantes: el modelo de la selección de productos, diseño del *frontend* y el modelo de los datos a utilizar.

3.1. Requisitos

Lo primero que debemos definir para diseñar una solución que resuelva el problema planteado en la Sección 1.1 son sus requisitos. Esto corresponde a definir completamente el comportamiento que deseamos para nuestro prototipo a desarrollar en esta memoria. Es importante recordar que el sistema que buscamos desarrollar debe basarse en el modelo de economía colaborativa, donde los usuarios del sistema se ayudan mutuamente.

1. El sistema debe permitir el registro de usuarios, los cuales a la vez pertenecen a un grupo de usuarios.
    - Los datos sensibles que sean recopilados de los usuarios deben permanecer confidenciales.
2. El sistema debe determinar quiénes realizarán las compras por los usuarios del grupo.
3. Los usuarios deben ser capaces de solicitar productos dentro del sistema.
4. El sistema debe ser capaz de seleccionar los productos más prioritarios del conjunto de productos que solicitan los usuarios.
    - La selección de productos debe tomar en cuenta las "prioridades" de los distintos productos y condiciones de salud de los usuarios solicitantes.

- La selección de productos debe ser consistente con las potenciales restricciones que el comprador pueda tener.
- La selección de productos debe realizarse de forma automática y debe realizarse en un tiempo razonable (menos de 1 minuto).
- En caso de que algún producto de los elegidos originalmente no se encuentre, el sistema debe ser capaz de seleccionar automáticamente un reemplazo a partir de los productos que originalmente fueron descartados.

5. El sistema debe incentivar a los usuarios para que compren por los demás usuarios de sus respectivos grupos.
6. El sistema debe ser capaz de recordar los cobros que los compradores deben realizar una vez terminadas las compras de los productos.
7. El sistema debe ser lo suficientemente intuitivo y accesible para que pueda ser utilizado por todo tipo de usuarios.

3.2. Decisiones de diseño y desafíos

Mencionamos de manera breve las decisiones de diseño que podemos tomar para poder cumplir con los requisitos de la sección anterior. También hablamos sobre las dificultades y desafíos que conlleva materializar estos requisitos.

Respecto al registro de usuarios, el mayor desafío es determinar qué tipos de datos es necesario solicitar. Por ejemplo, para saber qué tan vulnerable es el usuario frente al COVID-19, ¿que tipo de enfermedades son necesarias saber? ¿Y si además la persona posee restricciones de movilidad que no le permiten salir de su casa con mayor facilidad? También es posible que los usuarios se sientan vulnerados indirectamente al preguntar esto, ya que pueden considerarla como información privada. En cuanto a la implementación misma del registro, no es una tarea muy complicada debido a que existen numerosos ejemplos de servicios que contienen páginas de registro.

Respecto a *quién* debería realizar las compras del grupo, existen diversas formas de implementarlo. Una opción sería que los usuarios dentro del grupo son libres de solicitar productos de cualquier tienda durante el transcurso de un día, y a cierta hora el sistema selecciona a una persona dentro del grupo a partir de una lista circular (*round-robin*). Esto hace que todos los usuarios compren por el grupo al menos una vez, antes de que un usuario tenga que comprar por los demás otra vez. Así también todos los usuarios deben salir la misma cantidad de veces. El problema de esto es que considera que todos los usuarios podrán efectivamente estar disponibles cuando sea su turno para comprar. Además, puede que existan usuarios que tengan dificultades para salir de su casa, como por ejemplo, puede que tengan que hacer cuarentena en su hogar debido a un contagio, que tengan restricciones de movilidad que les impida moverse de su hogar, etc. . . .

Otra opción correspondería a que primero las personas deben anunciar por su cuenta cuando van a salir a comprar, y a partir de ese momento el resto del grupo puede solicitar

productos. A diferencia de la opción anterior, los compradores deciden a cuáles tiendas dirigirse, entonces así los solicitantes podrán pedir productos exclusivamente de esas tiendas, lo que hace que el comprador no tenga que ir a otros locales que no tenía previsto visitar. Una desventaja de esto es que los solicitantes no tendrían tanto tiempo para solicitar los productos, por lo que el sistema tendría que notificar al resto de los usuarios en el momento exacto cuando los compradores anuncian su salida.

Sobre los productos que los usuarios pueden solicitar, debemos definir cómo estos se pedirán. Por ejemplo, si pueden seleccionarlos a partir de una lista de productos disponibles por defecto, o si pueden encargar un producto de forma manual ingresando el nombre, precio, etc. Esto también dependerá de la opción que se elija en el párrafo anterior. Por ejemplo, si se decide que los compradores deciden las tiendas donde comprar, los solicitantes solo deberían poder elegir entre los productos pertenecientes a dichas tiendas.

La automatización de la selección de productos corresponde a una de las componentes más complejas del prototipo. Además de modelar este problema, se debe encontrar una forma automática de resolverlo que tome en consideración distintos factores al momento de decidir seleccionar o descartar un producto. Estos factores podrían ser la edad del usuario solicitante, si posee factores de riesgo frente al COVID-19, qué tan necesario es el tipo de producto solicitado.

La selección de productos debe tomar en consideración también las posibles restricciones que pueden tener los usuarios compradores, ya que estas son las razones por las cuales se debe seleccionar un subconjunto de productos. En particular, estas pueden ser por ejemplo el monto máximo de dinero que el comprador es capaz de usar para los productos solicitados.

Por supuesto, el tiempo de resolución debe ser breve debido a que si toma demasiado tiempo, no lograremos desarrollar una solución completamente satisfactoria, ya que esto podría desincentivar la adopción del sistema.

Adicionalmente, los productos que no se seleccionan por el sistema deben ser considerados para después en caso de que uno de los seleccionados originalmente no se encuentre. Lo complejo aquí es determinar qué productos son los que pueden ser seleccionados cuando sucede esto, ya que es probable que un usuario comprador tenga que volver a una de las tiendas que ya visitó si justo se selecciona un producto de esa tienda que fue inicialmente descartado.

Hay varias formas en las que se podría incentivar la participación dentro del sistema. Una forma es a través de recordatorios, donde se le puede recordar a los usuarios que al momento de que estos salgan a comprar, podrían aprovechar de comprar productos para el resto del grupo. Otra forma no excluyente corresponde a aumentar la prioridad de los productos solicitados por usuarios que hayan comprado en ocasiones anteriores por los demás, para que así tengan más posibilidad de que estos sean seleccionados durante la selección automática de productos.

Tomar en cuenta los cobros que los usuarios deben realizar después de la compra no es una tarea compleja debido a que estos montos se pueden calcular a partir del precio de los productos comprados por los usuarios compradores.

Respecto al último requisito que corresponde a construir un sistema intuitivo y accesible para todo público resulta ser uno de los más desafiantes, debido a que este sistema planea ser utilizado por usuarios de todas las edades, incluyendo de la tercera edad. Es por esto que tanto el diseño del *frontend* como la elección de la plataforma de donde se podrá acceder se vuelven fundamentales para resolver el problema. En particular, las propiedades del *frontend* más fundamentales corresponden a que los usuarios puedan identificar los propósitos de cada menú, que los colores y el tamaño del texto pueda ser leído por todo tipo de usuarios, entre otros. Sobre las plataformas posibles, podemos construir un prototipo de escritorio, *web* o móvil, y elegir entre una de estas 3 alternativas para saber cuál de estas ofrece la mayor accesibilidad.

3.3. Descripción de alto nivel

A continuación describimos las características de nuestro prototipo y cómo estas satisfacen los requisitos establecidos en la Sección 3.1.

Establecemos que cada uno de los usuarios registrados además de poseer un nombre de usuario y contraseña (como cualquier cuenta de usuario de cualquier servicio *web*), también debe registrar su fecha de nacimiento para que el sistema pueda determinar su edad. Además, se decide mostrarle una lista de posibles factores de riesgo contra el COVID-19 y restricciones físicas de movilidad, donde el usuario debe seleccionar los que posea. Esto, junto a la fecha de nacimiento, sirven para determinar qué tan vulnerable es el usuario, lo que corresponde a un factor importante para la selección de productos (más detalles en la Sección 3.4).

Respecto al procedimiento de encargo de productos, decidimos que funcionará de la siguiente manera. Un usuario dentro del sistema es capaz de anunciar cuando esté a punto de salir de su hogar a comprar productos a través de la publicación de un *anuncio* dentro del sistema. Así, todos los demás usuarios dentro del grupo pueden revisar este anuncio y solicitar productos a través de él, donde se despliegan en una lista todos los productos disponibles que pueden solicitar. Estos van a depender a su vez de las tiendas que visite el usuario comprador.

El comprador, además de seleccionar las tiendas que visitará, también debe especificar las restricciones que este posee al momento de comprar. Estos corresponden a:

- **Monto máximo total:** Cuánto dinero extra tiene el usuario comprador para los productos encargados.
- **Peso máximo total:** Cuánto peso extra el usuario es capaz de llevar.
- **Cantidad de productos máxima:** Cuántos productos extra el usuario quiere o es capaz de llevar.

Esto se solicita para poder realizar la selección de productos de manera automática. En

particular, lo que hacemos es modelar la selección de productos como el problema de elegir un subconjunto óptimo a partir del conjunto de productos que le solicitan los usuarios de un grupo a sus compradores. Así, los productos elegidos se incluirán dentro de una lista de compras que los compradores deben editar a medida que van comprando dentro de la tienda, mientras que el resto pertenece a una lista de espera, de la cual se agregarán productos hacia la lista de compras en caso de que uno de los productos originales no se pueda comprar por algún motivo (ej: no había *stock*). Esto lo podemos modelar como un problema de optimización que busque maximizar *la utilidad total* de la compra, sujeto a las restricciones mencionadas anteriormente.

Para calcular la utilidad efectiva de un producto solicitado, cada uno de estos poseerá dentro del sistema 4 factores que se utilizarán al momento de realizar la selección automática. Primero, qué tan *necesario* es el producto solicitado, donde, por ejemplo, alimentos y/o medicinas deberían ser de la mayor necesidad posible. Segundo, qué tan *vulnerable* es el usuario solicitante del producto, lo que determinamos a partir de la edad del mismo usuario y si posee algún factor de riesgo frente al COVID-19 o restricción de movilidad. Tercero, si el usuario solicitante ha salido a comprar por el resto de los usuarios en ocasiones anteriores, ya que consideramos que los usuarios que ayudan a los demás deben tener cierta prioridad al momento de que ellos soliciten. Finalmente, consideramos la fracción de compras que el usuario solicitante efectivamente recibió la última vez que solicitó a un anuncio, para que así, en caso de que hubieran productos que no recibió la vez anterior, tenga mayor prioridad y así tenga menor probabilidad de que esto vuelva a suceder.

Una vez concretadas las compras de los productos solicitados, los usuarios compradores pueden revisar dentro del mismo sistema los montos que deben cobrar a los usuarios que solicitan. Es importante notar que este monto puede diferir respecto al precio del producto original (al momento de comprar, el comprador podría encontrarlo en oferta, o por el contrario, más caro), por lo que es responsabilidad del mismo comprador indicar cuánto dinero pagó efectivamente por cada producto comprado, además de realizar el cobro correspondiente a los usuarios del grupo.

Para poder desarrollar un sistema intuitivo y accesible para todo tipo de usuarios (y así cumplir con el último requisito), se toman numerosas decisiones de diseño. Primero que todo, se decide que el sistema en sí debe corresponder a una *app* móvil por sobre una *app* web o aplicación de escritorio, ya que planeamos que los usuarios sean capaces de utilizar este sistema fuera de su hogar mientras realizan las compras. Además, esto resulta más accesible ya que es más común para las personas (especialmente mayores) poseer un dispositivo móvil que un computador.

Para lograr que el sistema sea lo más intuitivo posible para todo tipo de usuarios, el diseño del prototipo debe tomar en cuenta fuertemente las opiniones y los deseos de los usuarios al momento de utilizar el sistema, por lo que se decide realizar pruebas de usabilidad tanto antes como después de la implementación del prototipo. Primero diseñaremos bocetos que representen todas las funcionalidades mencionadas anteriormente para mostrárselas a usuarios candidatos a través de entrevistas y así recibir *feedback* directamente de ellos para saber si se deben hacer cambios menores y/o mayores. Después de realizar los cambios correspondientes, se implementa el prototipo y se realizan otras pruebas de usabilidad (ver Sección 5.1 para

mayores detalles).

3.4. Selección de productos

Como mencionamos anteriormente, buscamos transformar el problema de selección de productos prioritarios en un modelo matemático donde buscamos maximizar el beneficio total que recibe el grupo al realizar las compras. En otras palabras, esto corresponde a la *utilidad total* de las compras realizadas.

Lo primero que se busca definir en el modelo matemático es la *utilidad* (u) asociada un producto p solicitado por un usuario. Este corresponde a un número real en el intervalo $[0, 1]$, el cuál dependerá de una serie de factores relacionados tanto a las características del producto como del usuario que lo solicita. Capturamos esta utilidad a través de una *función de utilidad* $u(p)$, la cual definimos más adelante.

A cada producto p solicitado en un anuncio le asociaremos una variable binaria x_p, donde $x_p = 1$ significa que el producto p es escogido para la compra colaborativa, y $x_p = 0$ significa que el producto es descartado. Estas variables binarias están indexadas por el conjunto P de productos requeridos por los usuarios en el anuncio:

$$x_p \in \{0, 1\} \qquad \forall p \in P$$

Con los datos definidos anteriormente, podemos definir nuestro problema de optimización, que en particular se tratará de un problema de *programación lineal entera*, donde buscamos maximizar la utilidad total de los productos efectivamente seleccionados para la compra sujeto a las tres restricciones mencionadas en la Sección 3.3:

$$\text{Maximizar} \sum_{p \in P} u(p) * x_p$$

$$s.a \begin{cases} \sum_{p \in P} c_p * x_p & \leq & M^{\$} & \text{(monto máximo)} \\ \sum_{p \in P} w_p * x_p & \leq & M^{\text{Kg}} & \text{(peso máximo)} \\ \sum_{p \in P} x_p & \leq & M^{\#} & \text{(cantidad máxima)} \end{cases}$$

Aquí $M^{\$}$ corresponde al monto máximo que el comprador tiene disponible para la compra de los productos solicitados por el grupo, M^{Kg} al peso máximo que el comprador es capaz de llevar, y $M^{\#}$ la cantidad de máxima de productos. Las variables c_p y w_p corresponden al costo monetario y peso del producto p, respectivamente; estos valores se encontrarán almacenados en una base de datos que mantendrá la *app*.

Utilidad de los productos

Dado un producto p, definimos los cuatro parámetros que definen su utilidad $u(p)$, a partir de características tanto del producto como del usuario solicitante:

- **Grado de necesidad del producto** ($\hat{p}_1$) **:** Qué tan imprescindible es el producto que se solicita.
- **Grado de vulnerabilidad del usuario** ($\hat{p}_2$) **:** Qué tan vulnerable es el usuario, a causa de algún factor de riesgo que posea frente al virus, o alguna restricción de movilidad que le impida/dificulte salir de casa.
- **Grado de participación del usuario** ($\hat{p}_3$) **:** Qué tanto el usuario que solicita el producto ha salido a comprar para el resto de los usuarios en ocasiones anteriores.
- **Grado de *starving* del usuario** ($\hat{p}_4$) **:** La fracción de productos solicitados por un usuario que no se pudieron comprar (del conjunto de productos solicitados en una ocasión anterior).

Definimos la utilidad $u(p)$ de un producto p a partir de estos cuatro parámetros $\hat{p}_1, \ldots, \hat{p}_4$. Concretamente, los agregamos considerando una combinación convexa:

$$u(p) = \sum_{i=1}^{4} k_i * \hat{p}_i \ ,$$

donde el factor de ponderación $k_i \in [0, 1]$ representa el peso o relevancia del parámetro p_i y $\sum_{i=1}^{4} k_i = 1$. Como veremos más adelante, cada parámetro p_i se define en el intervalo $[0, 1]$, lo que hace que la función utilidad u tome también valores en el intervalo $[0, 1]$.

Inicialmente, propusimos que todos los factores de ponderación k_i sean igual a $1/4$, para que todos los parámetros tengan la misma relevancia. Sin embargo, posteriormente realizamos unas encuestas iniciales para saber si ciertos factores son más importantes que otros. Gracias a estas, nos damos cuenta que efectivamente las personas suelen tomar un factor con mayor importancia sobre los demás, por lo que modificamos los factores k_i.

Detallamos el cálculo de los parámetros $\hat{p}_1, \hat{p}_2, \hat{p}_3, \hat{p}_4$ a continuación. Para determinar qué tan esencial es el producto que solicita un usuario (parámetro $\hat{p}_1$), primero buscamos fuentes que muestren los productos más necesarios en Chile. A partir de una lista de los productos esenciales en pandemia hecha por los ministerios de Comercio, Salud, y Agricultura [47], y también una lista de la canasta básica elaborada por la Pontificia Universidad Católica [48], generamos 3 categorías de productos, cada una con su respectivo valor del parámetro:

- **Primera necesidad:** $\hat{p}_1 = 1$. Corresponde a medicamentos, dispositivos médicos, frutas y verduras, lácteos, papel confort, pasta de dientes, entre otros.
- **Segunda necesidad:** $\hat{p}_1 = 0{,}5$. Corresponde a pan, cereales, carnes, café, té, entre otros.

- **Otros:** $\hat{p}_1 = 0$. Corresponde al resto de los productos como por ejemplo, bebidas alcohólicas y golosinas.

La vulnerabilidad del usuario solicitante (parámetro $\hat{p}_2$) la podemos analizar a partir de dos variables: su edad ($\hat{p}_{2a}$), y si tiene factores de riesgo y/o condiciones de movilidad reducida ($\hat{p}_{2b}$). Cada uno de estos dos subparámetros tendrá un valor en el intervalo [0 , 0,5] y el valor parámetro completo lo obtendremos simplemente sumando los valores de ambos subparámetros:

$$\hat{p}_2 = \hat{p}_{2a} + \hat{p}_{2b}$$

Para el caso de la edad, primero buscamos fuentes para saber a partir de qué edades un contagio por coronavirus se puede volver mortal [49]. Dentro de la información más destacable que se adquiere, es que la tasa de mortalidad del grupo de edad entre 50-59 años (1,3 %) corresponde al menos el triple con respecto a personas menores (0,4 %). A su vez, para las personas entre 70 y 79 años aumenta aún más (8 %). Con estos datos en mente, generamos 3 categorías a partir de los rangos de edades:

- **Usuario entre 0-49 años de edad:** $\hat{p}_{2a} = 0$.
- **Usuario entre 50-69 años de edad:** $\hat{p}_{2a} = 0{,}25$.
- **Usuario mayor de 70 años de edad:** $\hat{p}_{2a} = 0{,}5$.

Para el caso de los factores de riesgo, investigamos cuáles son considerados graves frente al COVID. De acuerdo a la OMS [50], las personas con afecciones médicas subyacentes como hipertensión arterial, problemas cardíacos o pulmonares, diabetes, obesidad o cáncer, corresponden a las personas con mayor riesgo. Aún así, al momento que se realiza esta memoria también se cree que ciertas afecciones pueden ser más graves que otras [51]. Es por esto que clasificamos estos factores en 2 grupos: de mayor riesgo (por ejemplo obesidad o diabetes) o de menor riesgo (asma, hipertensión, entre otros).

En esta categoría también consideramos condiciones de movilidad particulares del usuario, y si estas son temporales (por ejemplo fracturas o esguinces) o crónicas (parálisis, ceguera, entre otros). Así, definimos 3 categorías dependiendo de la cantidad y tipo de condiciones/-factores de riesgo:

- **Usuario sin factor de riesgo o condición de movilidad** $\hat{p}_{2b} = 0$.
- **Usuario con un factor de riesgo menor y/o condición de movilidad temporal** $\hat{p}_{2b} = 0{,}25$.
- **Usuario con un factor de riesgo mayor y/o 2 factores de riesgo menor y/o condición de movilidad crónica** $\hat{p}_{2b} = 0{,}5$.

Para cuantificar el grado de participación de un usuario dentro de la *app* (parámetro $\hat{p}_3$), analizamos si el usuario ha actuado como el comprador en algunas de las últimas compras en las que ha participado dentro del sistema, generando las siguientes 4 categorías:

- **Usuario que fue comprador en alguna de las últimas 5 compras donde participó** $\hat{p}_3 = 1$.
- **Usuario que fue comprador en alguna de las últimas 10 compras donde participó** $\hat{p}_3 = 0{,}5$.
- **Usuario que no fue comprador en ninguna de las últimas 10 compras donde participó y no es vulnerable** $\hat{p}_3 = 0$.
- **Usuario que no fue comprador en ninguna de las últimas 10 compras donde participó y es vulnerable (es decir** $\hat{p}_2 \geq 0{,}5$**)** $\hat{p}_3 = 0{,}5$.

Primero debemos aclarar que el $\hat{p}_3$ como tal depende de otra variable perteneciente a los usuarios llamada *morosidad*. Lo que hace esto es contar todos los pedidos que el usuario encarga y son confirmados exitosamente desde la última vez que este fue a comprar por los demás. Así, indica la cantidad de compras seguidas en las que ha participado sin ser el comprador. Si el valor es menor a 5, la participación $\hat{p_3}$ del usuario es de 1. Si se encuentra entre 5 y 10, su participación es de 0,5. Y finalmente, si el valor es mayor a 10 y el usuario no es medianamente vulnerable ($\hat{p_2} < 0{,}5$), su participación es de 0. Por otro lado, si el usuario comprador realiza al menos 1 compra exitosa, su valor de *morosidad* se reinicia a 0 por ser el comprador de la última salida realizada en ese momento, haciendo que su participación tenga un valor de 1.

Cabe destacar que se opta por evitar que este factor baje a 0 para los usuarios vulnerables para que no sufran una penalización muy alta cuando su posibilidad de salir de casa se vea limitada por problemas de salud. En este caso, consideramos a un usuario como vulnerable cuando su parámetro $\hat{p}_2$ tiene un valor de 0,5 o mayor.

Finalmente, para analizar el grado de *starving* de los usuarios (parámetro $\hat{p}_4$) y evitar que existan usuarios "rezagados" que nunca reciban los productos que solicitan, la *app* asignará a cada usuario una variable α, que llamaremos *starving factor*. Este valor inicialmente será 0 para todos los usuarios y aumentará cuando el sistema descarte algún producto que el usuario haya solicitado. Este valor dependerá de la fracción de productos solicitados que fueron rechazados en un encargo. Esta fracción es calculada cuando un usuario comprador termina de realizar todas las compras que le encargaron, ya que en este instante la *app* sabe con certeza cuáles productos fueron comprados y cuáles no. Así, por ejemplo si un usuario solicita a un comprador 10 productos de los cuales solo se compran 6, como le faltaron 4 productos, su nuevo valor α será de 0,4.

Luego ordenamos a los usuarios de cada grupo de acuerdo a sus respectivos *starving factors*, estableciendo así una cola de prioridad (dinámica). Así, los usuarios en la parte superior de la cola tendrán los *starving factors* más altos, mientras que los usuarios en la parte inferior de la cola tendrán los *starving factors* más bajos. Una vez realizado el orden, asignamos los valores $\hat{p}_4$ de manera que su valor aumente de forma lineal a los largo de la

cola de prioridad, cubriendo el intervalo [0,1]. Por ejemplo, si la cola contiene 3 usuarios: uno con *starving factor* 0,2, otro con 0,3 y el último con 0,9, sus valores de $\hat{p}_4$ serán de $\frac{1}{3}$, $\frac{2}{3}$ y 1 respectivamente. Cabe destacar que los productos de usuarios con un valor α igual a 0, tendrán por defecto un $\hat{p}_4 = 0$, independiente de su posición en la lista debido a que estos usuarios no tuvieron productos rezagados en la última compra que ellos participaron. Después de que a un usuario le compran todos sus productos dentro de una compra colaborativa, el valor de α se "resetea" y vuelve a 0.

Ponderación inicial de parámetros k_i

Con el objetivo de establecer los valores k_i iniciales para la función de utilidad, realizamos unas encuestas generales sobre las "compras de primera necesidad" que realizan las personas tanto antes como durante la pandemia. Además, indagamos sobre cuáles de los factores propuestos para el algoritmo realmente son los que le importa más a la gente, a través de la presentación de un problema. Este corresponde a elegir a qué usuarios estaría dispuesto a ayudar dentro de un grupo de usuarios que le solicita productos.

Esto último lo realizamos con el objetivo de saber cuáles son los parámetros más importantes para las personas al momento de realizar una selección de productos "justa". El resultado de esta encuesta hará que la ponderación de los parámetros $\hat{p}_i$ (k_i) de la función de utilidad se pueda editar para que esté más alineado con los intereses de los usuarios.

Respecto a los detalles de la encuesta en sí, la creamos usando un formulario de Google Forms, el cual el memorista compartió por sus redes sociales (Facebook, Twitter, entre otras), para que tanto amigos como conocidos del memorista pudieran responderla. Una vez creada la encuesta, mantuvimos su acceso público por alrededor de una semana, la cual posteriormente cerramos.

La encuesta fue respondida por un total de 76 personas, donde se les preguntó que tan dispuestos estaban a comprar productos para ciertos vecinos, cuya información incluyó su edad, que tan saludable son, el producto requerido, y si este había comprado productos antes al resto del vecindario. En particular, por cada uno de estos vecinos, se les pide responder con un número del 1 al 5, donde 1 significa que no le comprarían en ninguna circunstancia el producto, y 5 que se lo comprarían en cualquier circunstancia. A continuación, mostramos el número promedio resultante de cada uno de los vecinos a partir de las respuestas de los encuestados:

- Vecino saludable que quiere comprar tallarines y medicamentos (entre 30 y 40 años de edad): **3,369**
- Vecino con condiciones de movilidad reducida que quiere comprar cervezas y carbón (entre 50 y 60 años de edad): **3,354**
- Vecino con factor de riesgo frente al coronavirus (ejemplo: cáncer) que quiere comprar un set de pinturas acrílicas (entre 20 y 30 años de edad): **3,721**
- Vecino el cuál le compro a usted todos los productos que necesitaba la semana pasada (tiene la misma edad que usted): **4,289**

A partir de estos resultados podemos concluir lo siguiente: las personas aprecian mucho a las personas que ayudan a la comunidad, lo que se refleja en la alta puntuación del último vecino, por lo que se aprecia bastante la *participación* de estos. Caso contrario puede ser la necesidad del producto, debido a que el primer vecino recibe una puntuación baja, muy parecida a la del segundo que solicita cervezas. Lo que también la gente considera importante son los factores de riesgo que pueden tener las personas, haciendo que el tercer vecino tenga la segunda puntuación más alta.

Así, establecemos que las ponderaciones $\hat{k_i}$ de las distintas variables $\hat{p_i}$ para calcular la utilidad de los productos en el algoritmo de selección deben tener los siguientes valores:

- Ponderación para necesidad de producto $\hat{k_1} = 0{,}1$.
- Ponderación para vulnerabilidad de vecino $\hat{k_2} = 0{,}3$.
- Ponderación para participación de vecino $\hat{k_3} = 0{,}4$.
- Ponderación para *starving* de vecino $\hat{k_4} = 0{,}2$.

3.5. Diseño de prototipos

Después de comprobar que efectivamente se puede automatizar la selección de productos, diseñamos el *frontend* del prototipo. Este tiene que estar fuertemente alineado con los casos de uso principales. Dentro de estos, se encuentran los siguientes:

- Registrarse como nuevo usuario e ingresar como usuario existente.
- Crear nuevo grupo de usuarios o unirse a un grupo existente.
- Anunciar cuándo se realizará una salida a comprar y a qué tiendas.
- Encargar productos a usuario a través del anuncio de su compra.
- Seleccionar qué productos solicitados se compraron y eliminar los que no se compraron.
- Ver los cobros que el comprador debe realizar a los demás usuarios.

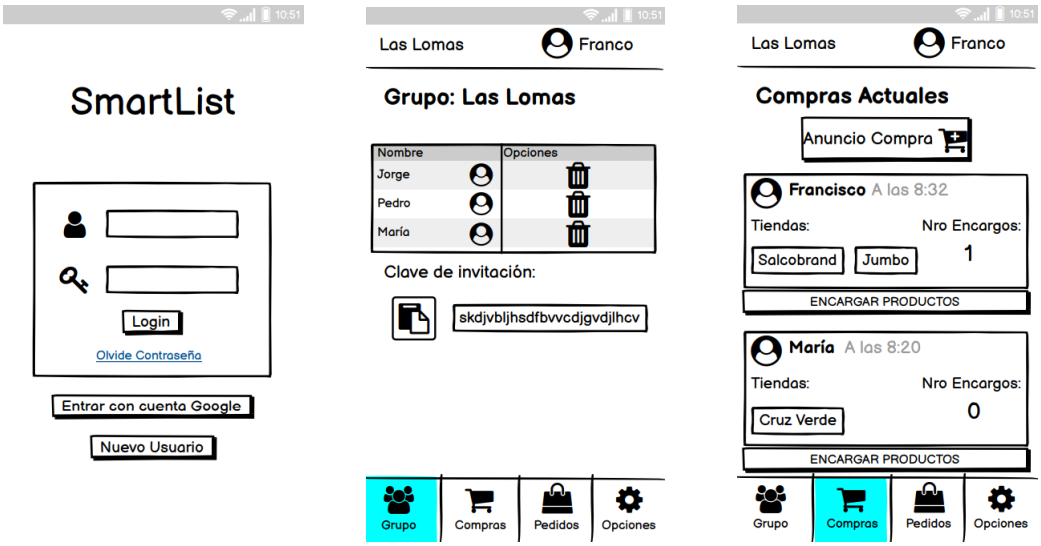
A partir de estos casos de uso, generamos *mockups* para utilizarlos en pruebas iniciales de usuario, que servirán para identificar si el prototipo satisface los criterios de usabilidad más importantes, como por ejemplo si la iconografía es la adecuada, si las instrucciones son claras, el manejo de colores, entre otros.

Con motivo de priorizar el tiempo de desarrollo de la *app*, los *mockups* que diseñamos estuvieron inspirados en las interfaces de soluciones existentes como Rappi o Cornershop, ya que al llevar en el mercado unos años, son soluciones consolidadas y los usuarios reportan, en general, buenas experiencias de uso.

Primero se diseña cómo los usuarios registrados pueden acceder a la *app*. Pueden hacerlo a través de un formulario donde deben ingresar su nombre de usuario y contraseña. Además, si no tienen un usuario registrado, pueden crear uno nuevo. La imagen (a) de la Figura 3.1 corresponde al prototipo de la interfaz junto a otras funcionalidades deseadas como recuperación de contraseña e ingreso con cuenta de Google.

Inicialmente, la estructura propuesta para la *app* se compone de 4 menús principales. El primero de estos corresponde a un menú de *Grupo* donde los usuarios son capaces de ver los nombres de los usuarios integrantes de su grupo. Adicionalmente, contiene una clave que los usuarios pueden compartir para que otras personas puedan unirse al grupo dentro de la *app*. El diseño de este menú se puede observar en la imagen (b) de la Figura 3.1.

El menú de *Compras* (ver imagen (c) de la Figura 3.1) corresponde a las "salidas a comprar" anunciadas por los usuarios que deseen recibir encargos de su grupo. Cada una de estas salidas despliega las tiendas a donde se dirigirán los compradores, el número de encargos recibido, y la hora a la que se anunció la salida. Así, un usuario puede ver los detalles de casa salida y encargarle productos al comprador (a través de un botón), para que este no tenga que salir de su hogar a comprarlo.



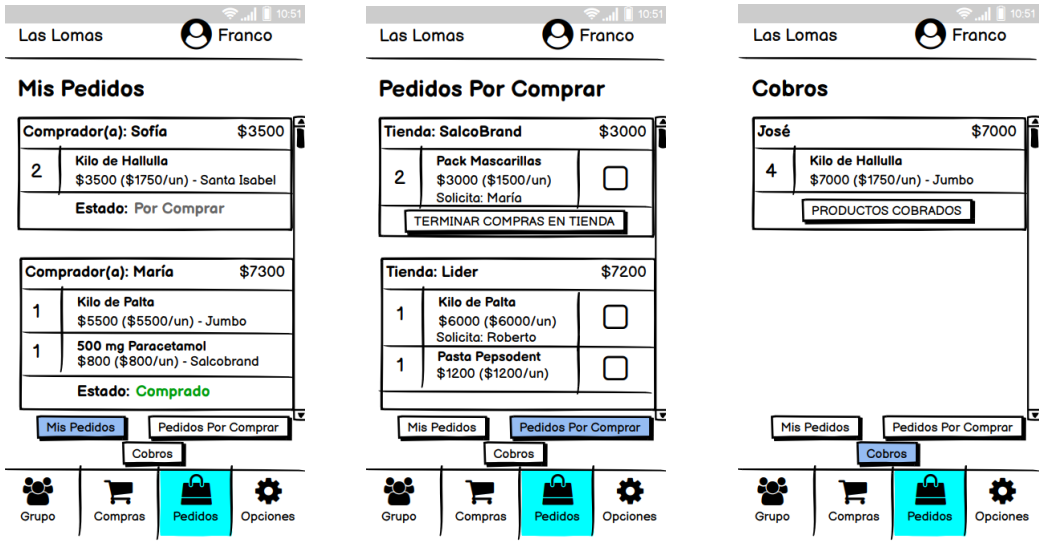
(a) Formulario de ingreso de usuarios.

(b) Menú de *Grupo*.

(c) Menú de *Compras*.

Figura 3.1: Primeros *mockups*.

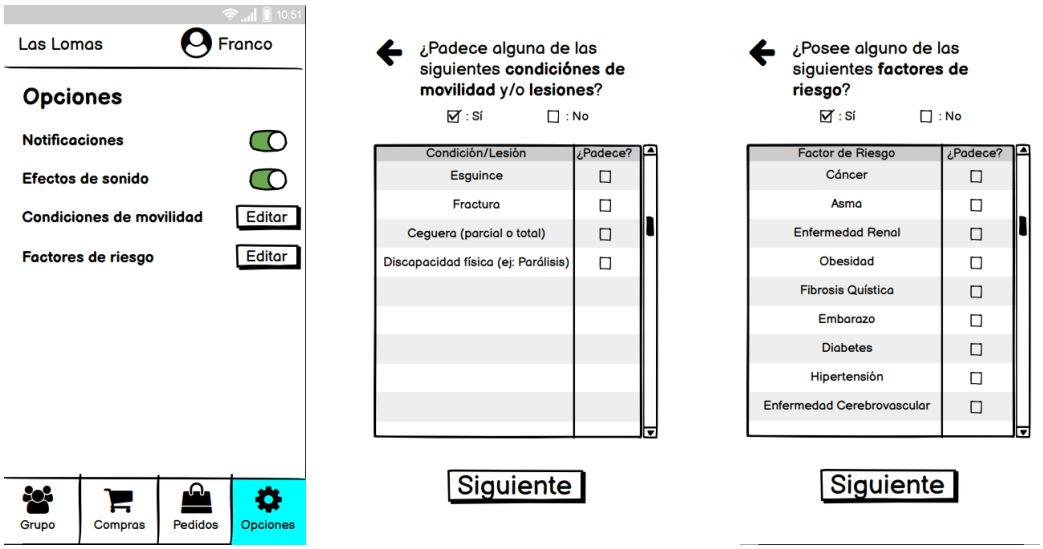
El menú de *Pedidos* (ver Figura 3.2) se encuentra divido en 3 submenús. El primero corresponde a *Mis Pedidos*, donde el usuario puede revisar el estado de los encargos que solicitó a los distintos usuarios compradores, junto al costo total de cada encargo. *Pedidos por Comprar* contiene una lista de los productos que el usuario debe comprarle al resto dividido entre las distintas tiendas por las que debe pasar. Aquí también debe confirmar qué productos fueron comprados exitosamente, y cuáles no. El último submenú de *Cobros* contiene información de los productos ya comprados que el usuario debe entregar a los solicitantes junto a la cantidad de dinero que les debe cobrar por la compra.



(a) *Mis Pedidos.* (b) *Pedidos Por Comprar.* (c) *Cobros.*

Figura 3.2: Menú de *Pedidos*.

Finalmente, consideramos también contar con un menú de *Opciones*, donde el usuario además de contar con opciones comunes dentro de *apps* móviles como activar o desactivar notificaciones, pueda editar sus propias condiciones de movilidad que puedan restringir su movimiento fuera del hogar, al igual que sus factores de riesgo frente al coronavirus. En la Figura (3.3) puede observar tanto el prototipo del menú como la configuración de las condiciones del usuario.



(a) *Opciones.* (b) Editor de condiciones de movilidad. (c) Editor de factores de riesgo.

Figura 3.3: Menú de *Opciones*.

Una vez que terminamos con el diseño de prototipos, realizamos 2 pruebas de usuario para comprobar qué tan intuitivo resulta el diseño propuesto para la *app*, y recibir *feedback* sobre qué aspectos se pueden mejorar. Los participantes de estas pruebas fueron conocidos

del memorista, a los cuales se les mostraron los *mockups* y se les dio la tarea de describir cada uno de sus elementos y mencionar para qué creen que sirven. Dentro de los comentarios más destacados, se encuentra que hay una falta de opciones al momento de confirmar las compras en *Pedidos por Comprar*, debido a que puede ocurrir qué el dinero gastado en ese producto no sea el mismo precio propuesto dentro de la *app*, o también puede que se compre una cantidad menor a la solicitada debido a falta de *stock* dentro de la tienda. Adicionalmente, se piensa que preguntar directamente por las condiciones del usuario puede resultar invasivo para su privacidad, por lo que sería ideal tener un mensaje de *disclaimer* refiriéndose a la confidencialidad de esta información y que el resto de los usuarios no tendrán acceso a esta información confidencial. Finalmente también se reciben comentarios sobre los menús de pedidos que encontraban que exponen demasiada información en poco espacio, lo que puede abrumar al usuario.

3.6. Modelo de datos

Respecto al modelo de datos a utilizar en nuestro prototipo, consideramos 2 lenguajes: SQL y NoSQL. Ambos poseen sus propias ventajas y desventajas. Para mayor información puede revisar la Sección 2.3.

Después de analizar ambas alternativas, optamos por utilizar SQL, es decir, una base relacional. La razón principal de esto son la cantidad de consultas por segundo que se espera procesar en nuestro prototipo. Además, se comprueba que esta era la alternativa correcta durante la implementación debido a la complejidad de ciertas consultas. Adicionalmente, el memorista posee mayor experiencia trabajando con este tipo de datos, lo que hace que la implementación de esta parte del prototipo sea desarrollada sin mayores inconvenientes.

Modelamos los datos necesarios para soportar las funcionalidades de la *app*, identificando con exactitud los atributos que se necesitan tanto para la interacción con los usuarios, como para resolver el problema de selección de productos bajo las restricciones del comprador. Vale la pena mencionar que todos los elementos de los modelos contienen un *id* único dentro de cada modelo que actúa como llave primaria dentro de estos. A continuación, listamos los modelos implementados en la base de datos (ver Figura 3.4):

1. **Usuarios:** Información de los usuarios registrados en la *app*. Además de contener las credenciales de acceso al sistema junto al *id* del grupo al que pertenecen, también incluye atributos para indicar su nivel de participación, *starving* y vulnerabilidad que son utilizados para calcular la prioridad de los distintos productos solicitados, por el algoritmo de selección. Es importante destacar que cada usuario debe pertenecer exactamente a un grupo a la vez.

2. **Grupos:** Representa a un grupo de usuarios dentro de la *app*. Cada grupo se compone de un nombre, clave de acceso para entrar al grupo y la *id* del usuario creador del grupo (no confundir con *id* del grupo).

3. **Condiciones:** Condiciones de movilidad que pueden tener los usuarios, con distintos niveles de prioridad dependiendo de la gravedad de estas. Los usuarios pueden indicar si poseen alguna de estas condiciones al momento de registrarse, pero también pueden editar su perfil en el caso de que dejen de poseer una o contraigan una nueva.

4. **Factores:** Representa a los factores de riesgo que los usuarios pueden poseer frente al coronavirus. Funciona de forma similar al modelo de Condiciones.

5. **Usuarios X Condiciones:** Tabla de datos que empareja a los usuarios de la *app* junto a las condiciones de movilidad que poseen, a través de las *ids* de los modelos Usuarios y Factores. Con esto se puede saber exactamente qué condiciones pueden tener los usuarios y sirve para cuantificar parte del índice de vulnerabilidad de estos.

6. **Usuarios X Factores:** Similar a Usuarios X Condiciones, pero utiliza la información de Factores en vez de la de Condiciones.

7. **Tiendas:** Representa a las distintas tiendas donde los usuarios pueden dirigirse a comprar y recibir encargos de parte de su grupo a través de la *app*. Para efectos de la memoria, se establecen tiendas aleatorias pertenecientes al comercio establecido de distinto tipo incluyendo supermercados (Jumbo, Líder) y farmacias (Salcobrand, Cruz Verde).

8. **Productos:** Información sobre todos los productos disponibles a encargar de todas las tiendas del modelo de Tiendas. Al igual que este modelo, la información utilizada para esta memoria es aleatoria e introducida manualmente dentro de la base de datos con el objetivo de poder experimentar y realizar *testing*. Incluye información sobre el nombre, precio, peso y el *id* de la tienda a la cuál pertenece (suponemos que cada producto solo puede pertenecer a una tienda), pero además incluye un nivel de *necesidad* que indica qué tan necesario efectivamente es el producto, de acuerdo a las condiciones establecidas en el modelo de selección de productos (ver Sección 3.4).

9. **Anuncios:** Corresponden a los anuncios que realizan los usuarios dentro de la *app* cuando van a salir a comprar, para que puedan recibir encargos de los demás usuarios de su grupo. Incluye, además, las restricciones que puedan tener al momento de salir a comprar, es decir, el presupuesto máximo destinado a los encargos recibidos, cantidad máxima de productos a comprar, y también el peso máximo adicional que pueden cargar.

10. **Anuncios X Tiendas:** Tabla de datos que empareja a los anuncios realizados en la *app* con las tiendas que el usuario comprador planea visitar, a través de las *ids* de los modelos Anuncios y Tiendas. Esto se hace para que cada anuncio despliegue exactamente a qué tiendas se dirigirá el usuario comprador al momento de anunciar su salida.

11. **Pedidos:** Corresponde a los pedidos o encargos que los usuarios pueden solicitar a los usuarios compradores. Cada pedido representa a un producto (identificado por su *id*), junto a la cantidad deseada por parte del solicitante. Contiene además los *ids* del producto solicitado, del usuario solicitante, del usuario comprador, de la tienda, y del anuncio a través del cual se realiza el pedido. La razón por la cual duplicamos el *id* de la tienda en esta tabla cuando podemos adquirirlo en *Productos*, es para optimizar el tiempo de consultas de la *app* cuando se quiere saber el nombre de las tiendas de ciertos pedidos. Adicionalmente, incluye un conjunto de atributos de estado para indicar

si el pedido se encuentra en progreso, si efectivamente fue elegido por algoritmo, si se encuentra dentro de la lista de espera, o si simplemente fue eliminado.

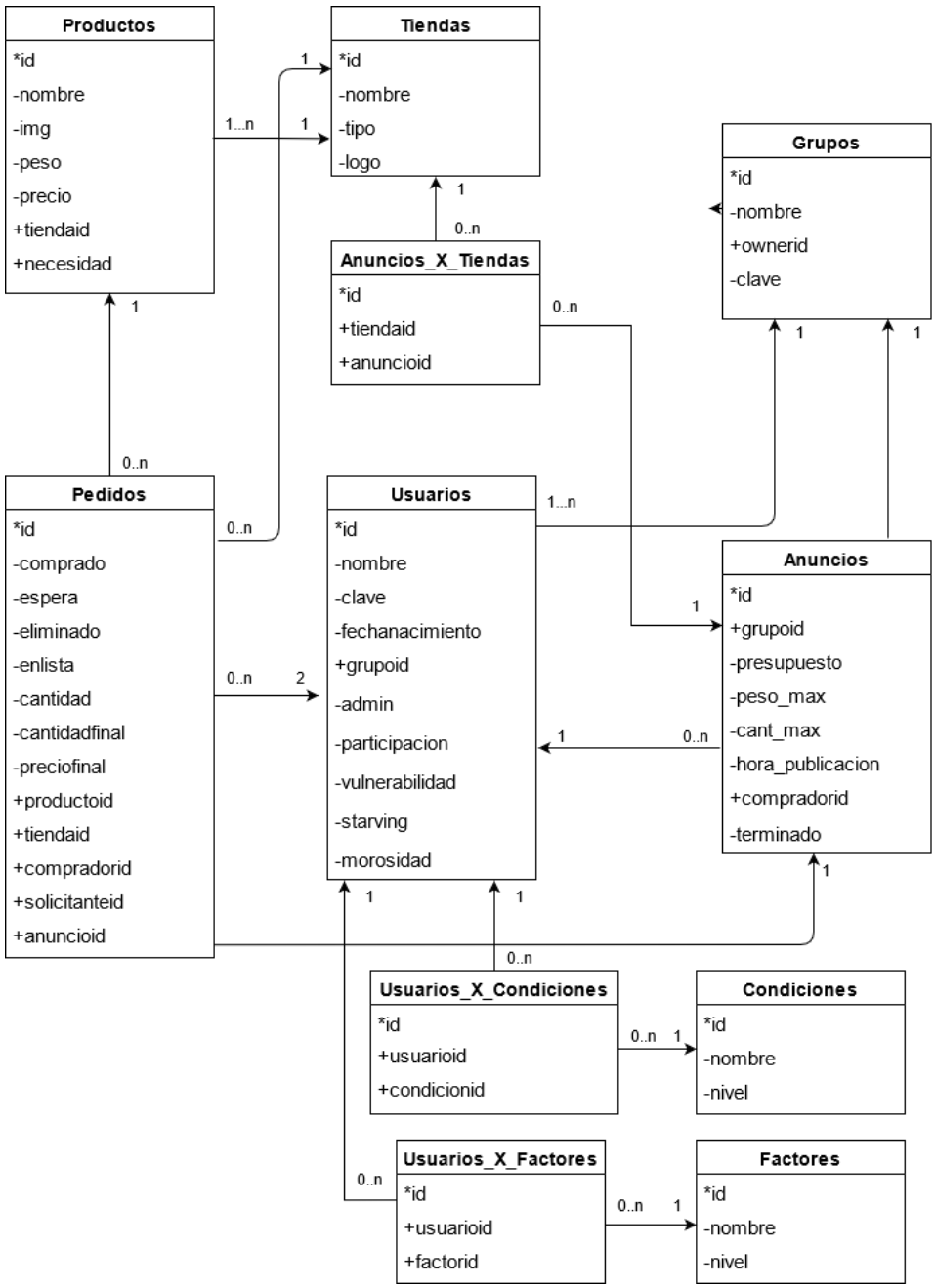


Figura 3.4: Modelos de datos junto a sus interacciones.

Para efectos de la memoria, los datos que utilizamos respecto a las tiendas y sus productos no reflejan la realidad en su conjunto, sino que utilizamos datos de prueba creados a partir de los productos disponibles en los sitios *web* de las tiendas que decidimos incluir en el sistema. Por ejemplo, para la tienda Salcobrand incluimos distintos tipos de remedios como productos, y para la tienda Jumbo incluimos alimentos de primera necesidad (arroz), de segunda necesidad (carnes) y otros productos de necesidad mínimas (cervezas), abarcando así todos los grados de necesidad.

Capítulo 4

Implementación de la solución

A partir del diseño presentado en el capítulo anterior, describimos en detalle la implementación del prototipo. Exponemos las tecnologías que utilizamos tanto para resolver el problema de selección, como para el desarrollo de la *app* en sí. Adicionalmente, explicamos el funcionamiento del producto final[1].

4.1. Solución modelo de selección de productos

Ahora que tenemos la función que se busca maximizar y las 3 restricciones que definen el espacio de soluciones factibles, investigamos sobre qué soluciones algorítmicas son las adecuadas para resolver nuestro problema. Dado que nuestro problema de optimización cae dentro de la categoría del problema de la mochila (ver Sección 2.4), partimos estudiando este problema. Además, como nuestro problema no está sujeto a una, sino que a varias restricciones, estudiamos en particular la versión multidimensional del problema de la mochila. Como este se trata de un problema *NP-hard*, primero buscamos resolverlo de manera heurística usando algoritmos de aproximación.

Una de las maneras más sencillas de encontrar soluciones aproximadas es a través de algoritmos genéticos. Para implementar esta solución, usamos una librería de Python diseñada para implementar algoritmos genéticos de manera sencilla llamada pyeasyga. Para probar esta solución aproximada, implementamos un programa de prueba con productos ficticios con sus respectivas utilidades, costos y pesos.

Dado que tenemos una cantidad de 3 dimensiones en nuestro problema (valor, peso y cantidad de productos), y el problema no involucrará una cantidad masiva de productos (los productos a elegir representan a las variables binarias de nuestro problema), también experimentamos resolviéndolo con algoritmos de solución exacta. Específicamente, probamos con la librería MIP de Python, la cual provee herramientas para resolver problemas de optimización lineales a variables enteras. Como nuestro problema se trata de variables binarias, podemos

[1]El código completo de este trabajo se encuentra en el repositorio Git `https://github.com/GCarboni96/smartlist/tree/master`

usar esta librería para resolverlo.

Específicamente, probamos ambas soluciones en un problema de prueba con una lista de 10 productos dentro de la base de datos, cada uno con distintos parámetros (solicitados por distintos usuarios), lo que resulta en productos con variadas utilidades asociadas. La solución aproximada usando algoritmos genéticos tomó un tiempo aproximado de 0,13 segundos en resolverse, mientras que la solución exacta usando la librería MIP tomó alrededor de 0,28 segundos, casi el doble que la anterior.

Sin embargo, descubrimos que el subconjunto escogido por el algoritmo genético no es consistente, lo que corresponde al comportamiento esperado por un algoritmo de este tipo. En ciertas ejecuciones, este no llega a calcular realmente el subconjunto con mayor utilidad, con un error aproximado entre 5 % y 10 %. En cambio el algoritmo de solución exacta es consistente con sus resultados siempre obteniendo el óptimo global. Dicho de otra manera, el algoritmo exacto siempre escogerá un subconjunto de mayor (o igual) utilidad que el algoritmo (genético) aproximado. Además, en términos de usabilidad, 0,14 segundos más pueden ser considerados despreciables para el usuario. Posteriormente comprobamos que las diferencias de tiempos con ejemplos de mayor tamaño (50 productos), donde el tiempo de la solución aproximada alcanza hasta a triplicar el tiempo del ejemplo anterior (llegando aproximadamente a 0,5 segundos), mientras que el de la solución directa se mantiene alrededor de 0,3 segundos. Esto significa que la solución exacta es más rápida que la aproximada en estos casos.

Debido a que el tiempo de cómputo de la solución exacta se mantiene bajo (menos de 1 segundo) para los problemas del tamaño que se espera que la *app* genere, decidimos utilizar un algoritmo de solución exacta para resolver el problema de optimización.

4.2. Tecnologías utilizadas

Para el desarrollo de esta *app* optamos por utilizar el lenguaje Python debido a su eficiencia e integración de numerosos módulos desarrollados por terceros. Dentro de estos, se encuentra la librería Kivy, la cual ofrece numerosas herramientas de creación de *frontend* para *apps* capaces de funcionar en distintas plataformas (más información en la Sección 2.1). Además de esto, otra librería relevante que utilizamos es MIP, la que nos permite resolver el problema de selección de productos de manera exacta en un tiempo razonable.

Otra ventaja de este lenguaje es su capacidad de manejar estructuras complejas. Esta característica, junto al hecho de que se es capaz de realizar programación orientada a objetos, nos permite desarrollar la lógica y *backend* de forma exitosa. Además, es uno de los lenguajes conocidos y uno de los más utilizados por el memorista.

Respecto al almacenamiento de datos, consideramos 2 alternativas. La primera corresponde a usar un ordenador para montar un servidor de prueba capaz de almacenar la información

y acceder a ella de forma directa. Para hacer esto, podemos utilizar algún software dedicado a montar servidores *HTTP*. La otra corresponde a utilizar servicios *cloud* de almacenamiento.

Después de analizar los costos entre montar un servidor local y utilizar servicios *cloud* para el almacenamiento de los datos, finalmente se opta por la segunda opción. Esto es debido a que el memorista no posee al momento de desarrollar la memoria un dispositivo apto para la mantención y constante actualización de una base datos.

En concreto, optamos por usar los servicios de IBMCloud debido a que estos incluyen planes de almacenamiento para *backend* gratis hasta un cierta capacidad y número de conexiones simultáneas. En particular, decidimos utilizar el servicio Db2 que ofrece almacenamiento de datos relaciones (SQL) con capacidad de conectarse a través de una API REST incluida dentro del mismo servicio para poder realizar consultas a la base de datos.

4.3. Desarrollo de la *app*

En esta sección describiremos primero la arquitectura de la *app* explicando también cómo esta interactúa con los componentes desarrollados hasta ahora. Seguido de esto, describiremos en detalle los distintos casos de uso definidos anteriormente (ver Sección 3.5).

4.3.1. Arquitectura de la *app*

Para desarrollar este prototipo, utilizamos el patrón de diseño Modelo Vista Controlador o MVC. Con el objetivo de ofrecer una explicación más segmentada de los distintos componentes que forman el prototipo, la dividimos en las secciones a continuación.

Controlador

Como mencionamos en la Sección 4.2, desarrollamos la *app* principalmente utilizando la librería Kivy del lenguaje Python. Esta librería contiene la implementación de distintas clases que representan a la lógica de distintos elementos que puede contener una *app multitouch* como la que buscamos desarrollar, desde botones capaces de ser presionados, hasta las mismas pantallas/interfaces incluidas dentro de la *app*. Estas últimas son definidas gracias a la clase *Screen* que ofrece Kivy.

Así, a partir del principio de *herencia*, desarrollamos distintas clases que heredan la totalidad de las funcionalidades y características ya definidas en la librería. En concreto, las ventanas definidas de esta forma contienen sus propios parámetros y métodos que son capaces de ofrecer la funcionalidad que queremos para cada una de las clases. Esto hace que cada pantalla sea definida como una subclase de *Screen*.

Adicionalmente, la librería también ofrece una clase *ScreenManager* que actúa como el administrador de las distintas pantallas dentro de la *app*. Su objetivo principal es almacenar

las pantallas definidas anteriormente, y usar los métodos que ofrece esta clase para decidir qué pantalla mostrar tras realizar alguna acción, como por ejemplo, apretar un botón.

El resto de las componentes que puedan incluir las pantallas, cómo por ejemplo botones y texto, las definimos de manera similar a las pantallas, utilizando la herencia de las clases *Button* y *Label* respectivamente. Una vez definidos, estos elementos pueden ser inicializados como objetos dentro de las clases de las pantallas.

Por supuesto, al utilizar la *app* necesitaremos conservar y almacenar información a través de la navegación por las distintas pantallas, como por ejemplo los datos del usuario ingresado. Para esto, Kivy también ofrece una clase diseñada específicamente para la conservación de estas variables llamada *EventDispatcher*. A partir de esto, implementamos distintas clases capaces de no solo almacenar información deseada, sino también definir métodos que se conecten con componentes externas de la información. Para este caso, lo utilizamos para realizar llamados a métodos externos que realizan consultas a la base de datos definida en el Capítulo 3 (Sección 3.6).

Debido a que muchos componentes usados en la librería resultan muy dependientes entre sí, todas las clases que definen la lógica de las pantalla y otros elementos de la *app* las definimos dentro de un mismo archivo *main.py*, el cuál podría segmentarse en distintos archivos: uno para representar a las pantallas, otro para los botones, etc...

Vale la pena también mencionar el algoritmo desarrollado por la librería MIP que resuelve el problema de selección de productos. Todas las funciones de la librería utilizadas por el prototipo, y los métodos definidos a partir de estas funciones las definimos en un archivo (*multiknapsackdirect.py*) aparte dentro de una clase Multiknapsack. Así, el archivo principal es capaz de importar esta clase junto a todas las funcionalidades necesarias para resolver el problema.

Modelo

Para poder realizar consultas a la base de datos, optamos por el desarrollo de una clase externa llamada *PostAPI*, la cual contiene distintos métodos para realizar una diversidad de consultas a la base de datos almacenada en Db2. Definimos las consultas aparte con el objetivo de seguir el patrón MVC establecido inicialmente. Así, esta clase crea un objeto *PostAPI* dentro de lógica principal de la *app* que incluye todos los métodos necesarios para realizar consultas a la base datos.

La conexión en sí la realizamos a través del envío de solicitudes *HTTP* hacia una API REST incluida en el servicio Db2 diseñada para conexiones externas. Por motivos de seguridad, como buscamos que la API REST solo sea capaz de recibir consultas desde la *app*, generamos credenciales dentro del mismo servicio que son necesarias para realizar las solicitudes correctamente. Para esto, incluimos estas credenciales dentro de los datos de cabecera de la solicitud *HTTP*.

Dentro de estas solicitudes, debemos incluir un mensaje en formato *json*, el cual debe

contener la consulta SQL que deseamos ejecutar. Para desarrollar estas consultas, los métodos de *PostAPI* utilizan la librería pypika [52] que contiene métodos para desarrollar consultas de forma sencilla, sin la necesidad de tener que desarrollarlas desde cero.

Vista

Si bien las clases representan a las pantallas y los demás elementos de la *app*, estos no definen cómo se verán o cómo los visualizará el usuario. Para esto, debemos programar todo lo correspondiente al *frontend* del sistema. Siendo específico, definimos esto en archivos externos escritos en el lenguaje kv implementado por la misma librería Kivy.

A través de estos archivos, definimos la posición, el tamaño, color, etc... de los elementos definidos por el controlador. Así, cada clase definida anteriormente que corresponda a una pantalla, botón, texto, entre otros, debe tener su propia definición dentro de estos archivos.

El funcionamiento de estos archivos es similar al que podrían tener los archivos CSS de una página web. Adicionalmente, también podemos definir qué métodos ejecutar tras realizar ciertas acciones. Por ejemplo, dentro de una pantalla podemos definir un botón llamado *TestButton*, el cual al ser presionado, llama directamente desde el archivo kv a un método definido en la clase *TestButton* de nuestro archivo principal controlador (*main.py*).

Así, importamos estos archivos directamente por *main.py* para poder ser utilizados por la *app*.

Diagrama

A continuación, mostramos un diagrama con todas las componentes del prototipo y las interacciones que realizan entre ellas.

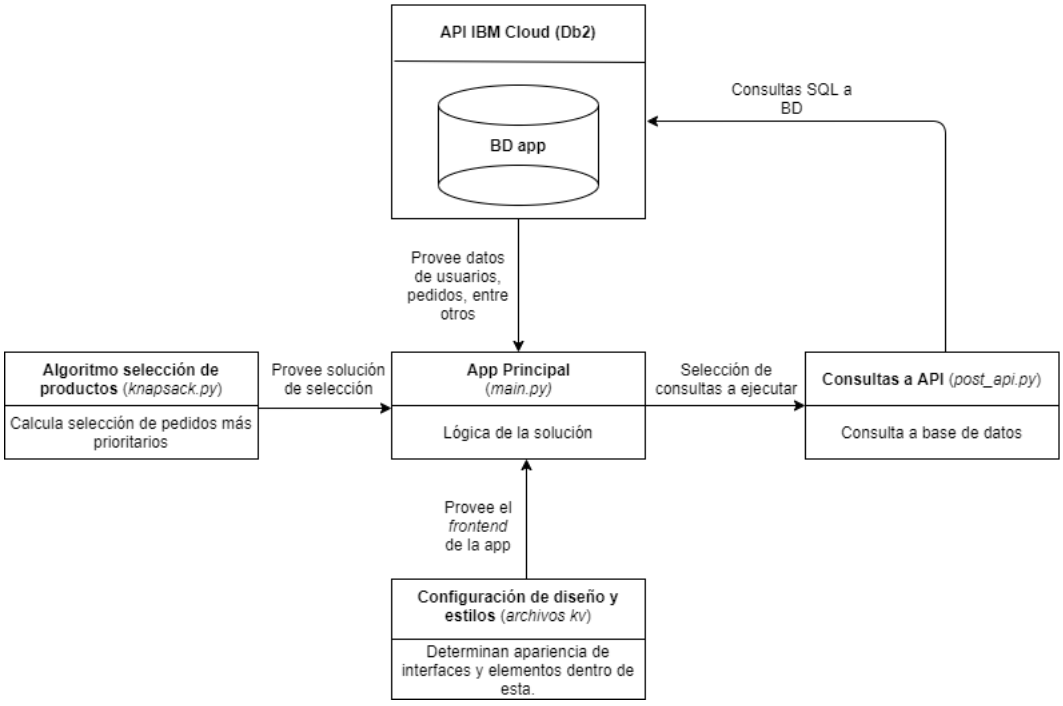


Figura 4.1: Diagrama general de los sistemas relevantes del prototipo, y cómo estos interactúan entre ellos.

Seguido de esto, detallamos cómo funcionan estas componentes para los casos de uso más relevantes. En específico, detallamos los casos descritos en la Sección 3.5.

4.3.2. Registro de usuarios/grupos

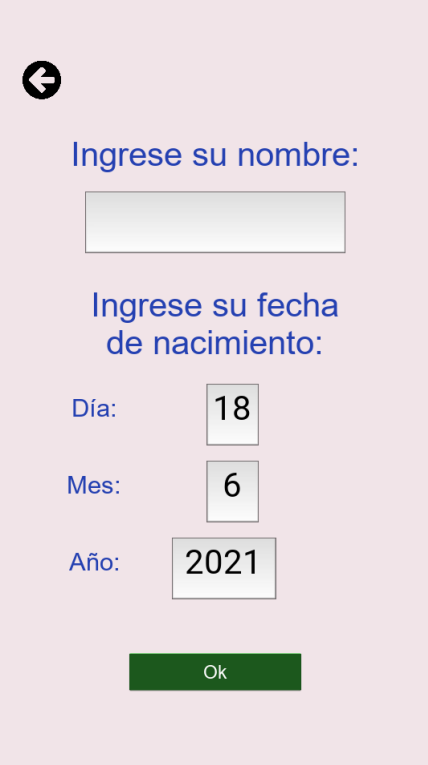
La primera acción que deben realizar los usuarios para usar la *app* correctamente, es registrarse como tal en el sistema. Para poder realizar esto, los usuarios deben inicializar la *app*, donde la primera interfaz que se les muestra corresponde a la de ingreso de usuarios (ver Figura 4.2). Para este caso, deben comenzar con su registro seleccionando el botón verde de *Nuevo Usuario*. En el caso de que estén registrados, simplemente deben ingresar su nombre de usuario y contraseña respectivas en las casillas de texto correspondientes.

Una vez seleccionado el botón, la *app* pregunta por los datos del usuario que se desea registrar. Estos incluyen, además del nombre y la contraseña, fecha de nacimiento y si el usuario posee alguna restricción de movilidad y/o factor de riesgo frente al coronavirus (ver Figura 4.3). Estos últimos son utilizados para determinar la edad del usuario y qué tan vulnerable es, para cuantificar el parámetro de *vulnerabilidad* del usuario dentro del algoritmo de selección de productos.

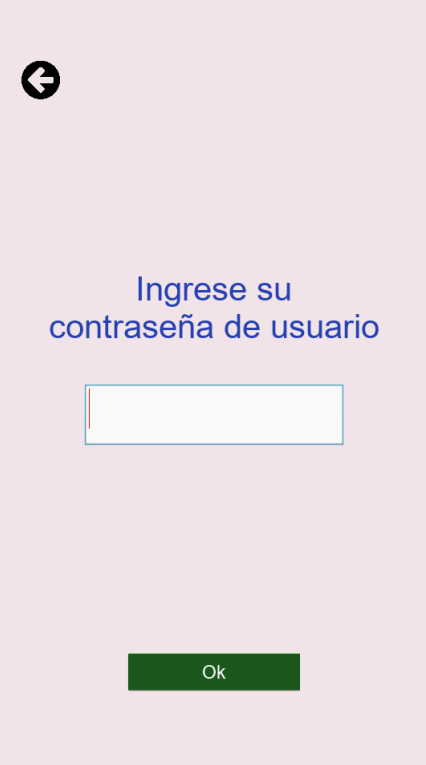
Es importante mencionar que tanto los factores a elegir como las condiciones no corresponden a una lista definitiva, debido a que se busca cubrir todos los tipos de factores y condiciones posibles sin perjudicar la privacidad del usuario al momento de preguntar. Adicionalmente, la lista de factores es la más sujeta a cambios debido a que no se ha confirmado

Figura 4.2: Interfaz de ingreso de usuarios.

con certeza por las autoridades si efectivamente varios de los factores mencionados perjudican a un contagiado por coronavirus, pero se consideran de todas formas.

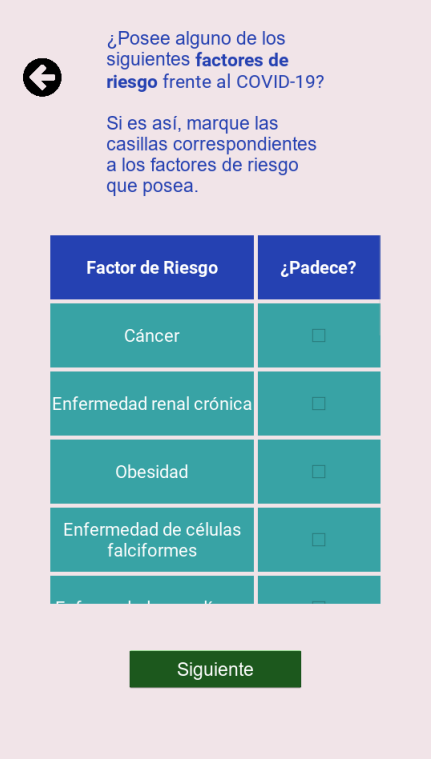


(a) Ingreso de nombre y fecha de nacimiento.



(b) Ingreso de contraseña.

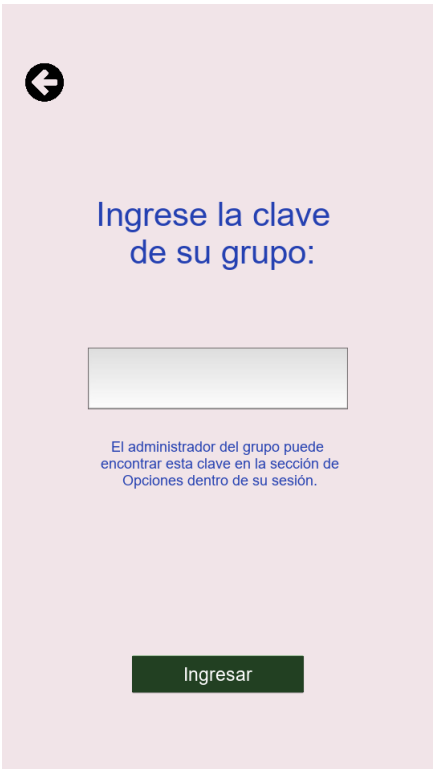
(c) Selector de condiciones.



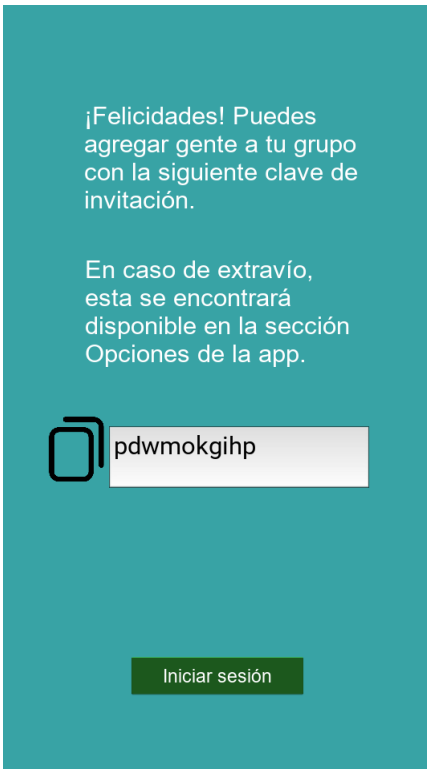
(d) Selector de factores.

Figura 4.3: Ingreso de datos de usuario.

Después de ingresar estos datos, se tienen 2 alternativas. La primera corresponde a formar parte de un grupo ya existente en el sistema, donde se debe ingresar una clave de invitación única para terminar el registro de usuario perteneciente a este grupo. La segunda alternativa es que el usuario pueda crear su propio grupo (volviéndose administrador del grupo), donde además de nombrarlo, se le concederá una clave de invitación la cual puede copiar y enviársela a las potenciales personas que quieran pertenecer a su grupo, para que estos se registren como usuarios nuevos y escojan la alternativa anterior al momento de registrarse. Si por alguna razón esta clave es extraviada por el usuario, este puede acceder a ella a través del menú de *Grupos* dentro de la *app*.



(a) Ingreso de clave de invitación para unirse a grupo existente.



(b) Clave de invitación para acceder a grupo recién creado.

Figura 4.4: Alternativas de registro.

Se decide que cada usuario debe estar asociado al menos y únicamente a un solo grupo por motivos de simplificación. Cada usuario posee parámetros, por ejemplo, el grado de *starving*, que son utilizados para determinar su prioridad al momento de que la *app* seleccione los productos solicitados, las cuales son únicas para el grupo. Esto hace que si el usuario perteneciera a múltiples grupos, tendría más variables asociadas del mismo tipo. Por otro lado, los usuarios actualmente no pueden salir de un grupo y unirse o crear uno nuevo, pero corresponde a una funcionalidad deseada que se podría implementar posteriormente.

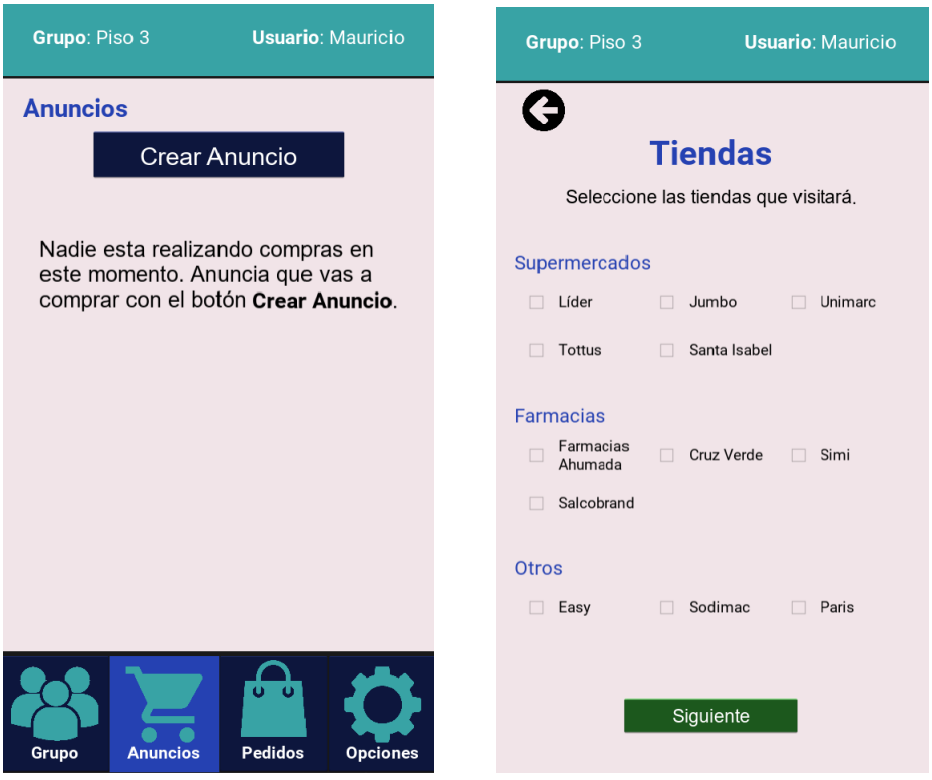
Después de completar el registro, los usuarios pueden acceder libremente a la *app*, a través de su nombre de usuario y contraseña. Sobre los parámetros que poseen que son utilizadas por el algoritmo (ver Sección 3.4):

- Grado de vulnerabilidad $\hat{p_2}$ depende exclusivamente de la edad (calculada a partir de la fecha de nacimiento) y factores/condiciones que ingresaron,
- Grado de participación $\hat{p_3}$ comienza en 0,5, que corresponde al nivel medio.
- Grado de *starving* $\hat{p_4}$ comienza en 0, ya que aún no ha solicitado pedidos.

4.3.3. Anuncio de salidas

Los usuarios que ingresen a la *app* pueden anunciar cuando vayan a salir de su casa a comprar, para así notificar a los demás miembros de su grupo y que estos puedan realizar encargos. Para esto, deben ingresar al menú de *Anuncios* donde se pueden observar los anuncios que realizan los miembros del grupo en el momento, incluyendo el nuestro si estamos realizando uno en ese momento. En el caso contrario, un usuario puede crear su propio anuncio seleccionando el botón *Crear Anuncio* (ver Figura 4.5-(a)). Adicionalmente, si nadie en el grupo está realizando un anuncio, se despliega un mensaje ofreciendo dicha información.

Una vez realizado esto, primero se le pregunta al usuario a qué tiendas específicamente se va a dirigir, a través de un menú selector de tiendas, donde estas se encuentran agrupadas en distintas categorías. Vale la pena recordar que la información de las tiendas seleccionables se encuentra almacenada en la base de datos de la *app*.

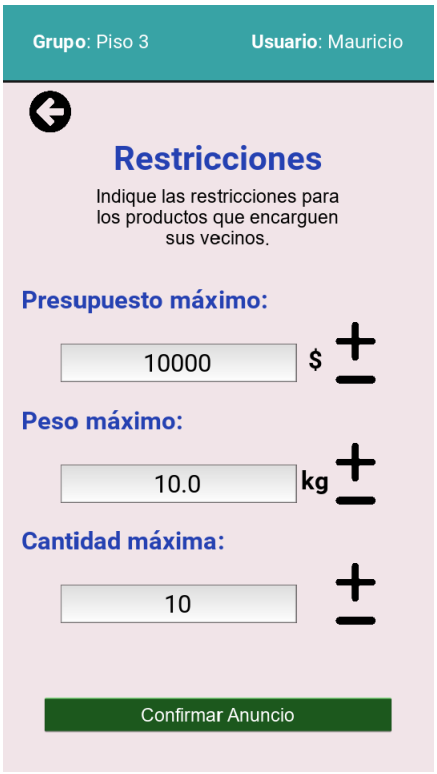


(a) Menú de *Anuncios* sin anuncios de grupo.

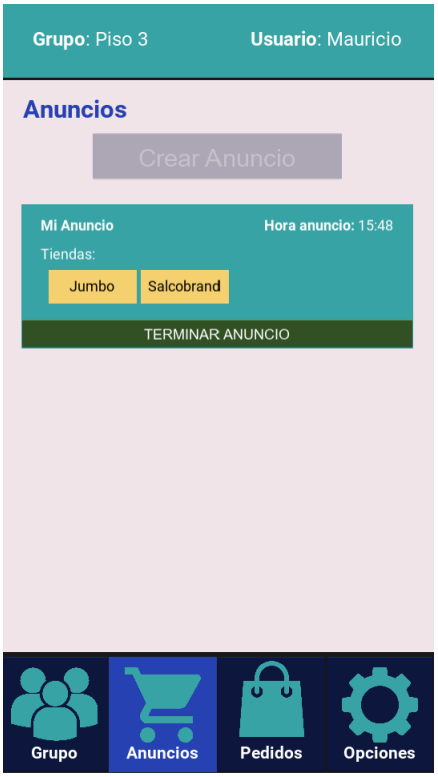
(b) Selección de tiendas.

Figura 4.5: Creación de anuncio (parte 1 de 2).

Seguido de esto, se le pregunta al usuario comprador cuáles son sus restricciones al momento de comprar. Estas corresponden al presupuesto máximo para los productos de sus vecinos, su peso máximo total y la cantidad total de estos. Para facilitar el ingreso de estos datos, los usuarios pueden ingresarlos manualmente o utilizar los botones de signos suma y resta como se puede apreciar en la Figura 4.6-(a). Después de confirmar estos valores, el anuncio es creado y se ingresa a la base de datos como un anuncio "activo". Esto último quiere decir que el resto de los usuarios puede encargar productos al usuario comprador a través de este anuncio, cuyo procedimiento se explica a continuación.



(a) Ingreso de restricciones por parte del comprador.



(b) Menú de *Anuncios* con anuncio recién creado.

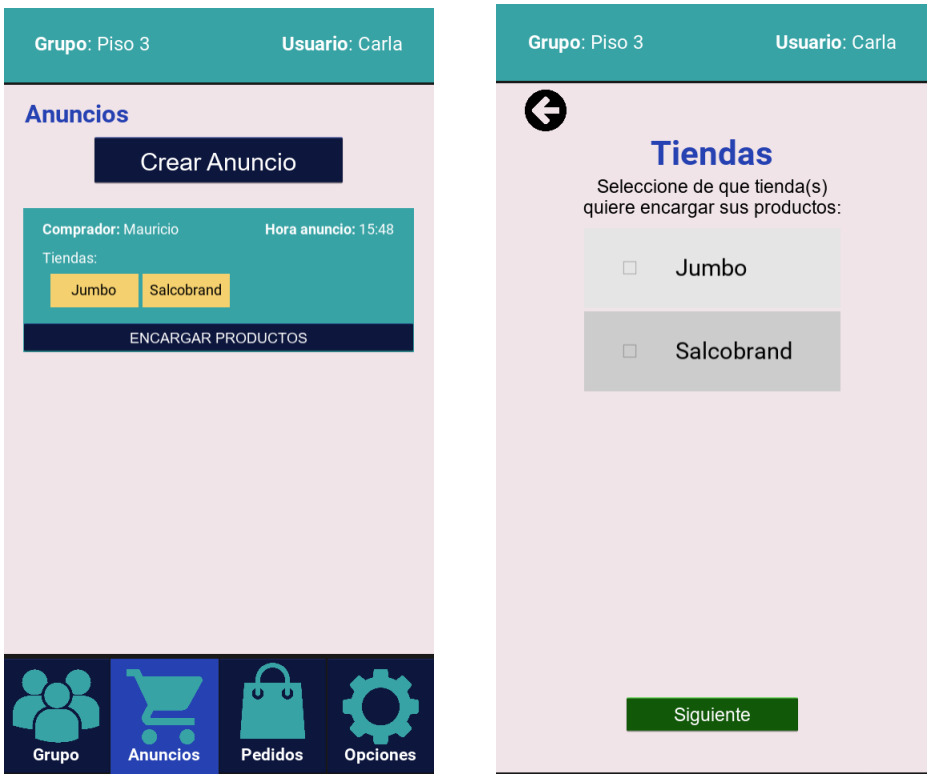
Figura 4.6: Creación de anuncio (parte 2 de 2).

Cuando el usuario comprador llegue a la primera tienda, debe cerrar el anuncio creado. Para esto, simplemente debe seleccionar el botón verde *TERMINAR ANUNCIO* que se encuentra debajo de su anuncio en la Figura 4.6-(b). Así, el comprador no podrá recibir más encargos del resto de los usuarios mientras realice las compras. Esto es conveniente, debido a que en el caso contrario, el comprador debería estar más pendiente de la *app* mientras va comprando, o incluso podría recibir encargos de tiendas que ya visitó, por lo que tendría que volver a estas. Al cerrar el anuncio, la *app* debe seleccionar los productos más prioritarios para comprar, proceso que se explica en la Sección 4.3.5.

4.3.4. Encargo de productos

Al momento de que el comprador anuncia una salida, el resto de los usuarios puede encargarle productos de las tiendas que visitará. Para realizar esto en el menú de *Anuncios* pueden observar los anuncios activos y si quieren encargar, seleccionan el botón azul *ENCARGAR PRODUCTOS* dentro del anuncio de interés (ver Figura 4.7-(a)). Por cada anuncio, se puede observar quién realiza las compras, a qué hora se anunció la salida, y las tiendas a visitar.

Posteriormente, se deben seleccionar las tiendas de donde se desean encargar los productos deseados (ver Figura 4.7-(b)). Por supuesto, la *app* permite seleccionar productos solo de las tiendas que visitará el vecino comprador. Para continuar, solo debe seleccionar el botón *Siguiente*.



(a) Menú de *Anuncios* con 1 anuncio disponible.

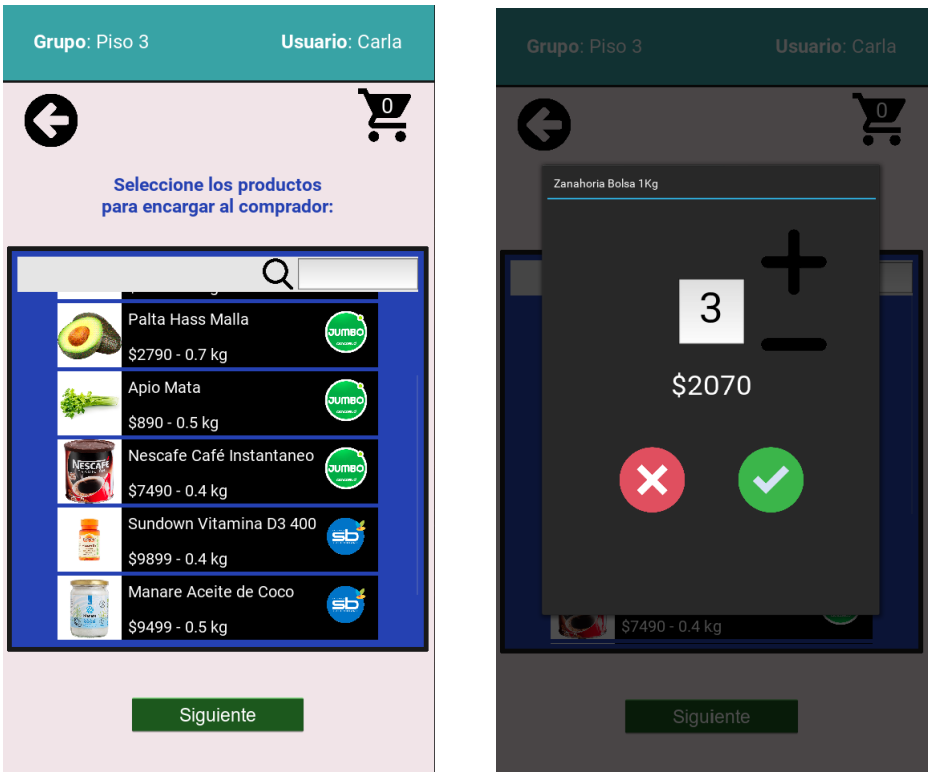
(b) Tiendas disponibles para realizar encargos.

Figura 4.7: Encargo de productos (parte 1 de 2).

En la siguiente interfaz, se despliega una lista de todos los productos disponibles de todas las tiendas seleccionadas. La información de estos productos es directamente extraída de la base de datos de la *app*. Por cada producto, se muestra una imagen, el nombre, precio, cuanto pesa aproximadamente, y un logo representando a la tienda que pertenece. Con el objetivo de facilitar la selección de productos, el usuario puede utilizar el buscador situado en la esquina superior derecha del cuadro conteniendo a los productos seleccionables (ver Figura 4.8-(a)). Puede buscar productos escribiendo tanto el nombre de los productos como el nombre de la tienda a la cual pertenecen.

Cuando el usuario selecciona un producto, se despliega una ventana *popup* preguntando por la cantidad deseada de ese producto, la cual pueden editar presionando los botones de suma y resta posicionados a la derecha (ver Figura 4.8-(b)). Además, despliega el monto total por esa cantidad que cambia dinámicamente acorde a la cantidad seleccionada. Para confirmar el producto, se debe seleccionar el botón verde con símbolo de *check*, o si se quiere cancelar, el botón rojo con una cruz.

El usuario puede revisar los productos a pedir seleccionando el ícono de carro en la esquina superior derecha de la interfaz, donde además puede deshacer cualquier producto de los que haya seleccionado. Una vez que se hayan seleccionado todos los productos deseados, se debe seleccionar el botón *SIGUIENTE*. Así, todos los productos pedidos por el usuario se transforman en nuevos pedidos asociados al anuncio, los cuales se ingresan a la base de datos de la *app*.



(a) Buscador de productos.

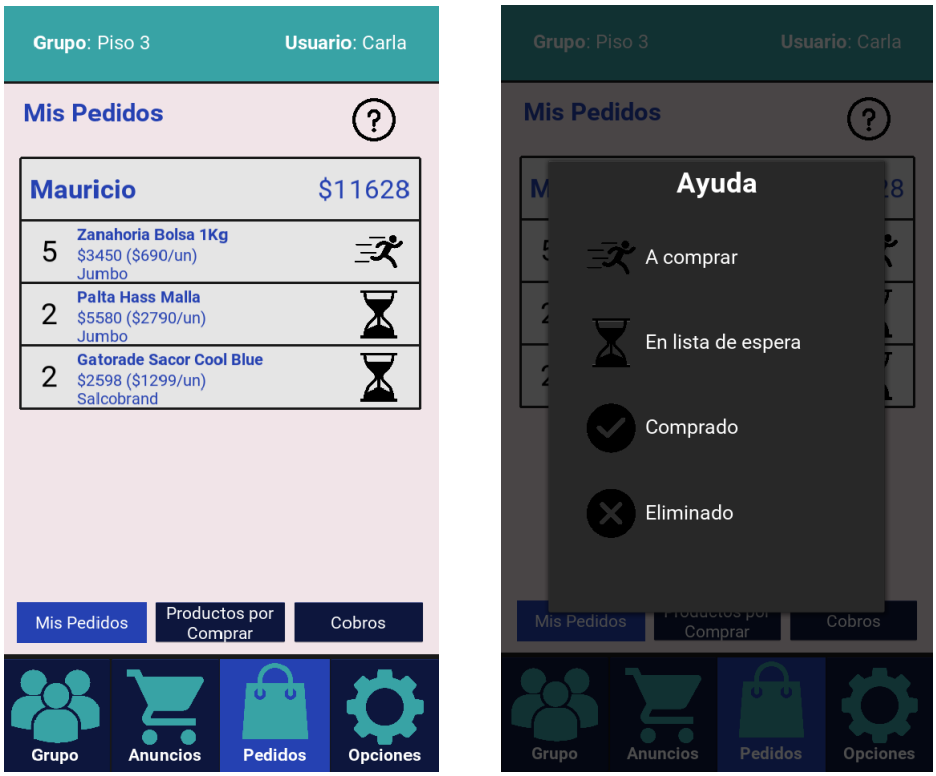
(b) Ingreso del número de unidades del producto seleccionado.

Figura 4.8: Encargo de productos (parte 2 de 2).

Una vez que los usuarios realizan sus pedidos, pueden revisar el estado de estos para chequear si se están yendo a comprar, si se cancelaron, etc... Para esto, se debe dirigir al menú de *Pedidos*, específicamente al submenú *Mis Pedidos* (el cual se despliega por defecto dentro de este menú). Dentro de este, los usuarios pueden revisar el estado de todos los productos que han encargado a los distintos compradores del grupo, además indicando su cantidad, nombre, precio y de qué tienda son. El estado se describe a través del uso de distintos íconos, cuyo significado se puede saber seleccionando el botón de ? en la esquina

superior derecha (ver Figura 4.9-(a)). En particular, cada pedido puede tener 4 estados:

- **A Comprar**: El pedido recién fue creado y/o ya se encuentra dentro de la lista de compras del comprador.
- **En lista de espera**: El producto correspondiente al pedido no fue seleccionado inicialmente por el algoritmo de selección de productos, y se encuentra en lista de espera en caso de que uno de los seleccionados inicialmente se haya cancelado.
- **Comprado**: El producto del pedido fue comprado exitosamente.
- **Eliminado**: El producto no se pudo comprar (por ejemplo, porque fue seleccionado pero no había *stock*, o porque fue descartado por el algoritmo de selección).



(a) Estado actual de pedidos encargados por el usuario.

(b) Leyenda de iconos.

Figura 4.9: Submenú *Mis Pedidos*.

A continuación, explicamos con mayor detalle donde y cuándo funciona el algoritmo de selección de productos.

4.3.5. Chequeo de compras

Volviendo al rol del comprador, para que este pueda confirmar las compras, primero debe terminar su anuncio dentro de la *app* a través del menú de *Anuncios*. Tras esto, se despliega información sobre todos los pedidos registrados en la base de datos que se hicieron a través este anuncio, donde se agrupan por las tiendas que se visitarán (Figura 4.10-(a)). Adicionalmente, abajo de esta lista se despliega el monto, cantidad y peso total de estos pedidos, y si estos valores superan o no las restricciones del comprador (donde cada restricción sale de color rojo si se viola, y verde si se respeta).

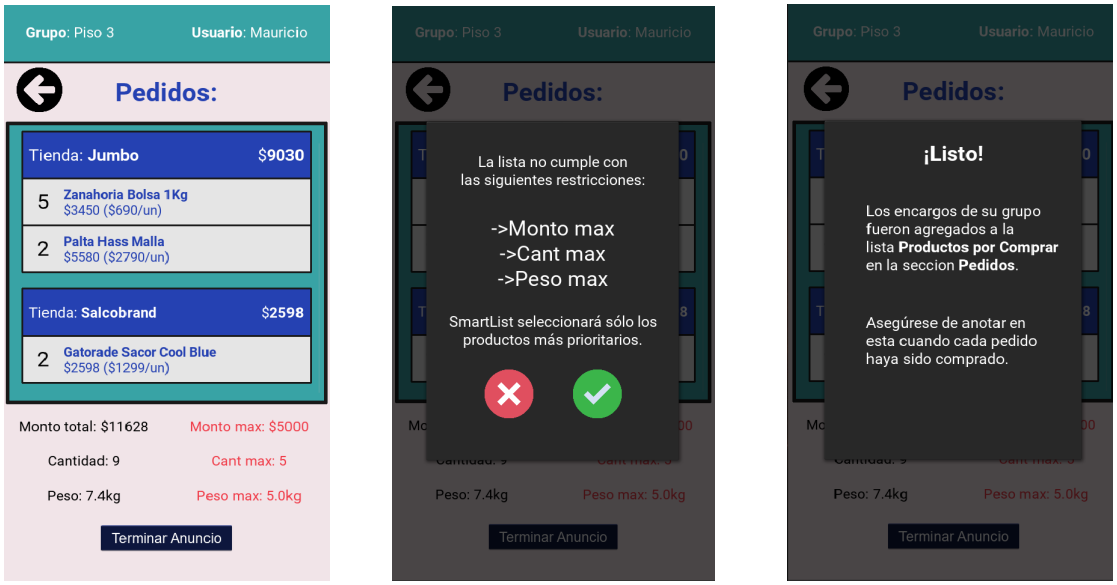
En caso de que efectivamente haya una restricción violada, se le notifica al comprador a través de un mensaje de esto, y advirtiendo que la *app* seleccionará los pedidos más prioritarios (ver Figura 4.10-(b)). Al momento de seleccionar el botón verde de *check*, la *app* ejecuta el algoritmo de selección de productos.

Para efectos prácticos, cada uno de los pedidos que tenga una cantidad mayor a 1 se desglosa en múltiples productos de cantidad 1 antes de introducirlos como posibles items a elegir dentro del algoritmo. Con esto el algoritmo puedes escoger una cantidad menor a la solicitada, en caso de que la cantidad original viole una restricción Así, por ejemplo, si un usuario pide 5 zanahorias y esta cantidad viola una de las restricciones, la *app* puede escoger un número menor para respetar esta restricción (por ejemplo, 3 zanahorias). Para efectos de la implementación, vamos a suponer que todos los productos se venden por unidad, incluyendo los que normalmente se venden por peso (como por ejemplo el pan o verduras).

Con esto, se tiene toda la información necesaria de cada producto para usarlos en el algoritmo. Después de la selección, si existen productos tanto en el grupo de seleccionados como no seleccionados que referencian al mismo producto, estos se englosan. Así, por ejemplo, si un usuario decide pedir 5 zanahorias y la *app* solo escoge 3, entonces habrán 2 pedidos resultantes, un pedido seleccionado de 3 zanahorias y uno no seleccionado de 2.

Una vez realizada la selección, todos los pedidos seleccionados cambian sus variables de estado para que queden "enlistados" para comprar, mientras que los no seleccionados también cambian sus variables para quedar "en espera". Posteriormente, se le recuerda al usuario comprador que puede revisar todos los productos a comprar pertenecientes a los pedidos recién "enlistados" dentro del submenú *Productos por Comprar*.

Es importante destacar que en este punto del proceso de compras, el anuncio como tal aún se encuentra dentro de la base de datos en estado "inactivo", es decir, corresponde a un anuncio que ya fue terminado por el usuario comprador, pero aún no se elimina dentro de la base de datos. Lo anterior se hace con el objetivo de que los pedidos realizados a partir del anuncio conserven una referencia al mismo anuncio. Esto último es necesario para que después se pueda analizar distintas proporciones relevantes, como por ejemplo, cuántos pedidos que solicitó un usuario efectivamente fueron completados para calcular su índice de *starving*.



(a) Lista de pedidos solicitados por grupo.

(b) Mensaje de exceso de restricciones.

(c) Confirmación de selección de productos.

Figura 4.10: Cierre de anuncio y selección de productos.

Mientras el usuario comprador realiza las compras dentro de las tiendas, este puede acceder fácilmente a la información sobre los pedidos seleccionados por la *app* en el submenú *Productos por Comprar*. Aquí los compradores pueden consultar tanto por la cantidad como el precio de los productos encargados agrupados en las distintas tiendas que el comprador va a visitar. Estos valores se encuentran desplegados inicialmente en la segunda y tercera columna de la lista de productos (ver Figura 4.11-(a)). Sin embargo, se puede dar la situación donde el precio de un producto en la tienda difiera del precio original en la base de datos, o que la cantidad que se puede comprar de ese producto sea menor a la solicitada originalmente por razones como falta de *stock* dentro de la tienda. Es por esto que al usuario comprador se le permite editar estos valores dentro de la lista, para que reflejen el precio que realmente tiene que pagar y la cantidad que realmente se compra.

Así, cuando el comprador termina sus compras dentro de una tienda, debe marcar los productos que le encargaron que efectivamente se compraron a través de un *check* en la última columna de la lista, editar los valores si es necesario, y confirmar la selección de productos encontrados "apretando" el botón *TERMINAR COMPRAS EN TIENDA*. Al realizar esta acción, los pedidos asociados a los productos que efectivamente se compraron cambian su estado de "enlistado" a "comprado".

Gracias a esto, se confirma además el precio final y cantidad de cada pedido. Esto es importante porque con esto los usuarios compradores saben exactamente cuánto deben cobrarle a cada usuario, por lo que cada uno de los pedidos "comprados" pasa a ser un cobro desde el punto de vista del comprador, por lo que posteriormente puede revisar estos valores en el submenú *Cobros* (ver Figura 4.11-(c)). Por otro lado, los solicitantes son notificados del cambio del estado (de "Por Comprar" a "Comprado") de sus pedidos gracias al submenú *Mis Pedidos*, donde pueden revisar no solo el precio final de los pedidos, sino también la cantidad

final que se compra por cada producto solicitado.

Si se da el caso de que en una tienda no se logran comprar todos los productos junto a sus cantidades originales, la *app* pregunta al usuario si desea agregar a la lista de compras productos de la lista de espera (ver imagen (b) de la Figura 4.11). Estos corresponde efectivamente a los productos de pedidos que quedaron fuera de la selección inicial realizada por el algoritmo. Así, si el usuario accede a esto, la *app* puede correr el algoritmo una vez más utilizando como restricciones el dinero, peso y espacio remanente a partir de los productos que aún se encuentran dentro de la lista de productos por comprar. También se considera que no se pueden agregar productos de tiendas que ya se visitaron, para que el usuario no tenga que volver a ellas.

(a) Lista de pedidos solicitados por grupo.

(b) Consulta para agregar productos de los pedidos en la lista de espera.

(c) Confirmación de compras terminadas.

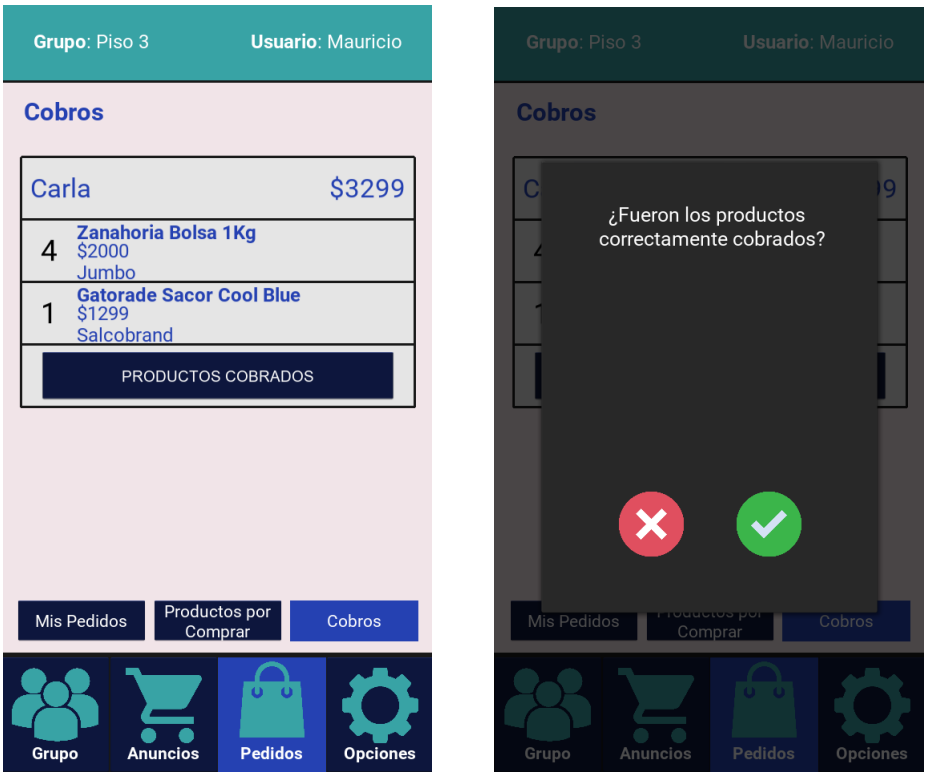
Figura 4.11: Compra de productos encargados por grupo.

4.3.6. Consulta de cobros

Al finalizar con las compras de una tienda en específico, si hubieron productos que se compraron, estos cambian su estado de "enlistado" a "comprado" para poder visualizarse en el submenú de *Cobros*. En particular, estos se visualizan en distintas listas, cada una representando a un usuario en particular, donde se muestra además de los productos a comprar comprados, el precio final de cada uno, y la cantidad total de dinero que se le debe cobrar al usuario (ver Figura 4.12-(a)).

Así, cuando el usuario comprador logre cobrar exitosamente el dinero que ha prestado al usuario que solicitó productos, se debe seleccionar el botón *PRODUCTOS COBRADOS*, lo que despliega un mensaje final para efectivamente confirmar que se realizó la transacción correspondiente (ver Figura 4.12-(b)). Esto hace que los pedidos correspondientes dentro de

la base de datos sean eliminados de manera permanente, logrando así completar el procedimiento de compra.



(a) Lista de cobros a realizar.

(b) Confirmación de cobros.

Figura 4.12: Submenú *Cobros*.

Capítulo 5

Validación y resultados

En este capitulo abordamos el proceso de validación del trabajo realizado. De acuerdo al problema planteado para este trabajo, evaluamos el rendimiento del prototipo desarrollado en distintos aspectos, comparándolo también con la situación actual, a través de distintas evaluaciones. En total se realizaron 3 evaluaciones, para las cuales presentamos el diseño y los resultados obtenidos.

5.1. Usabilidad

Una de las características más deseables de este prototipo, es que pueda ser utilizado por usuarios de todo tipo, incluyendo personas mayores. Es esencial que estos últimos usuarios puedan ocupar la solución con facilidad, debido a que estos usuarios son los que resultan más beneficiados al ocuparla. Por esta razón, una de los requisitos más importantes de este prototipo es que cumpla con estándares mínimos de usabilidad.

Diseño de la evaluación

Una vez que terminamos de implementar el prototipo, evaluamos cómo un usuario promedio interactúa con este a través de pruebas de usuario. Estas corresponden a pruebas que realizamos a potenciales usuarios que podrían ocupar la solución final, es decir, que son parte del público objetivo. Cada una de las pruebas es individual y, para mantener el distanciamiento social, las realizamos a través de reuniones por videollamada.

Las pruebas que diseñamos constan de 3 partes. La primera consiste en mostrarle las interfaces más relevantes del sistema al usuario candidato, para que este pueda familiarizarse con el sistema, donde indagamos por sus pensamientos iniciales al ver la *app* por primera

vez. En las Figuras 5.1 y 5.2 se muestran las interfaces principales que se usaron en esta fase, que corresponden a la principales de la *app*.

A partir de estas figuras, realizamos las siguientes preguntas a los usuarios:

- ¿Cuál cree que es el propósito de esta interfaz/ventana?
- ¿Puede leer satisfactoriamente los textos? ¿Qué es lo que interpreta a partir de estos?
- Si es que los hay, identifique cuáles son los botones disponibles en la interfaz/ventana y explique para qué cree Ud. que sirven.

Posteriormente, le encargamos al usuario ciertas tareas que debe realizar dentro de la *app*, sin indicarle cómo debe hacerlas. Estas tareas corresponden a acciones regulares que deben realizar los usuarios para utilizar la solución final de forma provechosa. En particular, diseñamos 2 versiones distintas de la prueba, cada una con un conjunto distinto de tareas a completar.

En la primera versión, los usuarios deben actuar como un vecino comprador y realizar distintas compras al resto del vecindario. En específico, las tareas que le encargamos son:

- Crear un usuario dentro de la *app* junto a un nuevo grupo de usuarios.
- Anunciar una salida a las tiendas Jumbo, Salcobrand y Unimarc, indicando ciertas restricciones.
- Terminar anuncio y confirmar la compra de productos.

En la otra versión, se realizan acciones bajo el rol del vecino que solicita los productos (a un vecino que sale a comprar por los demás). Estas acciones corresponden a:

- Crear un usuario dentro de la *app* y unirse a un grupo existente.
- Encargar productos a un vecino que sale a comprar.
- Revisar el estado de los productos durante la compra.

Finalmente, después de que los usuarios candidatos interactúan con la *app*, les consultamos por su experiencia general interactuando con el sistema. En concreto, las preguntas más importantes que realizamos corresponden a las que están incluidas en la encuesta *System Usability Scale* (SUS) [53] creada por John Brooke. Este método de evaluación se puede clasificar como *quick and dirty*, es decir, rápido para salir de apuros, pero que resulta efectivo debido a la cantidad pequeña de preguntas que la componen (10 en total).

Esta preguntas corresponden a frases que describen cuánto el usuario efectivamente se familiariza con el sistema, las cuales deben responder con un número de la escala del 1 al 5 (siendo 1 totalmente desacuerdo y 5 totalmente de acuerdo). Para esta ocasión, todas las preguntas fueron traducidas al español por el mismo memorista, debido a la falta de una traducción oficial. A continuación, se encuentran las 10 preguntas/frases utilizadas:

1. Creo que me gustaría utilizar este sistema con frecuencia.
2. Encontré el sistema innecesariamente complejo.
3. Pensé que el sistema era fácil de usar.
4. Creo que necesitaría el apoyo de un técnico para poder utilizar este sistema.
5. Encontré que las diversas funciones de este sistema estaban bien integradas.
6. Pensé que había demasiada inconsistencia en este sistema.
7. Me imagino que la mayoría de la gente aprendería a utilizar este sistema muy rápidamente.
8. Encontré el sistema muy complicado de usar.
9. Me sentí muy seguro usando el sistema.
10. Necesitaba aprender muchas cosas antes de empezar con este sistema.

A las respuestas a estas preguntas se les asigna un puntaje de la siguiente manera:

1. Se suman las respuestas de las frases impares y después se resta 5.
2. Se suman las respuestas de las frases pares y se resta ese total a 25.
3. Se suman ambos resultados y se multiplica por 2,5.

Este resultado final, que va de 0 a 100, entrega el puntaje de usabilidad del sistema. De acuerdo a esta escala, los sistemas con un puntaje sobre 68 son considerados "mejor que el promedio" en términos de usabilidad, por lo que compararemos el promedio de los puntajes obtenidos con respecto a este valor de referencia. Existen otras cotas que otros evaluadores utilizan para clasificar de manera más específica a los sistemas, donde por ejemplo, a partir de 73 el sistema es considerado "bueno", a partir de 85 es "excelente", hasta 100 donde el sistema es el "mejor posible" [54]. Sin embargo, para efectos de esta memoria decidimos ignorarlos debido a qué no corresponden a cotas establecidas por el autor original.

Aparte de estas 10 preguntas, indagamos también por los posibles elementos de la *app* que puedan causar confusión al momento de utilizarla. Para esto, simplemente le preguntamos a los participantes si hubieron ciertos elementos (botones, texto, etc . . .) durante el experimento que les costó entender o directamente no entendieron.

Realizamos un total de 12=7+5 pruebas de usabilidad con usuarios de distintas edades, las cuales separamos en 2 grupos distintos. Los primeros corresponden a los 7 usuarios más jóvenes con una edad entre 20 y 30 años, los cuales realizaron el set de tareas que cumplen el rol de usuario comprador, mientras que los restantes 5 usuarios pertenecen al rango de edad entre 40 y 60 años, los cuales tuvieron que cumplir el set de tareas correspondientes a un usuario solicitante. Decidimos que los más jóvenes son los que deben cumplir con el rol de comprador en este experimento, debido a que estas personas suelen ser las que más salen fuera de casa, por lo tanto generalmente se encuentran en una situación donde tienen mayor oportunidad para ayudar a los usuarios mayores.

Resultados

En la Figura 5.3 mostramos el puntaje obtenido en cada prueba, tras realizar las preguntas sobre la escala de usabilidad de sistemas. Como podemos observar, la gran mayoría de las pruebas logra una puntuación sobre la mínima esperada (línea verde en el gráfico), excepto en 2 ocasiones que obtienen puntajes de 67,5 y 57,5 respectivamente. Ambas puntuaciones pertenecen a pruebas donde los usuarios tuvieron que realizar tareas como compradores. A pesar de esto, logramos un promedio (línea amarilla en el gráfico) de 82,7, que supera en un 14,7 % la puntuación de 68 correspondiente a un sistema "mejor que el promedio".

Adicionalmente, calculamos también el promedio de ambos grupos con el objetivo de corroborar si existen mayores diferencias respecto a la dificultad entre ambos tipos de prueba. Estos corresponden a 78,2 (línea púrpura inferior en el gráfico) para las pruebas realizadas con tareas de comprador, y 89 para las pruebas con tareas de solicitante (línea púrpura superior en el gráfico). Ambos valores logran superar la puntuación de 68.

En cuanto a los elementos que más causaron confusión de acuerdo a las pruebas de usabilidad, se encuentran los siguientes problemas, los cuales nombramos con las variables $\mathbf{P_i}$ para así recordarlas a lo largo del documento:

- $\mathbf{P_1}$ Cuando la *app* termina de seleccionar los productos más prioritarios, no se recibe un *feedback* inmediato sobre lo que se elige y lo que no se elige.
- $\mathbf{P_2}$ No hay una distinción clara entre las interfaces que debería ocupar un usuario comprador y un usuario solicitante.
- $\mathbf{P_3}$ Debería dejarse más en claro el hecho de que existe una lista de espera de pedidos.
- $\mathbf{P_4}$ Sería conveniente desplegar mas información de cada pedido/producto si se "aprieta" dentro de las listas de la *app*.
- $\mathbf{P_5}$ Algunas ventanas *popups* no tienen un botón para cerrarlas (se debe "apretar" afuera de la ventana para que se cierre).
- $\mathbf{P_6}$ En la interfaz de anuncios, se piensa que las cajas que despliegan el nombre de las tiendas a visitar son botones que el usuario debe seleccionar para encargar productos a esa tienda en específico.

5.2. Ahorro de salidas

Para determinar el grado de cumplimiento del objetivo principal de esta memoria (*reducir el número de salidas para comprar productos de primera necesidad*), no podemos realizar un estudio práctico debido a la cantidad de usuarios y tiempo necesarios usando la *app*. Habría que primero encontrar grupos de potenciales usuarios que estén dispuestos a cambiar sus hábitos de compras. Además, requeriríamos un lanzamiento oficial por las distintas plataformas

(Android, iOS), lo que está fuera del alcance de esta memoria, ya que para subir la *app* a estas plataformas implica las creaciones de cuentas y pagos adicionales, en especial en iOS, donde que requiere un costo más elevado [55]. Sin mencionar también asegurar que la solución final debe funcionar de manera correcta en múltiples dispositivos, cuando en esta memoria sólo se utilizaron dispositivos Android para comprobar el funcionamiento correcto del prototipo.

Es por estas razones que desarrollamos un modelo matemático simplificado que, abstrayendo distintos aspectos del sistema, permite estimar la cantidad de salidas a comprar que puede ahorrar cada usuario al utilizar la *app*. En particular, el modelo desarrollado busca estimar, a partir de un conjunto de variables (ver Tabla 5.1), la fracción o porcentaje de salidas que un usuario promedio ahorraría al utilizar la *app*.

Tabla 5.1: Variables utilizadas en modelo matemático.

$\#v$	Número de miembros en el grupo
$\#sal_{/mes}$	Número de salidas mensuales promedio de los miembros del grupo.
α	Fracción de salidas que los miembros del grupo anuncian en la *app*.
β	Fracción de miembros del grupo que hacen algún encargo al publicarse un anuncio.
p_{exito}	Probabilidad de que un encargo se complete exitosamente.

Diseño de la evaluación

Partimos escribiendo las variables de la situación previa al uso de la solución final. Suponemos que existe un grupo de personas que viven juntas en una "comunidad". Con el objetivo de simplificar, diremos que se trata de un vecindario. Definimos la cantidad de vecinos totales como $\#v$. Adicionalmente, definimos la cantidad de salidas promedio mensuales de los miembros del grupo como $\#sal_{/mes}$. Para simplificar el modelo, supondremos que una salida equivale a la compra en una (única) tienda. Así, a partir de las variables $\#v$ y $\#sal_{/mes}$, podemos determinar la *cantidad total de compras mensuales que realiza el grupo* como:

$$\#v \cdot \#sal_{/mes} \tag{5.1}$$

A continuación, describimos la situación utilizando la *app* (suponemos que los vecinos constituyen un "grupo" de la *app*). Para el correcto funcionamiento de la solución final, debe existir un porcentaje del grupo de vecinos que anuncie las salidas que realicen para que los demás miembros del grupo puedan pedir productos. Por supuesto, no todas las salidas realizadas por estas personas van a ser anunciadas dentro de la *app*. Así, introducimos la variable $\alpha \in [0, 1]$ que representa la fracción de salidas que efectivamente se anuncian, con la que podemos determinar *cantidad de anuncios mensuales que se realizan dentro de la app* como:

$$\#v \cdot \#sal_{/mes} \cdot \alpha \tag{5.2}$$

A través de cada anuncio, todos los usuarios del grupo (excepto el vecino "comprador" que hace el anuncio) son capaces de encargar pedidos. Sin embargo, podemos suponer que solo una fracción $\beta \in [0, 1]$ de ellos lo hará. Por otro lado, no todos los pedidos se van a concretar, debido, por ejemplo, a las restricciones del comprador o a la falta de *stock* en tienda de los productos encargados. Es por esto que también introducimos la variable p_{exito}, que representa la probabilidad promedio de que un encargo a través de la *app* se complete exitosamente. Con estas variables, podemos calcular *la cantidad promedio de encargos que se completan a través de un anuncio* como:

$$(\#v - 1) \cdot \beta \cdot p_{exito} \tag{5.3}$$

Finalmente, considerando que un encargo exitoso de un usuario corresponde a una salida a comprar que el usuario se ahorra, es posible estimar el porcentaje de compras que los usuarios del grupo se ahorran al utilizar la solución final. Para obtener este porcentaje de ahorro, primero calculamos todos los encargos exitosos realizados al usar la *app* al mes y luego lo dividimos por la cantidad total de compras que realiza el grupo:

$$\frac{(\#v \cdot \#sal_{/mes} \cdot \alpha) \cdot ((\#v - 1) \cdot \beta \cdot p_{exito})}{\#v \cdot \#sal_{/mes}} \tag{5.4}$$

Observe que las variables $\#sal_{/mes}$ y $\#v$ se encuentran tanto en el numerador como en el denominador de la fracción, por lo que podemos simplificarlas, obteniendo:

$$\alpha \cdot (\#v - 1) \cdot \beta \cdot p_{exito} \tag{5.5}$$

Así, podemos concluir que el porcentaje de salidas ahorradas es independiente de la cantidad de salidas ($\#sal_{/mes}$) que los usuarios realizan al mes y aumenta linealmente con respecto al tamaño del grupo ($\#v$), la voluntad del grupo en publicar sus salidas en la *app* (α), el deseo del grupo de hacer encargos (β) y la factibilidad de satisfacer los encargos (p_{exito}).

Resultados

Ahora aplicamos el modelo desarrollado para obtener una estimación concreta del nivel de ahorro en salidas que induce el uso de la *app*. Para ello, partimos observando que en cuanto más encargos reciba un anuncio ($(\#v - 1) \cdot \beta$), menor será la probabilidad de completarlos exitosamente, debido a las restricciones propias que tiene un usuario comprador (peso total que puede cargar, número total de productos que puede comprar y costo total que puede cancelar). Es por esto que al elegir valores concretos de las variables del modelo, debemos dar cuenta de esta correlación.

Consideraremos un grupo de 8 miembros ($v = 8$), donde el 50 % de salidas que hacen los miembros son anunciadas en la *app* ($\alpha = 0,5$), y al hacerse un anuncio, el 70 % de los miembros hacen un encargo ($\beta = 0,7$). La probabilidad de satisfacer un encargo es en realidad difícil de predecir porque depende significativamente de las restricciones que presenten los usuarios compradores y del tipo de encargos que hagan los restantes miembros del grupo, pero si suponemos que es del 30 % ($p_{exito} = 0,3$) —lo que consideramos una cota segura—, los miembros del grupo se ahorrarán alrededor el 73 % de sus salidas.

Si el tamaño del grupo se reduce a 5 ($v = 5$), podemos suponer que la probabilidad de satisfacer un pedido va a aumentar, por ejemplo, a 35 % ($p_{exito} = 0,35$), y los miembros del grupo se ahorrarán el 49 % de sus salidas. Si además se reduce al 40 % la fracción de miembros que hacen encargos al publicarse un anuncio ($\beta = 0,4$), podemos suponer que la probabilidad de satisfacer un pedido aumentará, por ejemplo, al 40 % ($p_{exito} = 0,4$) y los miembros del grupo se terminarán ahorrando el 32 % de sus salidas.

5.3. Selección de productos

Por último hacemos una evaluación integral del mecanismo de selección de productos automático de la *app*, comprobando que los productos que selecciona estén alineados con los que elegiría un comprador, bajo las mismas circunstancias, siguiendo su criterio personal (de prioridades).

Diseño de la evaluación

El experimento parte recreando un escenario de aplicación de la *app*. A los participantes del experimento les presentaremos una situación hipotética donde deben salir a comprar para sus vecinos y a partir de ciertas restricciones e información sobre sus vecino, deben seleccionar manualmente qué productos comprar, del universo de todos los encargados recibidos (selección *baseline*). Por otro lado, correremos la *app* recreando el escenario, para obtener la selección automática que devolvería la *app*. Finalmente, determinaremos la pertinencia de esta selección automática comparándola con la *baseline* (para simplificar el proceso de comparación, recreamos un escenario que admita una selección óptima única, de acuerdo a los criterios adoptados por la *app*).

En el ejemplo utilizado para el experimento, los participantes deben seleccionar el subconjunto más prioritario dentro del siguiente conjunto de productos:

1. Bidón de 6 litros de agua Cachantun (6Kg, $1700) solicitado por persona saludable de 30 años que sale a comprar frecuentemente por los demás.
2. Botella de 1 litro de Gatorade (1Kg, $1500) solicitada por la misma persona del producto 1.

3. Paquete de cigarrillos Philip Morris (0,1Kg, $3300) solicitado por persona de 70 años que no sale a comprar por los demás debido a una fractura en la cadera.
4. Paquete de remedios Sinasmal (0,1Kg, $6000) solicitado por la misma persona del producto 3.
5. Caja de cereal Milo (0,5Kg, $2650) solicitado por persona de 30 años embarazada, que no sale a comprar por los demás, pero la ultima vez que solicitó productos no recibió ninguno.
6. Tarro de Nescafé Tradición (0,2Kg, $4000) solicitado por la misma persona del producto 5.
7. 1 kilo de pan hallulla (1Kg, $1540) solicitado por persona de 60 años que sale a comprar por los demás a veces, pero la última vez que solicitó recibió la mitad de los productos que pidió.

Las restricciones que presenta el comprador son:

1. Presupuesto máximo: $16000.
2. Peso máximo: 7Kg
3. Cantidad máxima de productos: 5.

Paralelamente, también le preguntamos a los participantes del experimento por las razones de su selección, donde argumentan por qué creen que los productos seleccionados corresponden a los más prioritarios. Esto se hace para indagar más sobre los factores que los potenciales usuarios de la *app* puedan considerar más importantes o relevantes.

Finalmente, para cuantificar qué tan consistente es la selección devuelta por la *app* con respecto a las selecciones entregadas por los participantes del experimento, calculamos para cada una de estas selecciones, el porcentaje de productos que también fueron escogidos por la *app*. Agregaremos los porcentajes obtenidos a través de su promedio, y consideraremos que si este valor está por encima del 75 %, la selección de la *app* se encuentra efectivamente alineada con los pensamientos del usuario promedio.

Resultados

En este experimento participaron un total de 14 personas, las cuales 9 seleccionan un subconjunto que cumple con las restricciones, es decir, que resuelve el problema. Sin embargo, los productos escogidos en específico varían. En la Tabla 5.2 mostramos los productos escogidos por cada uno de los participantes, indicando además si el subconjunto cumple con las restricciones, y si ese es el caso, el porcentaje de productos elegidos que también están dentro de la selección devuelta por la *app*. La selección de la *app* está compuesta por los productos 2, 3, 4, 5 y 7.

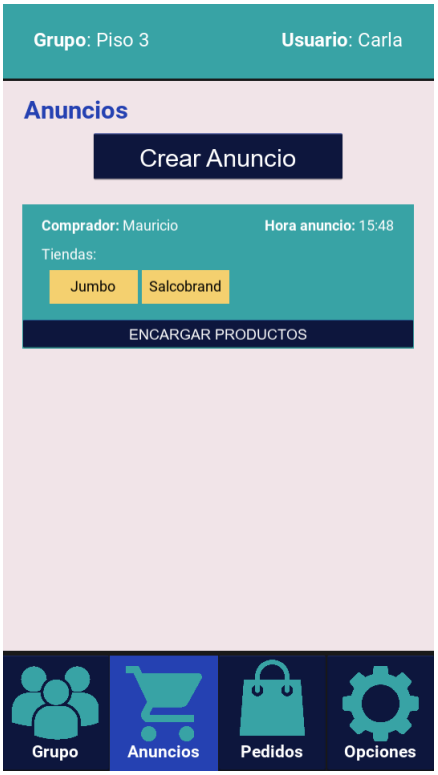
Tabla 5.2: Subconjunto de productos seleccionados por los participantes del experimento. Los productos marcados con una cruz en verde pertenecen a la selección devuelta por la *app*, y los marcados en rojo, no.

								% prod. dentro de selección *app*
1	X	X		X	X		X	Viola restricción de: peso máximo
2			X	X	X	X		60 %
3		X		X	X	X	X	80 %
4		X		X	X	X	X	80 %
5		X		X	X	X	X	80 %
6		X		X	X	X	X	80 %
7	X						X	20 %
8		X		X	X	X	X	80 %
9		X		X	X	X	X	80 %
10	X		X	X	X		X	Viola restricción de: peso máximo
11		X		X	X	X	X	80 %
12	X			X	X	X	X	Viola restricción de: peso máximo
13		X	X	X	X	X	X	Viola restricción de: cantidad máxima y presupuesto máximo
14	X			X	X		X	Viola restricción de: peso máximo

Obtenemos que el promedio de los subconjuntos escogidos es de un 71,1 % , que no logra superar a la cota de 75 % establecida (por un 3,9 %). Notamos además que ninguno de estos coincide exactamente con el subconjunto devuelto por la *app*.

De acuerdo a los comentarios de los participantes, su razonamiento detrás de las selecciones se basa principalmente en la necesidad efectiva del producto (si de verdad es de primera necesidad o no). Algunos también priorizaron la vulnerabilidad de los solicitantes, considerando tanto su edad como sus factores de riesgo y restricciones de movilidad. La gran mayoría tiene en consideración especial a los cigarros, considerándolo como un producto de necesidad mínima y que debería ser el de menor prioridad.

(a) Formulario de inicio de sesión.



(b) Menú de *Anuncios*.

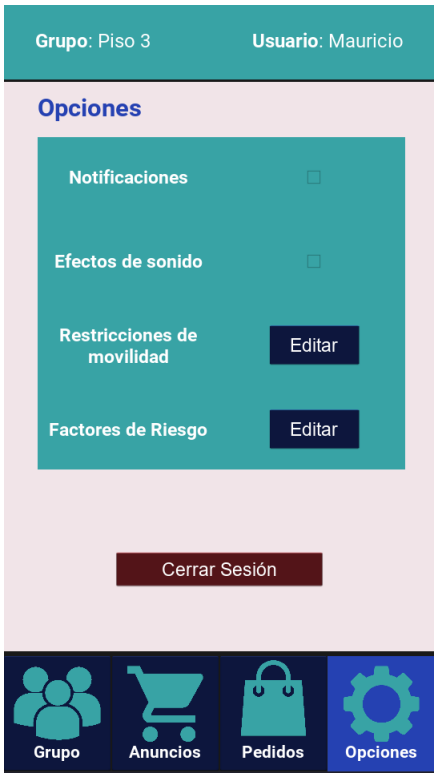
(c) Submenú de *Mis Pedidos*.

(d) Submenú de *Productos por Comprar*.

Figura 5.1: Interfaces utilizadas en la primera parte de la prueba (parte 1 de 2).

(a) Submenú de *Cobros*.

(b) Menú de *Grupo*.



(c) Menú de *Opciones*.

Figura 5.2: Interfaces utilizadas en la primera parte de la prueba (parte 2 de 2).

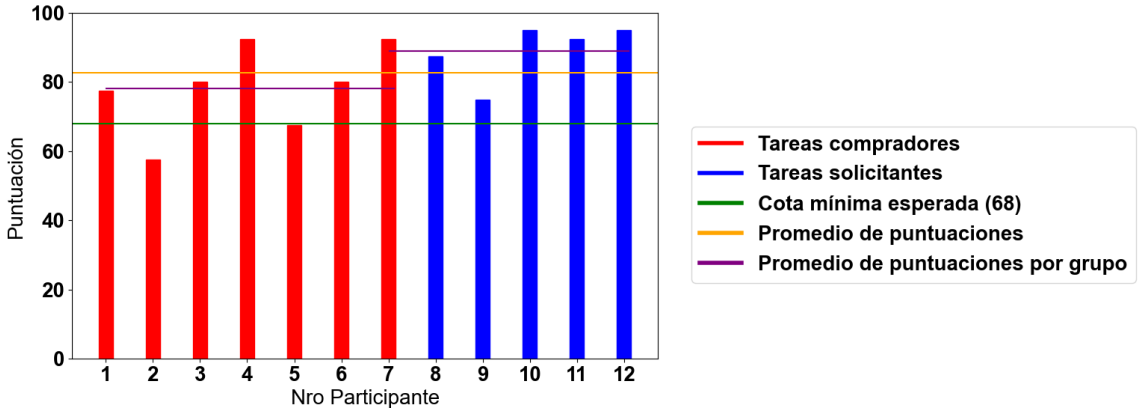


Figura 5.3: Puntuación obtenida por cada uno de los participantes de las pruebas de usuario.

Capítulo 6

Discusión

En este capítulo nos focalizamos en la discusión de los resultados obtenidos en las pruebas descritas en el capítulo anterior. Primero nos enfocamos en el análisis de los resultados donde buscamos explicar la razón de los resultados, para posteriormente proponer posibles mejoras para el prototipo. Adicionalmente consideramos otros conceptos que la solución final debería abordar. Finalmente, también proponemos trabajos a futuro que se podrían realizar para continuar el desarrollo de la solución final.

6.1. Análisis de resultados

6.1.1. Usabilidad

A partir de los resultados obtenidos después de realizar las preguntas relacionadas a la escala de usabilidad de sistemas, podemos concluir que el prototipo cumple con los requisitos mínimos de usabilidad propuestos en la Sección 5.1, obteniendo una puntuación promedio de 82,7 puntos, superando los 68 puntos mínimos propuestos por la misma escala utilizada.

Sin embargo, a pesar de esto, hubieron dos pruebas cuya puntuación no logró superar el mínimo, ambas enfocadas en realizar tareas que debe hacer el usuario comprador. Esto quiere decir que hay una mayor dificultad al usar la *app* para los usuarios compradores que para los solicitantes, lo que también se refleja en el promedio de los puntajes de cada grupo (compradores con 78,2, solicitantes con 88,5).

A pesar de estos resultados positivos, es importante recordar que el sistema utilizado para evaluar la usabilidad (SUS) es *quick and dirty*, lo que conlleva ventajas y desventajas comparándolo con otros métodos de evaluación. La desventaja es que al tratarse de una forma breve para evaluar, se puede clasificar como una evaluación rápida estimativa, lo que no se considera malo para la evaluación de la primera versión de un *software* (como lo es el trabajo realizado en esta memoria). Aún así, recomendamos evaluar las futuras versiones

de la solución presentada en esta memoria utilizando métodos más rigurosos como lo es la evaluación SUMI: *Software Usability Measurement Inventory* [56].

Muchas de las observaciones recibidas a través del *feedback* de los participantes (ver Sección 5.1) corresponden a problemas que no son difíciles de resolver, ya que muchos de estos corresponden a cambios que podemos realizar en la componente *frontend* del prototipo. A continuación describimos alternativas para resolver estos problemas $\mathbf{P_i}$:

$\mathbf{P_1}$ Cuando la *app* termina de seleccionar los pedidos, podría desplegar una ventana *popup* que muestre los pedidos que se eligieron y los que no (y mencionar que estos últimos estarán en la lista de espera).

$\mathbf{P_2}$ Para marcar mejor la distinción entre los roles, todos los submenús relacionados al rol del comprador podrían agruparse en un solo menú. Lo mismo para el rol de solicitante.

$\mathbf{P_3}$ Cuando se seleccionan los productos, recordar a los usuarios involucrados sobre la existencia de la lista de espera.

$\mathbf{P_4}$ Para mostrar más información en las listas, al seleccionar un pedido podría desplegarse una ventana *popup* con información mas detallada sobre este.

$\mathbf{P_5}$ Todas las ventanas *popups* deberían tener un botón de cierre en la esquina superior izquierda para que estas puedan ser cerradas.

$\mathbf{P_6}$ En los anuncios de la *app*, podrían mostrarse las tiendas de cada anuncio con letras destacadas y no dentro de un rectángulo para que los usuarios no piensen que se trata de un botón.

El más complejo de todos corresponde a marcar una diferencia entre los roles del comprador y el solicitante ($\mathbf{P_2}$), ya que esto requeriría la implementación de nuevas interfaces, pero a la vez también podríamos reutilizar la mayoría de las que ya fueron implementadas. A continuación, explicamos cómo podemos realizar esto de forma mas detallada:

En vez de tener un menú dedicado a los anuncios que realiza el grupo dentro de la *app* y uno dedicado a los productos en sí, una alternativa para saber qué menús utilizar corresponde a tener un menú dedicado a las acciones del comprador y otro a las del solicitante.

En el menú de comprador se podría ver los detalles de los anuncios que pertenecen al usuario, los productos que debe comprar y los cobros que debe realizar, mientras que la interfaz del solicitante solo puede ver los anuncios que realizan los otros miembros del grupo, y revisar el estado de los productos que este mismo solicita a los distintos compradores. La estructura de las interfaces en sí no cambiaría mucho a las originales, pero sí el orden y el lugar donde las interfaces se presentan.

Respecto al desarrollo de estas mejoras, al momento de terminar este trabajo no las hemos podido implementar por cuestiones de tiempo. Es por esto que estos cambios a realizar quedan pendientes como trabajo futuro.

6.1.2. Ahorro de salidas

A partir de los resultados que obtenemos del modelo (ver Sección 5.2), podemos sacar varias conclusiones. Primero que nada, el porcentaje de salidas que pueden ahorrar los usuarios al usar la solución aumenta cuando el tamaño del grupo crece. Esto quiere decir que mientras más grande el grupo, más porcentaje de salidas se ahorran.

Tomando el ejemplo del grupo de 8 miembros ($v = 8$), donde estimamos un ahorro en salidas del 73 %: si cada uno de los miembros realizan aproximadamente 20 salidas al mes para comprar productos de primera necesidad, gracias a esta solución pueden ahorrar potencialmente 15 salidas al mes, lo que además de ser un ahorro en términos económicos, se traduce también en 15 situaciones evitadas donde los usuarios se pudieron haber contagiado.

No obstante, es importante destacar que si alguna de las variables se vuelve muy pequeña (si no existiera voluntad por publicar, deseo por realizar encargos, o no se logra concretar ningún encargo), entonces el ahorro el ahorro de salidas también será despreciable, y lo que a su vez desalentará la adherencia de los usuarios a la *app*.

Adicionalmente, debido a a la complejidad del sistema que estamos modelando, no podemos saber cómo la variación de una variable afecta al resto de las variables (por ejemplo, cómo disminuye la probabilidad de concretar un encargo a medida que aumentan los encargos que recibe un pedido), por lo que nos basamos en suposiciones que consideramos razonables. Es por esto último que se vuelve esencial estudiar en el futuro (de manera práctica) cómo la variación de estas variables afecta al número total de salidas ahorradas. Aún así, podemos estimar gracias a este modelo que mientras exista apoyo y disposición dentro de un grupo, es posible disminuir la cantidad de salidas gracias a esta solución.

6.1.3. Selección de productos

A partir de los resultados de las encuestas mostradas en la Tabla 5.2, podemos concluir lo siguiente. Primero que nada, el promedio de similitud (71,1 % de similitud) de los subconjuntos que cumplen las restricciones no logra superar la cota establecida (75 % de similitud). A pesar de que la mayoría de los subconjuntos correctos son de un 80 %, una de los participantes elige un subconjunto con sólo un 20 % debido a que este participante escoge 2 productos cuando la mayoría de los participantes escoge entre 4 y 5 productos. Si excluimos este caso extremo, el promedio que se obtiene es de un 77,5 %, logrando así superar la cota. No obstante, ninguno de los subconjuntos escogidos por los usuarios es exactamente igual al escogido por la *app*.

Los subconjuntos que llegan a parecerse más contienen un 80 % de similitud con el subconjunto escogido por la *app*. Estos subconjuntos contienen los siguiente productos: una botella de Gatorade, paquete de Sinasmal, caja de cereal Milo, tarro de Nescafé, y 1Kg de pan hallulla. El subconjunto escogido por la *app* contiene todos estos productos, excepto el tarro de café que lo reemplaza por el paquete de cigarros Philip Morris. De acuerdo a los comentarios hechos por los entrevistados, este producto en específico lo toman como una excepción, debi-

do a que reconocen que este no se trata de un producto de primera necesidad, considerando que el café es más importante. La razón por la cual el algoritmo de selección de la *app* prioriza más los cigarrillos es porque toma como prioridad a la persona que está pidiendo, ya que esta persona se trata de una señora de edad avanzada que no puede salir de su casa, por lo que a pesar de que no puede ayudar a los demás, el algoritmo considera que es prioritario ayudarla gracias a sus condiciones. En cambio, la persona que pide el café es un usuario que si bien es una embarazada de 30 años, el algoritmo considera que es una condición menos grave que la de la señora de 70, y tampoco ha ayudado al resto del vecindario, por lo que se le penaliza por su baja participación.

A pesar de todo esto, la mayoría de los usuarios prefieren comprar el café por sobre los cigarros. Esto, junto al resto de los resultados, evidencia una clara preocupación por la necesidad del producto como uno de los factores más importantes al momento de elegir los pedidos más prioritarios, por lo que la ponderación original ($\hat{k_1}$) de este factor debería ser mayor respecto a la que se le da para este ejemplo, que corresponde a 0,1 (en comparación, el factor de ponderación $\hat{k_3}$ asociado a la participación de un vecino es de 0,4). Otra forma de resolver esto corresponde a modificar el algoritmo para que deje a los pedidos de productos que no son de primera necesidad con la mínima prioridad posible, para que así el algoritmo seleccione primero a los productos de primera necesidad, sin importar quién los pida.

Otra conclusión importante de esta evaluación es el efecto del error humano que puede ocurrir en una situación sin la solución. De los encuestados, 5 de 14 no logran seleccionar un subconjunto que cumpla con todas las restricciones, lo que corresponde a un 35,7 %. Esto quiere decir que seleccionar productos con el objetivo de que se cumplan todas las restricciones no es una tarea fácil, lo que puede llevar a malentendidos con el resto del grupo si por ejemplo, el comprador promete la compra de un producto cuando en realidad no tiene el dinero suficiente para comprarlo. La *app*, al seleccionar un subconjunto a través del algoritmo, logra que se cumplan todas las restricciones del comprador y pueda comprar efectivamente todo lo que selecciona sin problemas.

6.2. Conceptos a considerar

En la siguiente sección discutimos sobre posibles conceptos que la solución final podría abordar, con el objetivo de mejorar las instancias de "colaboración" entre los usuarios al utilizar esta solución.

6.2.1. Confianza

La gran mayoría de los sistemas y *apps* de colaboración ya establecidas en el mercado (como, por ejemplo, Karpool) poseen métodos para evaluar la confianza de cada uno de los participantes al momento de interactuar con los demás. Esto sirve para evaluar qué tan bueno el usuario realizó el trabajo asignado, como por ejemplo en Karpool los usuarios pueden evaluar qué tan seguro y eficiente conduce el conductor de un vehículo.

Generalmente esta evaluación se puede realizar por sistemas simples de utilizar, como por ejemplo la evaluación usando X de 5 estrellas. Estos sistemas, además de asignar un puntaje promedio a los usuarios de un sistema a partir de las evaluaciones de los demás, permiten que los demás usuarios puedan enterarse de estos puntajes y saber qué tan confiable pueden ser. Adicionalmente, existen sistemas de confianza más elaborados que se están desarrollando hoy en día [57] cuyo objetivo es mejorar la colaboración entre los usuarios, y que evalúan el trabajo que realiza cada uno de los usuarios de una manera más específica y acertada.

Dicho esto, desarrollar un sistema de confianza para una versión futura de esta memoria puede contribuir de manera significativa a la interacción entre usuarios. En particular, se podrían usar métodos simples como la evaluación de estrellas para evaluar a los usuarios compradores al momento que ellos entregan los pedidos a los demás. Además, este puntaje podría afectar la prioridad que se usa actualmente en el sistema de prioridad de selección de productos, para que así los usuarios compradores con un mayor puntaje de confianza resulten más beneficiados. Por su puesto, los usuarios también deberían ser capaces de incluir un *feedback* o un comentario escrito para explicar porque colocan cierto puntaje a tal comprador.

Por otro lado, estos sistemas de evaluación si bien resultan ser esenciales para sistemas de colaboración a gran escala, para ciertos grupos de personas puede resultar redundante tener un sistema así. Esto es debido a que los usuarios objetivos de esta solución corresponden a grupos de personas de tamaños no tan masivos al tratarse de vecindarios o incluso grupos familiares. En varias ocasiones, estos usuarios ya se conocen entre sí, a menos que se trate de un grupo de vecinos que no tenga tanta cercanía entre sí.

Aún así, resultaría muy beneficioso que la solución final contenga un sistema de confianza que mantenga los niveles de usabilidad de la *app*, y estudiar si efectivamente aumenta la participación de los usuarios dentro de un grupo. Adicionalmente, debido a que ciertos grupos podrían encontrar esta evaluación como redundante, el administrador del grupo podría habilitar o deshabilitar este sistema de confianza.

6.2.2. Consciencia

Como el sistema desarrollado en esta memoria se trata de un sistema colaborativo, es importante analizar efectivamente qué tanta consciencia o *awareness* existe de parte de los usuarios respecto a los demás miembros de su grupo. Si bien en grupos pequeños o familiares los usuarios sabrán las características y/o el perfil de los demás, en grupos más grandes como vecindarios esto puede que no ocurra.

Saber más de la situación de los demás usuarios podría motivar a los usuarios que salen más de su hogar a comprar a utilizar la *app* para que así sepan si de verdad existen usuarios con mayor necesidad y restricciones. El gran problema de esto, es que puede que los usuarios no quieran mostrar esta información a los demás, y es por esto que desde el principio de esta memoria se decide dejar esta información como oculta para los demás.

Una alternativa más simple para apelar a la consciencia de los compradores podría ser el uso de notificaciones automáticas recurrentes que les recuerden a los usuarios que han

comprado últimamente que anuncien sus salidas dentro de la *app*. La dificultad sería en saber *cuándo* los usuarios deberían recibir esta notificación. Una opción podría ser que la misma *app* memorice las horas a las que los usuarios normalmente salen a comprar, para que así puedan recibir la notificación unos minutos antes de su hora habitual de compra.

6.2.3. Coordinación

Para que los usuarios sean capaces de encargarle productos a los compradores, se debe analizar también la coordinación que existe entre los solicitantes y los compradores. Si no hay coordinación, los solicitantes no sabrán cuándo deberán encargar sus productos, o puede que los compradores no anuncien sus compras a tiempo.

En nuestro prototipo desarrollado, la decisión de hacer que los compradores anuncien las tiendas a las que se dirigirán justo antes de salir de su hogar corresponde al caso más simple que pensamos al momento de trabajar en la memoria. Sin embargo, puede que las tiendas a las que el comprador se dirija no se encuentren a una distancia muy lejana, lo que hará que se demore menos tiempo en llegar a la primera tienda, y así el resto de los usuarios no tengan mucho tiempo para poder encargar sus productos.

Gracias a esto, comprobar que exista una buena coordinación entre solicitantes y compradores resulta esencial para una versión más avanzada de la solución trabajada en esta memoria. Como alternativa, quizás los usuarios compradores pueden anunciar exactamente a qué hora llegarán a la primera tienda desde antes de salir, a través de una alarma o temporizador, para que así el resto de los usuarios se entere, con suficiente antelación, de que alguien más saldrá a comprar y así poder solicitar los productos que necesita sin tener que salir él mismo a comprarlos por su cuenta.

6.3. Limitaciones y trabajos futuros

Es importante recalcar que el trabajo desarrollado en esta memoria corresponde a un prototipo para una posible solución final que efectivamente sea capaz de reducir el número de salidas para comprar productos de primera necesidad. Debido a esto, además de analizar los tres conceptos mencionados en la sección anterior (confianza, consciencia y coordinación), mencionamos cuatro trabajos prácticos a realizar en la versión final de este trabajo, para que así esta solución pueda ser completada y así pueda ser utilizada por un conjunto de usuarios.

6.3.1. Extracción de datos sobre tiendas y productos

Para efectos de la memoria, los datos de los productos dentro de las distintas tiendas son ingresados de forma manual a la base de datos. Como cada una de las tiendas contiene aproximadamente más de mil productos distintos con datos que van variando constantemente como los precios, el ingreso manual de datos se vuelve inviable. Gracias a esto, uno de los

objetivos principales del trabajo a futuro se vuelve el de encontrar una manera de automatizar la extracción de estos datos desde los sitios *web* de estas tiendas.

Para lograr esto, una alternativa es el desarrollo de *web scrappers* que permitan extraer la información de cada una de las tiendas. Por supuesto, para lograr esto tendría que realizar bajo el consentimiento de cada una de las tiendas que deseen involucrarse con la *app*, por lo que se vuelve un trabajo de largo plazo. La gran desventaja de este método, es que generalmente el *scrapping* es considerado como un método controversial [58], debido a que muchas tiendas proveedoras de productos de primera necesidad no deseen que esta información se exponga a través de otro sitio (dependerá de los términos de uso de esta información).

Otra forma para resolver la recolección de datos, es que los mismos usuarios compradores sean capaces de actualizar esta información de los productos al momento de comprar. Para este método, todos los productos disponibles a través de la *app* deben incluir un precio inicial (que pueden ser los precios disponibles al momento de implementarse este método), los cuales van a ir cambiando a través del tiempo. Así, cuando uno de los precios cambie, un usuario comprador puede actualizar los productos a precios nuevos (siempre y cuando no se trate de una oferta temporal) al momento de comprar estos productos.

Adicionalmente, como esperamos que los datos de cada uno de los sitios se estructure de manera distinta, el almacenamiento de datos en lenguaje SQL, por lo que debería analizarse la posibilidad de guardar estos datos en una base NOSQL, debido a la flexibilidad que ofrece este lenguaje.

6.3.2. Encriptación de datos

Otros datos de la solución que deberían tener un almacenamiento más seguro corresponde a los datos de los usuarios de la *app*. Estos ingresan a la base datos información que pueden considerar como privada, por lo que se vuelve crucial comprobar qué tan seguro es la encriptación de datos que ofrece el almacenamiento actual de IBM Cloud, y en caso de que no sea suficiente, encontrar una alternativa más segura.

6.3.3. Portabilidad/Sistema de notificación

Debido a que el trabajo de esta memoria se concentra en el desarrollo de un prototipo de la *app*, no alcanzamos a analizar el comportamiento que esta puede presentar dentro de distintos sistemas operativos, en especial los móviles son los sistemas a los que apunta esta solución. Esto es debido a que el testing de la solución lo realizamos en forma de *app* de escritorio y en un dispositivo Samsung Galaxy S3 Mini, por lo tanto aún deben realizarse pruebas cuando esta se porta como una *app* móvil en otros dispositivos.

Al realizar estas pruebas, es posible implementar un sistema de notificaciones que notifique a los usuarios de acciones importantes que ocurran dentro de la *app*, como por ejemplo la creación de un nuevo anuncio, cuando un pedido es aceptado, entre otras. Esta *feature* creemos

que es fundamental para mejorar la usabilidad de la solución final, debido a que en el caso contrario, los usuarios deben entrar a la *app* constantemente para enterarse de estas acciones.

Sobre la portabilidad en sí, comprobamos que es posible portar una *app* de Kivy a Android. Siendo más específico, utilizamos una *app* de ejemplo (una calculadora móvil), la cual logramos empaquetar exitosamente [59] a un archivo *.apk*. El funcionamiento de esta también lo comprobamos a través del emulador de Android Studio que permite recrear un celular Android usando una máquina virtual. Este entorno resultaría ideal para probar el comportamiento de la *app* en distintos celulares, y también implementar el sistema de notificaciones.

6.3.4. Reemplazos y Ofertas

Al utilizar la *app*, al momento en que los compradores llegan a la tienda, puede darse el caso de que los compradores no encuentren ciertos productos solicitados por los usuarios, debido a falta de *stock*, o porque la sucursal de esa tienda en específico no tiene esos productos. Para estos casos, el prototipo actual simplemente descarta estos productos e intenta buscar un reemplazo dentro de los otros productos solicitados por los demás, y ver si este reemplazo no viola los límites del comprados.

Sin embargo, puede darse el caso de que existan productos en la misma tienda que pueden ser candidatos de reemplazo que contengan características similares al producto originalmente solicitado (por ejemplo, no se encuentra disponible la botella de Cocacola Light de 3L, pero si la de Cocacola Zero de 3L). Lo que suelen hacer sistemas existentes como Cornershop en estos casos, es que los compradores mandan a los usuarios distintas opciones de reemplazo que se encuentran disponibles en ese momento, y así los usuarios deben elegir si desean alguno de los reemplazos, o simplemente no eligen ninguno. Para nuestra solución, podríamos resolver esto mediante un sistema de reemplazo, donde el comprador y solicitante se puedan comunicar en un canal, y así el comprador es libre de editar la lista para cambiar el producto. Sin embargo, esto hace que el solicitante deba estar atento en caso de que no se encuentran sus productos solicitados, y puede que esta nueva posibilidad de interacción comprometa la usabilidad de la solución, el cual es uno de los elementos más importantes a mantener.

También puede darse el caso de que ciertos productos solicitados se encuentren en oferta, lo que puede hacer que los solicitantes quieran más de ese producto. Otro caso extremo que se podría dar, sería que el solicitante deseara un producto sobre alguno de los que solicita originalmente sólo si el deseado se encuentra en oferta. Así, los solicitantes podrían también tener una lista de productos "deseados" a comprar en el caso de que se encuentren en oferta. Sin embargo, creemos que esto puede aumentar bastante la complejidad de uso al momento de solicitar productos, por lo que no cubrimos estos casos al momento de desarrollar este prototipo.

Así, creemos que es posible implementar un reemplazo de productos al momento de comprar que sean de calidad y precio similar, pero se debe ser cuidadoso con no comprometer la usabilidad de la solución. Para el caso de las ofertas sin embargo, creemos que dándole más opciones a los solicitantes terminaremos aumentando la complejidad de uso, por lo que estos casos el comprador simplemente edita el pedido para que tenga el precio reducido, y la *app*

así puede revisar si es capaz de agregar pedidos de la lista de espera, que sigan respetando las restricciones del comprador.

Capítulo 7

Conclusión

Motivados por la pandemia COVID-19 y los potenciales riesgos que cada salida de casa representa, en este trabajo de memoria desarrollamos el prototipo de una *app* que funciona como una herramienta de coordinación para realizar compras de primera necesidad, de manera colaborativa, entre los distintos miembros de una comunidad, con el objetivo de reducir el número de salidas a comprar que estas personas deberían hacer.

Para ayudar a las personas que deseen comprar productos para su comunidad, utilizamos un algoritmo que, a partir de un modelo de optimización lineal entera, elige los productos que efectivamente son los más importantes a comprar a partir de las restricciones del comprador, para que este no tenga que seleccionarnos manualmente. Este algoritmo a su vez, además de considerar la necesidad de los productos solicitados, toma en consideración las distintas condiciones en la que se pueden encontrar los miembros de la comunidad. Por ejemplo, si tienen restricciones de movilidad, cuanta colaboración han realizado dentro del grupo, entre otros.

Con el propósito de que la *app* pueda ser utilizada por cualquier persona, realizamos pruebas de usabilidad antes de comenzar con el desarrollo del prototipo para comprobar cómo debe estar estructurado. Para ello desarrollamos *mockups* que buscan representar las distintas interfaces de la *app*, los cuales son posteriormente validados gracias a estas pruebas, comenzando así la implementación del prototipo.

Para almacenar los datos a utilizar, utilizamos una base de datos *cloud* ofrecida por IBM Cloud, la cual se muestra útil y capaz de recibir de manera eficiente las consultas necesarias al utilizar el prototipo. Por su parte, para resolver el problema de optimización que selecciona los encargos de los vecinos que efectivamente serán comprados, usamos la librería MIP del lenguaje Python, la cual no solo resuelve el problema de forma correcta, sino que también lo resuelve en un tiempo aceptable para poder usarla dentro de la *app*.

El desarrollo de la *app* lo realizamos utilizando la librería Kivy de Python, la cual ofrece herramientas útiles para el desarrollo de aplicaciones *multitouch*, como por ejemplo, el desarrollo de *apps* móviles, que es precisamente lo que busca realizar esta memoria.

Finalmente, para validar el prototipo desarrollado, realizamos evaluaciones de 3 aspectos distintos: pruebas de usuario con la *app* para evaluar la usabilidad de esta, un modelo matemático para aproximar la cantidad de salidas que los usuarios de la *app* se ahorrarían, y un experimento para evaluar qué tan "justa" es la selección realizada por el algoritmo. Las primeras 2 evaluaciones son exitosas y logran evaluar de manera positiva el trabajo desarrollado, mientras que la última no logra cumplir con las expectativas por un margen menor.

Respecto a las pruebas de usabilidad, se evalúa la usabilidad mediante unas preguntas que se realizan a los participantes, donde se obtienen puntajes asociados a la usabilidad del sistema. Estos puntajes se promediaron y se obtuvo un resultado que indica que el prototipo cumple satisfactoriamente los estándares mínimos de usabilidad. Aún así, se recomienda para futuras versiones de este trabajo utilizar métodos de evaluación más rigurosos. El modelo matemático que planteamos demuestra a su vez, que el porcentaje de compras que los usuarios se ahorrarían al utilizar la *app* es significativo, y dependiendo de ciertas factores, es capaz de evitar un número significativo de salidas para realizar compras de primera necesidad.

En cuanto a la evaluación de que tan "justa" es la selección realizada por el algoritmo, no se logra cumplir con las expectativas por un margen menor. En esta prueba los participantes tuvieron que seleccionar subconjuntos de productos para comprarle a sus vecinos a partir de una situación específica, donde se espera que su selección sea similar a la selección de productos del algoritmo. Sin embargo, el promedio porcentaje de similitud de estas selecciones no es lo suficientemente similar. Aún así, podemos decir que se alinea en la mayoría de los casos con lo que los usuarios pueden considerar como "justo", ya que a pesar de que el promedio de los subconjuntos escogidos por los participantes no supera la cota planteada inicialmente (75 % de similitud con la selección del algoritmo) , si removemos el caso de menor similitud donde el participante no maximizó la cantidad de productos que podía comprar dadas sus restricciones, logramos superar esta cota con un 77,5 % de similitud.

Gracias a todo esto, podemos decir que el prototipo cumple los objetivos planteados en la Sección 1.2, tanto el general como los específicos. Si bien en el trabajo de esta memoria nos enfocamos únicamente en el desarrollo de la solución planteada, el modelo matemático que desarrollamos nos permite concluir que el uso de un sistema colaborativo para ayudar a comunidades efectivamente permite reducir el número de salidas que las personas deben realiza para comprar productos de primera necesidad. Además de esto, el hecho de que el prototipo mismo sea accesible a los usuarios (gracias a sus niveles de usabilidad) y que las selecciones que realiza el prototipo se alinean con lo que los usuarios consideran "justo" hace que el producto final sea confiable y motive a los usuarios a utilizarla para así reducir su cantidad de salidas.

Respecto a los resultados en sí, además de confirmar que el prototipo desarrollado cumple con la mayoría de las cotas de validación planteadas en el Capítulo 5, también reflejan parte de los elementos a mejorar para la solución final.

Se cree que para el caso de las pruebas de usuario, la puntuación obtenida a través de estas pudo haber sido más alta no solo si hubieramos realizado los cambios de *frontend* propuestos en la Sección 6.1.1, sino también si hubieramos realizado un estudio de usuarios más profunda, no solo con el uso de *mockups*, sino también a través del uso de otras técnicas de

estudio de usuarios, como por ejemplo entrevistas con un mayor número de posibles usuarios que busquen alinear todas las necesidades es estos (*focus groups*), o incluso el *card sorting* donde los participantes agrupan de acuerdo a su propio modelo mental etiquetas o tarjetas en *clusters* que les hacen sentido.

Si bien el modelo indica que podemos reducir la cantidad de salidas en cantidades significativas, este, como todo modelo de un sistema complejo, asume suposiciones que si bien nos parecen razonables, deberían ser validadas en la práctica. Es por esto que es recomendable que, si este trabajo se continúa y el prototipo se vuelve disponible para el público, realizar un estudio con distintos grupos para tener estimaciones más precisas de cuánto ayudaría la solución final a resolver el problema abordado.

Recomendamos modificar el algoritmo de selección con el objetivo de que este pueda comportarse de manera distinta en ciertos casos excepcionales. Como por ejemplo, si existen pedidos de productos que son de la menor necesidad posible (alcohol, tabaco, etc...), estos podrían tener la menor prioridad posible dentro del algoritmo, para que el resto pueda ser seleccionado antes.

A pesar de haber desarrollado *software* en ocasiones anteriores, este trabajo corresponde al primer desarrollo del memorista de una *app* móvil. Sin embargo, gracias al conocimiento adquirido a lo largo de la carrera, el desarrollo de la *app* en sí no resultó de mayor complejidad y podemos decir que logramos trabajar a un buen ritmo. Adicionalmente, me familiaricé no sólo con el uso de nuevas librerías de desarrollo como Kivy, sino también con el uso de tecnologías *cloud* capaces de componer el *backend* de un *software*, las cuales se utilizan cada vez más en el campo laboral del desarrollo de *software*.

Sobre el trabajo en sí, queda como trabajo propuesto además realizar las mejoras correspondientes al *frontend* y el algoritmo en sí, buscar una manera confiable de extraer la información de las tiendas para que no se tengan que ingresar directamente, asegurar que los datos utilizados por la *app* se encuentren seguros y no sean accesible por terceros, verificar el funcionamiento correcto en distintas plataformas, y ver si es posible implementar un sistema de reemplazos de productos que no comprometa a la usabilidad de la *app*.

Bibliografía

[1] World and Health Organization. *QA on coronaviruses (COVID-19).* 2020. https://www.who.int/emergencies/diseases/novel-coronavirus-2019/question-and-answers-hub/q-a-detail/q-a-coronaviruses.

[2] World and Health Organization. *Advice for public.* 2020. https://www.who.int/es/emergencies/diseases/novel-coronavirus-2019/advice-for-public.

[3] elplural. *¿Cuál es la tasa de mortalidad del coronavirus por edades?* 2020. https://www.who.int/emergencies/diseases/novel-coronavirus-2019/question-and-answers-hub/q-a-detail/q-a-coronaviruses.

[4] Python. *Sitio oficial del lenguaje Python.* 2021. https://www.python.org/.

[5] Java. *Oracle Java.* 2021. https://www.oracle.com/java/.

[6] Simplilearn. *Best Programming Languages to Learn in 2021.* 2021. https://www.simplilearn.com/best-programming-languages-start-learning-today-article.

[7] Wikipedia. *Framework para aplicaciones web.* 2020. https://es.wikipedia.org/wiki/Framework_para_aplicaciones_web.

[8] Django. *Django documentation.* 2021. https://docs.djangoproject.com/en/3.1/.

[9] Kivy. *Kivy: Cross-platform Python Framework for NUI Development.* 2021. https://kivy.org/#home.

[10] Kivy. *Kv language.* 2021. https://kivy.org/doc/stable/guide/lang.html.

[11] IBM. *Programación orientada a objetos.* 2020. https://www.ibm.com/docs/es/spss-modeler/SaaS?topic=language-object-oriented-programming.

[12] Red Hat. *El concepto de IDE.* 2021. https://www.redhat.com/es/topics/middleware/what-is-ide.

[13] Android. *Download Android Studio and SDK tools.* 2021. https://developer.android.com/studio.

[14] Jetbrains. *IntelliJ IDEA.* 2021. https://www.jetbrains.com/es-es/idea/.

[15] Google Cloud Platform. *Ventajas de Google Cloud.* 2021. https://cloud.google.com/why-google-cloud.

[16] Wikipedia. *Model-view-controller*. 2021. https://en.wikipedia.org/wiki/Model-view-controller.

[17] Apache. *Overview*. 2021. https://www.apache.org/foundation/.

[18] Tomcat. *Apache Tomcat*. 2021. http://tomcat.apache.org/.

[19] AWS. *AWS*. 2021. https://aws.amazon.com/es/.

[20] IBM. *IBM Cloud, transforme su empresa con Cloud Computing*. 2021. https://www.ibm.com/cl-es/cloud.

[21] IBM. *Db2*. 2021. https://cloud.ibm.com/catalog/services/db2#about.

[22] Wikipedia. *SQL*. 2021. https://es.wikipedia.org/wiki/SQL.

[23] MySQL. *MySQL Documentation*. 2021. https://dev.mysql.com/doc/.

[24] PostgreSQL. *Documentation*. 2021. https://www.postgresql.org/docs/.

[25] Wikipedia. *NoSQL*. 2021. https://es.wikipedia.org/wiki/NoSQL.

[26] MongoDB. *Homepage*. 2021. https://www.mongodb.com/es.

[27] Apache CouchDB. *Homepage*. 2021. https://couchdb.apache.org/.

[28] Wikipedia. *Programación lineal*. 2021. https://es.wikipedia.org/wiki/Programación_lineal.

[29] Wikipedia. *Problema de la mochila*. 2021. https://es.wikipedia.org/wiki/Problema_de_la_mochila.

[30] Wikipedia. *List of knapsack problems*. 2021. https://en.wikipedia.org/wiki/List_of_knapsack_problems.

[31] Wikipedia. *NP-hard*. 2020. https://es.wikipedia.org/wiki/NP-hard.

[32] Wikipedia. *NP*. 2020. https://es.wikipedia.org/wiki/NP_(clase_de_complejidad).

[33] Wikipedia. *Algoritmo voraz*. 2020. https://es.wikipedia.org/wiki/Algoritmo_voraz.

[34] Wikipedia. *Algoritmo de aproximación*. 2020. https://es.wikipedia.org/wiki/Algoritmo_de_aproximación.

[35] Python-MIP. *Documentation*. 2021. https://python-mip.readthedocs.io/en/latest/.

[36] Nag. *nag*. 2021. https://www.nag.com/content/nag-library-for-java.

[37] pypi. *pyeasyga 0.3.1*. 2015. https://pypi.org/project/pyeasyga/.

[38] Kiteknology. *En Chile 94 % de los consumidores realizan compras online*. 2020. https://www.kiteknology.com/es/noticias/en-chile-94-de-los-consumidores-realizan-compras-online.

[39] CNN Chile. *Cornershop, tus compras a domicilio en 90 minutos*. 2016. https://www.cnnchile.com/tecnologias/cornershop-tus-compras-a-domicilio-en-90-minutos_20160229/.

[40] Carla Cabello. *El impacto que tuvo la pandemia en las ventas de delivery de locales de comida en 2020*. La Tercera, 2021. https://www.latercera.com/pulso/noticia/el-impacto-que-tuvo-la-pandemia-en-las-ventas-de-delivery-de-locales-de-comida-en-2020/JWJH4NSXLVGAFEGPRLK6JUX3IU/.

[41] El Libero. *Coronavirus: La estrategia de las empresas de reparto a domicilio para evitar contagios*. 2020. https://ellibero.cl/actualidad/coronavirus-la-estrategia-de-las-empresas-de-reparto-a-domicilio-para-evitar-contagios/.

[42] Cristobal Ríos. *El relato de los trabajadores de delivery que viven la pandemia sin acceso a baños ni elementos de protección*. Ciper, 2021. https://www.ciperchile.cl/2020/04/07/el-relato-de-los-trabajadores-de-delivery-que-viven-la-pandemia-sin-acceso-a-banos-ni-elementos-de-proteccion/.

[43] Listonic. *Funciones*. 2021. https://listonic.com/es/funciones/.

[44] Google Play. *Lista de la compra y despensa gratis - Pantrify*. 2021. https://play.google.com/store/apps/details?id=com.gartorware.pantryapp&hl=es&gl=US.

[45] Cooperativa. *Karpool: La nueva app de viajes compartidos que buscar ser alternativa en Chile*. 2020. https://www.cooperativa.cl/noticias/tecnologia/industria/karpool-la-nueva-app-de-viajes-compartidos-que-buscar-ser-alternativa/2020-02-17/151438.html.

[46] Airbnb. *Airbnb – Home*. 2021. https://www.airbnb.cl/.

[47] El Espectador. *Estos son los 26 productos de primera necesidad fijados por el Gobierno*. 2020. https://www.elespectador.com/coronavirus/estos-son-los-26-productos-de-primera-necesidad-fijados-por-el-gobierno-articulo-913554/.

[48] Pontificia Universidad Católica de Chile. *Glosario*. 2020. http://www7.uc.cl/sw_educ/cyv/html/glos.htm.

[49] TRT Español. *La tasa de mortalidad del coronavirus por edades*. 2020. https://www.trt.net.tr/espanol/photogallery/infografia/la-tasa-de-mortalidad-del-coronavirus-por-edades.

[50] OMS. *Información basíca sobre la COVID-19*. 2020. https://www.who.int/es/emergencies/diseases/novel-coronavirus-2019/question-and-answers-hub/q-a-detail/coronavirus-disease-covid-19.

[51] CDC. *Ciertas afecciones y riesgo de enfermedades*. 2020. https://espanol.cdc.gov/coronavirus/2019-ncov/need-extra-precautions/people-with-medical-conditions.html.

[52] Github. *pypika*. 2021. https://github.com/kayak/pypika.

[53] J Brooke. *SUS – A quick and dirty usability scale*. Adobe Xd, 1996. Usability evaluation in industry 189 (194), pp. 4-7.

[54] A. Bangor. *Grade rankings of SUS scores*. Research Gate, 2008. https://www.researchgate.net/figure/Grade-rankings-of-SUS-scores-from-Determining-What-Individual-SUS-Scores-Mean-Adding-an_fig1_285811057.

[55] Marie Pireddu. *¿Cómo publicar tu aplicación en Google Play y en la App Store?* Good Barber, 2019. https://es.goodbarber.com/blog/como-publicar-tu-aplicacion-en-google-play-y-en-la-app-store-a647/.

[56] SUMI. *What is SUMI?* 1995. https://sumi.uxp.ie/about/whatis.html.

[57] Q.-V. Dang. *Trust assessment in large-scale collaborative systems*. Université de Lorraine, 2018. PhD Thesis.

[58] promptcloud. *The Ethics of Data Scraping*. 2020. https://www.promptcloud.com/blog/is-data-scraping-ethical/.

[59] Kivy. *Create a package for Android*. 2021. https://kivy.org/doc/stable/guide/packaging-android.html.