PriceList

Néstor Aguirre Martínez de Goñi

Proyecto de Desarrollo de Aplicaciones Multiplataforma

DAM 2A

Índice

1. Descripción inicial de la aplicación 4
1.1 Breve explicación del proyecto 4
1.2 Justificación de la necesidad o problema que soluciona 5
2. Objetivos del proyecto 6
2.1 Resultados esperados al finalizar el desarrollo 6
2.2 Beneficios que se esperan obtener 6
3. Recursos hardware y software 7
3.1 Hardware 7
● Ordenador de sobremesa 7
● Móviles android 7
3.2 Software 8
● VSCode 8
● GitHub 8
● Figma 8
● Excel 8
● Word 8
● Oracle VirtualBox 8
4. Fases del desarrollo 9
4.1 Diagrama con la temporalización del proyecto 9
4.2 Descripción de cada fase 10
4.2.1 Fase de análisis 10
4.2.2 Fase de diseño 11
4.2.2.1 Diagrama de la base de datos 11
4.2.2.2 Diagrama de casos de uso 12
4.2.2.3 Planificación de la interfaz 13
4.2.3 Fase de desarrollo 16
4.2.3.1 Enlace al repositorio 16
4.2.3.2 Explicación de código 16
4.2.4 Fase de pruebas y depuración 30
4.2.4.1 Pruebas realizadas 30
4.2.4.1.1 Importación de tickets 30
4.2.4.1.2 Extracción de datos 31
4.2.4.1.3 Organización de productos en familias y subcategorías 31
4.2.4.1.4 Base de datos 31
4.2.4.1.5 Interfaz 32
4.2.4.2 Errores detectados y soluciones 32

4.2.5 Fase de lanzamiento 34
5. Conclusiones **35**
5.1 Reflexión razonada acerca del proyecto 35
5.2 Puntos débiles y fuertes del proyecto 37
5.2.1 Puntos débiles 37
5.2.2 Puntos fuertes 37
5.3 Reflexión de posibles mejoras 38
6. Bibliografía y referencias **39**

1. Descripción inicial de la aplicación

1.1 Breve explicación del proyecto

PriceList es una aplicación para android que analiza tickets de supermercado en formato PDF. Inicialmente estará enfocado en el supermercado Mercadona, pero en un futuro se podría extender a otros supermercados adaptando las funciones de lectura del ticket a los distintos supermercados.

La aplicación extrae los productos comprados y los organiza en familias: alimentos envasados, alimentos frescos, bebidas, congelados, cuidado personal y limpieza. Estas categorías se mostrarán en la pantalla inicial, y al seleccionar una de ellas, se accederá a una nueva vista con distintas subcategorías relacionadas. Una vez que se haya navegado hasta la última subcategoría, se mostrará un listado con los productos correspondientes. Cada producto tendrá los siguientes datos: el precio del último ticket subido, el precio más alto, el más bajo y la media.

La aplicación funciona de la siguiente manera. Primero de todo se debe importar el ticket del mercadona donde se indica, obviamente este tendrá que ser en formato PDF, de lo contrario, dará error. Una vez que se haya importado, la aplicación leerá el ticket y extraerá los nombres de los productos junto a su precio unitario. Tras haberlo extraído, se guardarán los datos en una base de datos que será representada como se ha mencionado anteriormente, con una tabla donde aparecen los siguientes datos: precio del último ticket subido, el precio más alto, el más bajo y la media. Una vez que se haya añadido un ticket, si se inicia la aplicación de nuevo, se podrá navegar directamente a las familias para ver los productos.

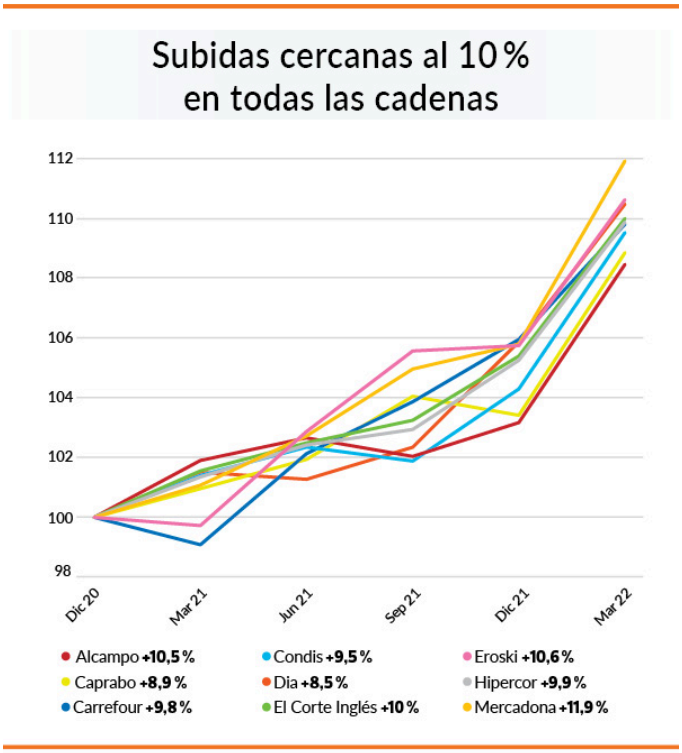
Además, gracias al diseño de la base de datos es posible consultar los precios por fechas, es decir, que se podrá establecer una fecha y en la tabla aparecerán los productos con los precios de dicha fecha. Esto permite hacer un seguimiento de cómo el precio de los productos ha ido subiendo, bajando o manteniéndose.

PriceList está desarrollada con Python y su interfaz con Kivy. Además para la lectura de los tickets en PDF, se ha hecho uso de la librería de python pdfplumber. Para el almacenamiento de los productos en base de datos, como cada usuario tendrá su propia base de datos con sus productos, se ha utilizado una base de datos local con SQLite.

1.2 Justificación de la necesidad o problema que soluciona

La idea de PriceList, es tener una aplicación en el móvil que permita en cualquier momento consultar los precios de los productos de un supermercado (en este caso Mercadona). Esto permite en cualquier momento comparar precios entre un supermercado y el de referencia y ver qué producto interesa comprar y donde.

Por otro lado, PriceList ayuda a aquellas personas preocupadas por la inflación y subidas de precios ya que permite llevar un seguimiento por fechas de los precios. Como se puede ver en la imagen de abajo, prácticamente todos los supermercados han subido los precios en los últimos años exageradamente:



Todos los supermercados están cerca del 10% o incluso lo superan. Los datos del gráfico son del 2022 pero actualmente, en 2025, los precios han seguido subiendo sin parar. Esto es un problema que puede que mucha gente no se esté dando cuenta pero es un problema real que a la larga, hace que los consumidores finales gasten mucho más dinero en sus compras habituales. Con PriceList, este problema se podrá ver a la perfección gracias al seguimiento de precios por fechas y de esta manera, ayudar a los consumidores finales a darse cuenta de este hecho que se está viviendo en el día a día.

2. Objetivos del proyecto

2.1 Resultados esperados al finalizar el desarrollo

Haber conseguido una aplicación móvil funcional para dispositivos Android, desarrollada con Python y el framework Kivy. Uno de los principales objetivos es que la aplicación sea capaz de extraer de forma automática los productos listados en el ticket, junto con sus precios unitarios, y almacenarlos correctamente en una base de datos.

Otro objetivo importante es diseñar una base de datos de tal manera que permita al usuario buscar los productos por fechas para llevar un seguimiento de los precios. Además, los productos se organizarán de forma automática en diferentes familias (alimentos envasados, frescos, bebidas, congelados, productos de cuidado personal o limpieza/hogar), y cada una de estas familias se dividirá en subcategorías para facilitar la navegación. La interfaz desarrollada con Kivy estará diseñada para ser clara, intuitiva y fácil de usar, para que sea cómoda al usuario.

Por último, al acabar el proyecto espero que se quede una aplicación escalable para el futuro ya que como he mencionado anteriormente, en un futuro me gustaría que la aplicación pudiera leer los tickets de más supermercados y no solamente de mercadona.

Dejando de lado los resultados técnicos, se espera conseguir una aplicación que arregle la preocupación de las personas preocupadas por todo el tema que abarca la inflación y la subida de precios. Esto se consigue ofreciendo los datos de los precios según la fecha que el usuario ponga.

También, se quiere conseguir una aplicación novedosa ya que no hay ninguna otra parecida en el mercado que ofrezca lo que PriceList ofrece. Esta aplicación está dirigida al consumidor final ya que son ellos los que compran los productos y se gastan el dinero en ellos.

2.2 Beneficios que se esperan obtener

Al tratarse de una aplicación novedosa que todavía no existe en el mercado se podría valorar la opción de que la aplicación en un inicio sea gratuita y dependiendo de cómo va evolucionando y si se ve que las descargas van aumentando, ponerla de pago y ganar algo de dinero.

Otra manera de obtener beneficio económico sería poner anuncios en la aplicación y si el usuario desea eliminarlos que pague una cantidad de dinero.

Además, se podría valorar la opción de enseñarla a distintas empresas y gracias a la facilidad de escalabilidad para adaptar la aplicación a otros supermercados, ofrecerles añadir la lectura de tickets de su supermercado a cambio de una recompensa económica.

También, se podría lanzar la aplicación inicialmente al consumidor pero a lo largo del tiempo, darle un enfoque hacia las compañías. La aplicación se venderá a aquellas compañías interesadas y estas, posteriormente, se la ofrecerán a los clientes.

Dejando aparte los beneficios económicos, se esperan obtener beneficios sociales. Como se ha mencionado anteriormente, PriceList cubre el problema de la inflación gracias a su seguimiento de precios, de esta manera, todas aquellas personas o familias que utilicen la aplicación podrán ahorrar dinero en las compras y así, conseguir una economía doméstica más sostenida.

3. Recursos hardware y software

3.1 Hardware

- Ordenador de sobremesa

El ordenador de sobremesa ha sido completamente esencial ya que sin él no se podría haber hecho el proyecto. Ha sido utilizado para programar la aplicación y gracias a sus especificaciones me permitía programar y ejecutar el programa rápidamente y sin perder tiempo.

Especificaciones:

- AMD Ryzen 5 3600 (221,43 €)
- 16GB de RAM (64,43 €)
- SSD de 240GB (37,86)
- HDD de 1TB (41,93 €)
- GTX 1650 Super (197,18 €)

- Móviles android

Ya que la aplicación está desarrollada para android he utilizado distintos dispositivos de este tipo para comprobar el correcto funcionamiento de la aplicación en todos los dispositivos.

Los móviles donde se ha probado la aplicación han sido:

- Xiaomi Redmi Note 12 Pro 5G
- Xiaomi Redmi Note 11 Pro 5G
- Xiaomi Redmi Note 9
- Vivo Y33s

3.2 Software

- ### VSCode

El entorno de desarrollo con el que se ha desarrollado la aplicación ha sido VSCode. He decidido utilizarlo además de porque es gratuito, porque durante todo el curso de DAM se ha utilizado y es bastante útil y sencillo de usar además de que cuenta con herramientas como debug y plugins que ayudan al desarrollo.

- ### GitHub

El uso de GitHub ha sido esencial para llevar un buen control de versiones del proyecto, de esta manera en caso de equivocación durante el desarrollo del proyecto siempre se puede volver atrás. Además es gratuito.

- ### Figma

Para desarrollar los bocetos de la interfaz ha sido clave utilizar Figma ya que es gratis, muy útil y fácil de utilizar para desarrollar los bocetos que después serán representados en la aplicación con el uso de Kivy.

- ### Excel

Excel ha sido clave para la hora de organizar todos los productos y clasificarlos por familias ya que te permite tener todo de una manera limpia y organizada para después utilizarlo de base para desarrollar el código de la clasificación de las familias. En este caso, se ha pagado la licencia de Excel pero se podría haber utilizado perfectamente las hojas de cálculo de Google.

- ### Word

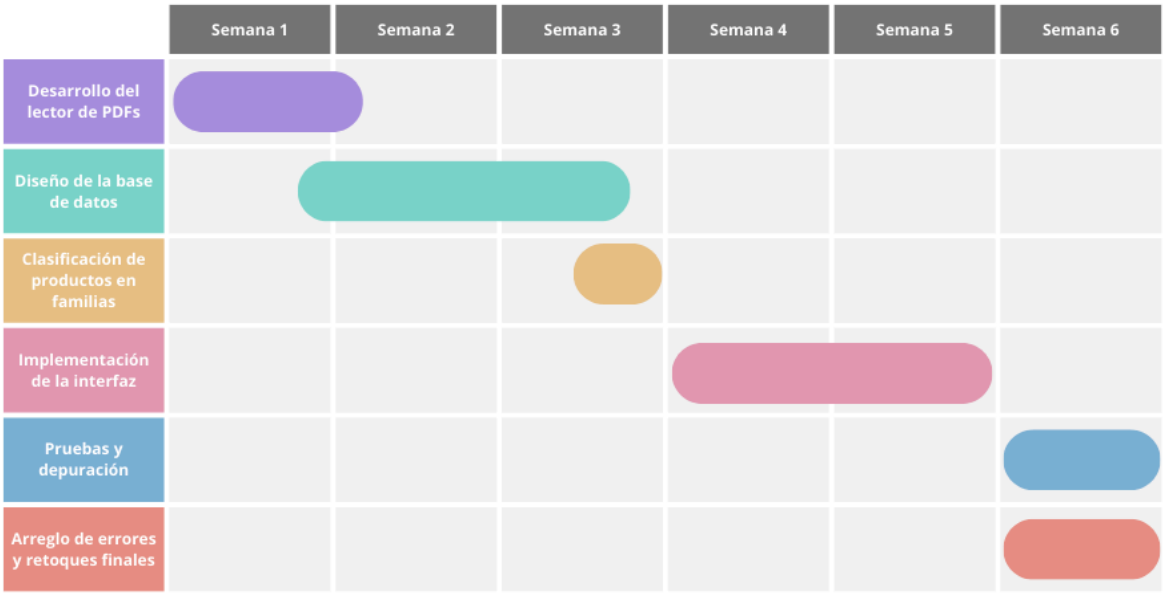
Al igual que Excel, Word también ha sido bastante útil para el tema organización ya que ahí se iban apuntando las ideas de la aplicación y la división de las distintas familias de los productos.

- ### Oracle VirtualBox

A la hora de compilar el proyecto para obtener el archivo .apk es bastante complejo. Para facilitar este proceso, se ha creado una máquina virtual con sistema operativo Linux ya que permite realizar la compilación de una manera sencilla gracias a la herramienta buildozer. Este software es gratuito.

4. Fases del desarrollo

4.1 Diagrama con la temporalización del proyecto



Durante la primera semana se lleva a cabo el desarrollo del lector de PDFs. Esta es una de las claves del proyecto ya que es la funcionalidad que se hará cada vez que el usuario importe un ticket.. Para ello, como se ha mencionado anteriormente, se hace uso de la librería pdfplumber, que permite extraer los nombres de los productos y sus precios unitarios del documento.

En la segunda semana se comienza el diseño de la base de datos, que es otra parte clave ya que almacena la información extraída de los tickets. El diseño de la base de datos se ha alargado un poco más que la primera fase ya que se estuvieron probando distintos diseños de base de datos hasta dar con el adecuado. Además durante la semana 3 también se implementó la funcionalidad de la división de cada producto en su familia correspondiente.

Durante la cuarta y quinta semana se ha desarrollado la interfaz basándose en los bocetos previos hechos con Figma. Esta fase ocupa 2 semanas ya que nunca se había utilizado el framework Kivy para el desarrollo de la interfaz y además se ha intentado conseguir una interfaz intuitiva y simple.

Finalmente, la última semana se ha enfocado para la prueba y depuración de la aplicación. Gracias a esto, se han sacado errores de la aplicación los cuales han sido solucionados y se han aplicado unos retoques finales a la aplicación.

4.2 Descripción de cada fase

4.2.1 Fase de análisis

- **Lectura del ticket (PDF):** Esta funcionalidad es esencial para el desarrollo de la aplicación ya que es la base para poder sacar los productos y los precios de cada producto. Mediante la librería de Python pdfplumber se lee el ticket que el usuario pasa (el ticket tiene que ser PDF). Esto crea un array con todo el contenido que tiene el ticket así que mediante funciones se extraen los productos y los precios unitarios de cada producto y se meten en un diccionario en el cual la clave es el nombre del producto y el valor su precio unitario. Este diccionario será esencial para después meter todo en la base de datos. Dado que esta parte es la base de la aplicación su prioridad es muy alta.

- **Organización de productos:** Para que la aplicación sea más organizada los productos se clasifican en distintas familias y cada familia tiene distintas subfamilias. Para ello, se han escaneado todos los tickets disponibles y se han sacado todos los nombres de los productos. Una vez con todos los nombres de los productos se han pasado a un Excel para clasificarlos por la subfamilia a la que cada uno pertenece. Una vez organizados todos los productos, se guardan en un diccionario de python donde las claves son el nombre de la subfamilia y el valor un array con cada producto de esa familia. Ahora que ya están todos los productos clasificados con su subfamilia perteneciente, solamente queda unir todas las subfamilias a las familias que estas pertenezcan. La prioridad de esta funcionalidad es alta ya que sino, la aplicación estaría muy desorganizada y no darían ganas de usarla.

- **Organización de la base de datos:** La base de datos se ha diseñado de tal manera que dentro de la aplicación se pueda ver la tabla de los productos según la fecha que elija el usuario. Para ello hay 3 tablas, una de tickets, otra de productos y otra de precios. La tabla de tickets almacena ID y fecha, la de productos ID, nombre y familia y finalmente la de precios guarda su ID, el ID de un ticket, el de un producto y el precio del producto. Esto facilitará la búsqueda de precios según la fecha. Para el desarrollo de toda la base de datos se utiliza SQLite. La base de datos tiene una prioridad muy alta ya que es también, una de las partes esenciales de la aplicación para poder consultar los productos y sus precios.

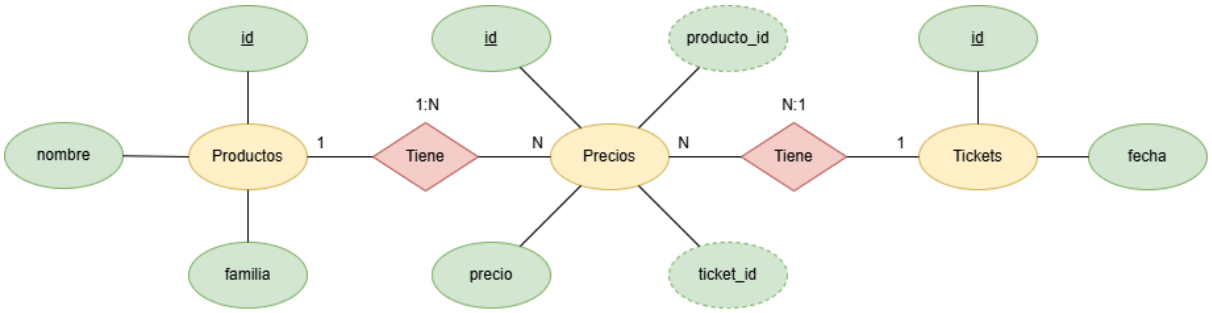
- **Interfaz:** La interfaz será sencilla e intuitiva. Para su desarrollo se ha utilizado Kivy, un framework que permite hacer interfaces multiplataforma sin preocupación de las medidas del dispositivo. La prioridad de esta funcionalidad es muy alta porque sin interfaz no hay aplicación.

- **Estadísticas de precios:** La aplicación, a parte de mostrarte los precios te mostrará su precio más alto, el más bajo y la media. Ya que no se necesita mucho trabajo y simplemente hay que realizar unas consultas su prioridad es media.

- **Precios por fechas:** Finalmente, como se ha mencionado anteriormente, PriceList permitirá hacer un seguimiento de los precios por lo que ofrece la posibilidad de poder consultar los precios por fechas de manera que se pueda comparar los precios de hace 2 meses por ejemplo con los de ahora. La prioridad media ya que consiste en hacer consultas.

4.2.2 Fase de diseño

4.2.2.1 Diagrama de la base de datos



Para el desarrollo de la base de datos, este va a ser el diagrama que se ha tomado como referencia, a continuación se explicará por qué se ha decidido organizarlo así.

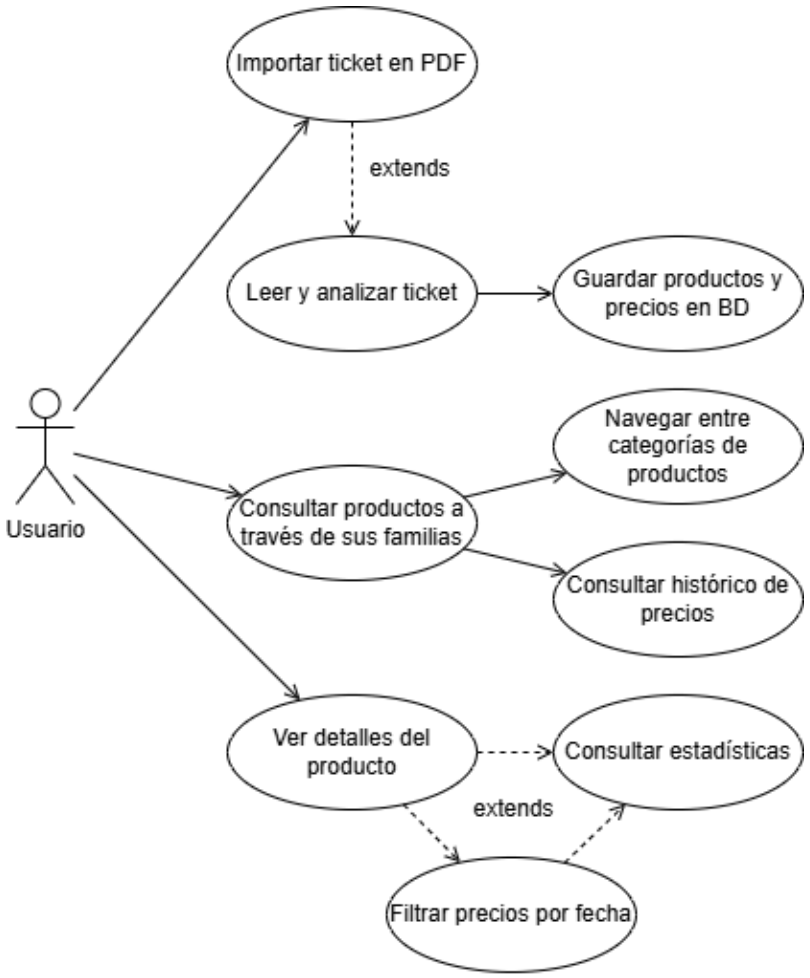
- **Tickets**
  La tabla tickets es una tabla sencilla donde solamente se guardará el ID del ticket (autogenerado) y la fecha en la que se ha insertado el ticket. La fecha cogerá el día, mes y año.

- **Productos**
  La tabla productos se encarga de almacenar el nombre del producto y la familia a la que pertenece. Además al igual que las demás tablas, guarda un ID autogenerado. Los nombres de los productos de esta tabla son únicos por lo que nunca habrá un nombre de un producto que se repita 2 veces.

- **Precios**
  Finalmente está la tabla precios que es la encargada de juntar todas las tablas. Esta cuenta con su propio ID (autogenerado), el precio y los ID de la tabla productos y tickets como claves foráneas. Además, esta tabla será la encargada de mostrar las estadísticas de los productos y para ello se utilizará MAX(), MIN() y AVG()

La tabla que engloba todas las demás tablas es la de precios ya que por como se quiere enfocar la aplicación, es la manera más correcta y sencilla de hacerlo. Esto es debido a que la aplicación va a contar con una opción que sea cambiar la fecha y entonces los precios de los productos cambiarán a los de esa fecha, será simplemente hacer una consulta llamando a la clave foránea del ticket y cogiendo la fecha que el usuario desea.

4.2.2.2 Diagrama de casos de uso



Este es el diagrama de casos de uso que representa la interacción de un usuario con la aplicación.

Primero de todo, el usuario puede importar un ticket en PDF, esto implica que después de importarlo, el sistema lo lea y analice y después de ello, se guarde en la base de datos de la manera que se ha explicado anteriormente.

Además, el usuario también puede consultar los productos a través de sus familias, es decir, que el usuario navegará por las distintas familias y subfamilias del producto para finalmente ver todos los productos pertenecientes a una categoría. Aparte de esto, también se puede consultar el histórico de los productos de una categoría gracias a la arquitectura de la base de datos que se ha mencionado anteriormente.

Finalmente, el usuario también puede ver los detalles de un producto, es decir las estadísticas de este. Estas son: precio, precio máximo, mínimo y la media. Además, los precios se podrán filtrar por fecha.

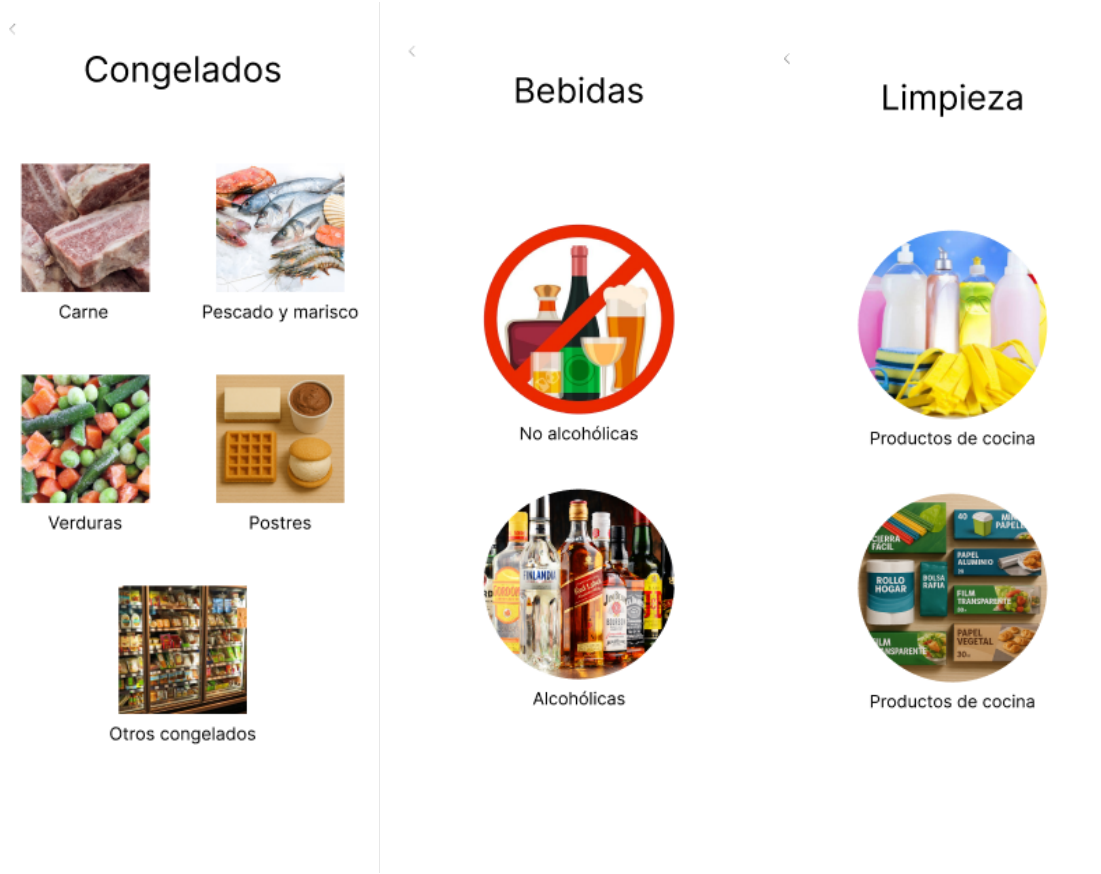
4.2.2.3 Planificación de la interfaz

- **Pantalla principal:**

Esta es la pantalla principal de la aplicación. Contiene las familias más generales de los productos en las que el usuario puede pinchar en una de ellas y navegar por las subcategorías que cada familia contenga hasta llegar al listado final donde se encuentran los productos.

Por otro lado, esta pantalla tiene otro botón distinto a los demás que es donde el usuario importará su ticket del mercadona en formato PDF. Al pinchar en esta opción se abrirá un 'pop up' donde se verán todos los archivos del dispositivo android que se está utilizando. El usuario simplemente tendrá que dirigirse a donde tenga el ticket descargado y seleccionarlo.

- **Navegación subfamilias:**



Cuando se navega por las distintas categorías que tiene una familia, se irán mostrando distintas pantallas como las que se muestran en la imágenes de arriba. Cada una de ellas contiene el título de la familia actual e imágenes de distintas categorías pertenecientes a la familia.

Para la navegación en estas pantallas se debe pinchar sobre una imagen ya que estas actúan como botones, una vez pinchado en una de ellas, puede o que navegue a una nueva subfamilia o al listado final de los productos.

- **Listado de productos:**

Por último, la imagen de arriba muestra cómo se verá el listado de productos una vez que se haya navegado hasta esta pantalla.

Esta última pantalla contiene como título el nombre de la familia final y debajo estará la tabla de productos que mostrará las estadísticas de estos: nombre del producto, último precio, precio máximo, mínimo, y la media.

Además, en esta pantalla es donde se podrá hacer el seguimiento de los precios dependiendo de la fecha que se ajuste. Esto se hará mediante el selector de fecha que hay entre el título de la pantalla y la tabla con los productos.

Paleta de Colores

Principal: Salmón

Fondo: Beige

Texto: Gris oscuro o blanco, dependiendo del color con el que tenga que combinar

Tipografía

Títulos: DancingScript-VariableFont_wght

Texto Principal: DancingScript-VariableFont_wght

Estadísticas: Roboto Mono Medium

4.2.3 Fase de desarrollo

4.2.3.1 Enlace al repositorio

➢ Repositorio - PriceList

4.2.3.2 Explicación de código

- **Lector de tickets:**
  Para la lectura de los tickets se ha creado una clase la cual se encarga única y exclusivamente de esta funcionalidad.

```
class LectorTicket:
    def __init__(self, ticket):
        self.ticket = ticket
        self.arrayTicket = []
        self.diccionarioProductos = {}
        self.extraerTexto()
        self.detectarTicket()
        self.inicioProductos, self.inicioPrecios = self.inicioProductosPrecios()
```

Este es el constructor el cual recibe como parámetro la ruta del ticket y se guarda. Además, se crea un array que contendrá el contenido del ticket y un diccionario de productos que va a ser lo que se devuelve al final de todo el proceso.

Después de ello, se llaman a 3 funciones, “extraerTexto”, “detectarTicket y “inicioProductosPrecios”, esta última guarda 2 valores.

```python
def extraerTexto(self):
    try:
        self.arrayTicket = []
        with pdfplumber.open(self.ticket) as pdf:
            for page in pdf.pages:
                texto = page.extract_text()
                if texto:
                    lineas = texto.split("\n")
                    for linea in lineas:
                        sublineas = re.split(r'\s{2,}', linea.strip())
                        for sub in sublineas:
                            sub = sub.strip()
                            if sub:
                                self.arrayTicket.append(sub)
        return self.arrayTicket
    except Exception:
        raise ErrorTicket("Error leyendo el PDF")
```

Esta es la función “extraerTexto”. Primero abre el contenido del ticket que se ha pasado por parámetro al constructor y recorre todas las páginas. Por cada página, extrae todo el texto y lo guarda.

Debido a los formatos de los tickets, todo el texto queda extraído en distintas líneas y algunas están vacías, por ello, se guarda todo el texto en un array “lineas” mediante el método “split” separándolo por “\n”, es decir, por líneas. Finalmente se recorre este array y por cada línea, debido al formato en el que se escanea el ticket, se divide la línea cada vez que encuentra 2 espacios. Por último, se añade al array.

```python
def detectarTicket(self):
    if not self.arrayTicket or str(self.arrayTicket[0]).strip().upper()[0:9] != "MERCADONA":
        raise ErrorTicket("No se ha podido leer el ticket. Parece que no es un ticket del Mercadona.")
```

A continuación está la función “detectarTicket”. Esta se encarga simplemente de hacer una comprobación de que el ticket pertenece al Mercadona, en caso contrario, saltará un error personalizado avisando de ello.

Dado que por el formato de los tickets en la primera posición del array siempre se va a guardar la palabra “MERCADONA”, eso es lo que se comprueba en la condición del if.

```python
def inicioProductosPrecios(self):
    try:
        for i, linea in enumerate(self.arrayTicket):
            if re.search(r'(?i)\bImporte\b', linea):
                return i
        raise ErrorTicket("No se encontró 'Importe'")
    except Exception:
        raise ErrorTicket("Error en la lectura del ticket")
```

La función “inicioProductosPrecios”, es la que se encarga de detectar en qué posición del array se encuentra el inicio de los productos.

Al igual que para la función de detectar el ticket, los tickets del mercadona siguen un patrón y en este caso, después de la palabra “Importe” empiezan los productos.

Es esencial el uso del módulo de Python “re” ya que da una facilidad increíble para buscar palabras o partes que se necesitan obtener. Lo que hace la función, es recorrer el array que se ha obtenido anteriormente y si encuentra la palabra ‘Importe’, devuelve la posición del array ya que ahí se encuentra el inicio de los productos.

```python
def extraerProductos(self):
    try:
        lista_productos = []
        for linea in self.arrayTicket[self.inicioProductos + 1:]:
            linea = linea.strip()
            if not linea or "TOTAL" in linea or "TARJETA" in linea:
                continue

            match = re.match(r"^(\d+)\s+(.*?)(?:\s+\d+,\d{2}){1,2}$", linea)
            if match:
                cantidad = int(match.group(1))
                nombre = match.group(2).strip().title()
                lista_productos.append((nombre, cantidad))
            else:
                tokens = linea.split()
                if len(tokens) >= 2 and tokens[0].isdigit():
                    cantidad = int(tokens[0])
                    nombre = " ".join(tokens[1:]).title()
                    lista_productos.append((nombre, cantidad))
        return lista_productos
    except Exception:
        raise ErrorTicket("Error en la lectura de productos")
```

Para extraer los productos se hace uso de esta función. Primero de todo se inicializa el array que se devolverá al final de la función. Después, se recorre el array que se ha conseguido con el método “extraerTexto” desde la posición donde se inician los productos, obtenida nuevamente, gracias al método anterior.

Posteriormente, se vuelve a utilizar 're' para filtrar ya que todos los productos siguen un mismo patrón. Si el patrón coincide, divide lo que se ha obtenido en 2 ya que de esta forma se saca la cantidad y el nombre del producto. En caso de que no coincida, hace una revisión menos estricta para ver si se puede obtener algún producto más. Finalmente devuelve la lista de productos.

```python
def extraerPrecios(self):
    try:
        precios = []
        for linea in self.arrayTicket[self.inicioProductos + 1:]:
            if not linea or "TOTAL" in linea:
                continue

            if "€/kg" in linea:
                match = re.search(r"(\d+,\d{2})\s*€/kg", linea)
                if match:
                    precio = float(match.group(1).replace(",", "."))
                    precios.append(round(precio, 2))
                    continue

            # Extraer precios y coger el primero (precio unitario)
            partes = re.findall(r"\d+,\d{2}", linea)
            if partes:
                precio = float(partes[0].replace(",", "."))
                precios.append(round(precio, 2))
        return precios
    except Exception:
        raise ErrorTicket("Error en la lectura de precios")
```

Para la extracción de precios se utiliza esta otra función. Primero se inicializa el array que se devolverá al final de la función. Después al igual que en la función anterior se recorre el array del ticket.

Una vez dentro del bucle, se hacen comprobaciones debido al formato de los tickets. Una comprobación muy importante son para aquellos productos que su precio depende de su peso. En este caso se cogerá el precio de un kilogramo, por ello esa comprobación. Dentro de esa condición utiliza una lógica parecida a la de los productos.

Finalmente, para los productos normales, se vuelve a utilizar 're' para filtrar y se hace un cambio muy importante que es cambiar las comas por puntos. Esto es debido a que los números 'float', en Python se tienen que poner con punto ya que si no, generará fallos.

Por último, se devuelve el array con los precios.

```python
def cargarDiccionario(self):
    try:
        productos = self.extraerProductos()
        precios = self.extraerPrecios()

        for i in range(min(len(productos), len(precios))):
            self.diccionarioProductos[productos[i][0]] = precios[i]

        for basura in ["Parking", "Parking:"]:
            self.diccionarioProductos.pop(basura, None)

        return self.diccionarioProductos
    except:
        raise ErrorTicket("Error cargando diccionario")
```

Como última función en esta clase tenemos “cargarDiccionario”, esta es la encargada de juntar los productos y sus precios. Primero de todo carga tanto la lista de productos como el array de precios.

Una vez cargados, recorre ambos arrays con la condición ‘min’. Esto se ha hecho debido a que a la hora de extraer productos y precios, puede que algún número se haya colado, entonces la propiedad min, recorrerá el bucle hasta que un array se acabe. Después, añade al diccionario el producto como clave y su precio unitario como valor.

Además, se hace una comprobación ya que puede que el ticket contenga la palabra “Parking” y esta se escanea como un producto más. Como esto no se quiere incluir, se borra.

- **Controlador de la base de datos:**

```python
class DBController():

    def __init__(self, nombreDB):
        self.nombreDB = os.path.join(os.path.dirname(__file__), "../data/", nombreDB)
        self.crearDB()
        self.crearTablas()
```

El constructor de esta clase recibe como parámetro el nombre de la base de datos y este se guardará automáticamente con el nombre de la carpeta donde luego se quedará guardada. Además, se crea la base de datos y las tablas que en este caso son tickets, productos y precios.

```python
def crearTablas(self):
    conexion = sql.connect(self.nombreDB)
    cursor = conexion.cursor()

    cursor.execute(
        """
            CREATE TABLE IF NOT EXISTS tickets (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                fecha DATE
            );
        """
    )

    cursor.execute(
        """
            CREATE TABLE IF NOT EXISTS productos (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                nombre TEXT,
                familia TEXT
            );
        """
    )

    cursor.execute(
        """
            CREATE TABLE IF NOT EXISTS precios (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                producto_id INTEGER,
                ticket_id INTEGER,
                precio FLOAT,
                FOREIGN KEY (producto_id) REFERENCES productos(id),
                FOREIGN KEY (ticket_id) REFERENCES tickets(id)
            );

        """
    )

    conexion.commit()
    conexion.close()
```

Para la creación de las tablas se hace uso de esta función donde primero se abre la conexión con la base de datos y se crea un cursor que será el encargado de ejecutar las consultas.

Como se ha mencionado, el cursor se encarga de ejecutar las consultas y en este caso son de creación de tablas. Para ello se utiliza el método “execute” donde como parámetro se le pasa la consulta. Esto se hace para la creación de las 3 tablas que se necesitan.

Finalmente se hace commit para que las tablas se guarden en la base de datos y se cierra la conexión con esta.

```python
def insertarPrecio(self, producto_id, ticket_id, precio):
    conexion = sql.connect(self.nombreDB)
    cursor = conexion.cursor()

    cursor.execute("INSERT INTO precios(producto_id, ticket_id, precio) VALUES (?, ?, ?)", (producto_id, ticket_id, precio))

    conexion.commit()
    conexion.close()
```

Para la inserción de datos en las tablas se han creado funciones para cada tabla. En la imagen de arriba se muestra la de insertar precios que recibe los parámetros necesarios que requiere la tabla.

Al igual que para crear las tablas se abre la conexión y se crea el cursor pero la consulta del método “execute” cambia. En este caso es una consulta de inserción y en los campos donde se meten los datos se ponen interrogantes. Los valores de estos se declaran posteriormente como se observa en la imagen.

Finalmente se vuelve a hacer commit para guardar los datos y se cierra la conexión. Las funciones de inserción son iguales para las 3 tablas aunque en el caso de la tabla de productos hay una condición.

```python
cursor.execute("SELECT id FROM productos WHERE nombre = ?", (nombre,))
resultado = cursor.fetchone()

if resultado is None:
    cursor.execute("INSERT INTO productos(nombre, familia) VALUES (?, ?)", (nombre, familia))
    conexion.commit()
```

En este caso primero se hace una consulta para comprobar si existe algún producto con ese mismo nombre y se guarda el resultado. En caso de que el resultado sea “None” significa que no existe ningún producto con ese nombre entonces se hace la consulta y se guarda, sino, se cierra la conexión.

El resto de la clase son consultas necesarias para el funcionamiento de la aplicación pero que utiliza el mismo formato que todas las demás funciones:

```python
def getProdutoPorNombre(self, nombre_producto):
    conexion = sql.connect(self.nombreDB)
    cursor = conexion.cursor()

    cursor.execute("SELECT id FROM productos WHERE
    resultado = cursor.fetchone()

    conexion.commit()
    conexion.close()

    return resultado[0]
```

```python
def getUltimoTicket(self):
    conexion = sql.connect(self.nombreDB)
    cursor = conexion.cursor()

    cursor.execute("SELECT id FROM tickets OR
    resultado = cursor.fetchone()

    conexion.commit()
    conexion.close()

    return resultado[0]
```

La función más importante de la base de datos es la siguiente:

```python
def getResumenProductosPorFamilia(self, nombre_familia, fecha_limite):
    conexion = sql.connect(self.nombreDB)
    cursor = conexion.cursor()

    cursor.execute("""
        SELECT
            p.nombre,
            (
                SELECT pr.precio
                FROM precios pr
                JOIN tickets t ON pr.ticket_id = t.id
                WHERE pr.producto_id = p.id
                AND t.fecha <= ?
                ORDER BY DATE(t.fecha) DESC
                LIMIT 1
            ) AS precio_actual,
            MIN(pr.precio) AS minimo,
            MAX(pr.precio) AS maximo,
            AVG(pr.precio) AS media
        FROM productos p
        JOIN precios pr ON pr.producto_id = p.id
        JOIN tickets t ON pr.ticket_id = t.id
        WHERE p.familia = ?
        AND t.fecha <= ?
        GROUP BY p.id
        ORDER BY p.nombre ASC
    """, (fecha_limite, nombre_familia, fecha_limite))

    resultados = cursor.fetchall()
    conexion.close()

    productos = []
    for nombre, precio, minimo, maximo, media in resultados:
        productos.append({
            "producto": nombre,
            "precio": f"{precio:.2f}" if precio is not None else "N/A",
            "minimo": f"{minimo:.2f}" if minimo is not None else "N/A",
            "maximo": f"{maximo:.2f}" if maximo is not None else "N/A",
            "media": f"{media:.2f}" if media is not None else "N/A"
        })

    return productos
```

Esta es la función que permite que en el listado de los productos se muestre el nombre del producto junto con su último precio, su precio máximo, mínimo y la media. Para ello, al igual que todas las funciones se crea la conexión con la base de datos y se crea el cursor que va a ejecutar la consulta. Esta consulta hace una unión de todas las tablas ya que, gracias a la arquitectura de la base de datos, todo está relacionado y facilita la extracción de estos datos. Esta consulta recibe como parámetros el nombre de la familia y la fecha límite, esta fecha se consigue de un selector de fecha.

Una vez que se ha hecho la consulta, se guardan todos los datos en un array. Este array se recorre y se guardan los datos en el array que devolverá la función. Los decimales de los precios se limitan a 2 ya que sino no cabrían en la tabla.

- **Carga de productos:**
  Para la carga de productos en el listado, en la interfaz se ha diseñado un “RecyclerView” que se encargará de mostrar los datos. La función encargada de mostrar los productos va ligada a la última función de la base de datos que se ha mencionado anteriormente.
  La función es la siguiente:

```
def cargar_productos(app, familia, nombre_pantalla="listadoproductos"):
    try:
        if platform == "android":
            db = DBController("data/pricelist.db")
        else:
            ruta_absoluta_db = os.path.join(os.path.dirname(__file__), "..", "data", "pricelist.db")
            db = DBController(os.path.abspath(ruta_absoluta_db))

        app.date_picker.current_family = familia

        if app.date_picker.fecha_seleccionada:
            fecha_limite = app.date_picker.fecha_seleccionada
        else:
            fecha_limite = date.today().strftime("%Y-%m-%d")

        datos = db.getResumenProductosPorFamilia(familia, fecha_limite)
        screen = app.sm.get_screen(nombre_pantalla)

        screen.ids.rv.data = []
        screen.ids.rv.refresh_from_data()
        screen.ids.rv.data = datos

    except Exception as e:
        Logger.error(f"Mostrar productos: Error al cargar datos -> {e}")
```

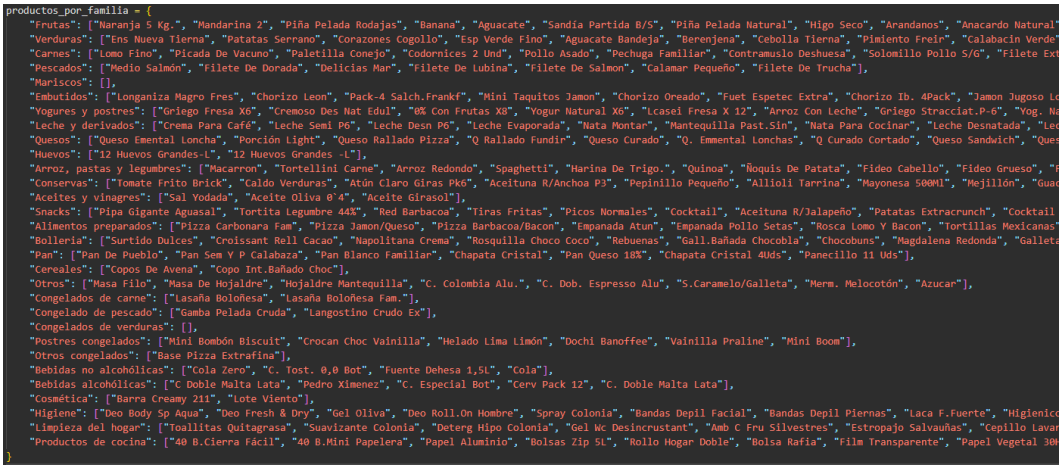
  Primero de todo, dependiendo de en qué plataforma se abra la aplicación, se conecta de una manera distinta a la base de datos. Después se comprueba si el usuario ha introducido una fecha en el selector de fecha y si no, se pone por defecto la fecha actual. Una vez que ya hay una fecha disponible, se llama a la función mencionada anteriormente y se guardan los datos que devuelve. Finalmente se carga la pantalla donde está el “RecyclerView” y a este se le cargan los datos que ha devuelto la consulta. Para ello, hay una clase específica que guarda los datos que devuelve la consulta:

```
class ProductosRecyclerView(BoxLayout):
    producto = StringProperty("")
    precio = StringProperty("")
    maximo = StringProperty("")
    minimo = StringProperty("")
    media = StringProperty("")
```

- **Organizar productos en familias:**
  Para la organización de los productos en familias se han planteado varias opciones. Primero se planteó la opción de utilizar una API con los productos del mercadona pero no existe ninguna donde los nombres coincidan con los mismos nombres que se ponen en los tickets.

  Otra opción que se planteó fue implementar una API de inteligencia artificial a la cual se le enviaba un prompt y la IA devolvía el nombre de la familia. El problema de esto ha sido que la mayoría de inteligencias artificiales no permitían realizar esto de manera gratuita. Finalmente se encontró una que era capaz de hacerlo, el problema era que tardaba demasiado tiempo para clasificar los productos y además había varios productos que se clasificaban en familias que no eran o directamente ni se clasificaban.

  Finalmente se ha decidido crear un diccionario donde la clave es la familia y el valor un array con todos los productos pertenecientes a dicha familia. Este diccionario se ha rellenado manualmente añadiendo los máximos productos posibles. Estos productos se han sacado de todos los tickets que se tenían en disposición.



  Es posible que cuando se inserte un ticket, haya productos nuevos que no existan en el diccionario. Para solucionar este problema, si el producto no existe se preguntará al usuario a qué familia pertenece y este producto se añadirá al diccionario y se quedará guardado para las demás veces.

- **Interfaz:**
  Como se ha mencionado anteriormente, la interfaz se ha realizado con el framework Kivy. Se ha escogido gracias a su facilidad de hacer la interfaz para distintos dispositivos.

Esta es la pantalla principal de la aplicación. Cuenta con el título y 7 botones de los cuales 6 son para acceder a las familias.

```
Button:
    text: "Envasados"
    font_size: app.button_font_size
    bold: True
    color: 0.2, 0.2, 0.2, 1
    background_normal: ''
    background_color: 0.88, 0.48, 0.37, 1
    font_name: "assets/fonts/DancingScript-VariableFont_wght.ttf"
    canvas.before:
        Color:
            rgba: self.background_color
    on_release: app.change_screen('envasados')
```

Cada botón tiene la misma estructura. Todo es bastante intuitivo pero hay partes donde pone “app.”. Lo que viene después de ello son nombres de funciones desarrolladas en otro archivo de Python. En el caso de los botones de las familias se utiliza para el tamaño del texto y para cuando se pulsa el botón.

```
def change_screen(self, screen_name):
    if self.sm.current != screen_name:
        self.screen_history.append(self.sm.current)
        self.sm.transition.direction = 'left'
        self.sm.current = screen_name
```

La función de la imagen de arriba es para cambiar de pantalla. Recibe como parámetro el nombre de la pantalla a la cual se quiere dirigir. Cada nombre de pantalla se ha declarado previamente en otra función:

```
self.sm = ScreenManager()
self.sm.add_widget(MenuScreen(name="menu"))
self.sm.add_widget(BebidasScreen(name="bebidas"))
self.sm.add_widget(CuidadoPersonalScreen(name="cuidadopersonal"))
self.sm.add_widget(LimpiezaScreen(name="limpieza"))
self.sm.add_widget(CongeladosScreen(name="congelados"))
self.sm.add_widget(EnvasadosScreen(name="envasados"))
self.sm.add_widget(ListadoProductosScreen(name="listadoproductos"))
```

El parámetro “name” es el que hay que pasar por parámetro para acceder a la pantalla que se desea.

Continuando con el método de cambiar de pantalla, empieza con una comprobación para comprobar que no se está ya en la pantalla a la que se quiere cambiar. Después, se guarda en un array que contiene el historial de las pantallas que se han visitado. Esto se utilizará para el botón de volver atrás que se explica posteriormente. Una vez guardado en el historial, se declara hacia que lado se va a hacer la transición y se cambia el nombre de la pantalla actual.

Esto se realiza para todos los botones de las familias de la pantalla principal.

El botón para importar el ticket funciona de una manera distinta ya que, obviamente, no tiene que hacer lo mismo que el resto de botones. En este caso, cuando se pulsa el botón, llama a la función “abrir_filechooser”.

```
on_release: app.abrir_filechooser()
```

Esta función funciona de la siguiente manera:

```
def abrir_filechooser(app):
    try:
        filechooser.open_file(
            title="Selecciona un archivo PDF",
            filters=["*.pdf"],
            on_selection=lambda seleccion: on_archivo_seleccionado(seleccion, app)
        )
    except Exception as e:
        Logger.error(f"FileChooser: Error al abrir el selector de archivos: {e}")
```

Primero abre el “filechooser” y le asigna un título, el filtro para que solo aparezcan los archivos en PDF y al seleccionar un archivo llama a la función “on_archivo_seleccionado”:

```
def on_archivo_seleccionado(seleccion, app):
    if not seleccion:
        Logger.info("No se seleccionó ningún archivo.")
        return

    ruta = seleccion[0]
    Logger.info(f"Archivo seleccionado: {ruta}")

    try:
        lector = LectorTicket(ruta)
        db_path = get_db_path()
        db = DBController(db_path)

        db.insertarTicket(lector.getFechaTicket())
        ticket_id = db.getUltimoTicket()

        for producto, precio in lector.cargarDiccionario().items():
            producto_normalizado = producto.strip().lower()
            familia = "otraFamilia"

            for key, lista_productos in productos_por_familia.items():
                for p in lista_productos:
                    if p.strip().lower() == producto_normalizado:
                        familia = key
                        break
                else:
                    continue
                break

            if familia != "otraFamilia":
                db.insertarProducto(producto, familia)
                db.insertarPrecio(db.getProductoPorNombre(producto), ticket_id, precio)

        Logger.info("Archivo procesado correctamente.")
        Clock.schedule_once(lambda dt: Snackbar(text="Ticket importado correctamente", duration=2).open())

    except Exception as e:
        Logger.error(f"Procesamiento de PDF: Error -> {e}")
        Clock.schedule_once(lambda dt: Snackbar(text="Error al importar el ticket", duration=2).open())
```

Lo primero que hace es comprobar que se ha seleccionado un archivo, sino, se acaba la función. En caso de que se haya seleccionado un archivo, se guarda la ruta del archivo ya que luego se necesita cuando se crea el lector de tickets. Al mismo tiempo, se conecta a la base de datos y lo primero que hace después de ello es insertar el ticket con la fecha de este y después se coge el ID del ticket que se acaba de añadir a la base de datos, este se necesitará más tarde.

Después, se inicia un bucle que recorre todos los productos que se han extraído del ticket, de este se cogen los productos y sus precios. Posteriormente, recorre el diccionario donde se guardan todos los productos clasificados en familias. De este diccionario se recorren todos los productos de cada familia y si en alguno de ellos coincide un producto con el producto del ticket, se guarda la familia y se sale del bucle. Finalmente, si el producto tiene familia, se inserta el producto y el precio donde se utiliza el id del ticket mencionado anteriormente.

En el caso de aquellos productos a los que no se haya asignado ninguna familia, se guardarán y tras la carga de todos los productos, se preguntará al usuario para que vaya clasificando aquellos productos nuevos. Para ello se ha creado un controlador con 2 funciones clave:

```
def mostrar(self):
    self.app.clasificador_popup = self  # Para acceso desde .kv

    from kivy.factory import Factory
    layout = Factory.ClasificadorPopupLayout()
    self.label = layout.ids.producto_label
    self.label.text = f"Producto: {self.producto}"
    self.button_select = layout.ids.button_select

    scroll = ScrollView(size_hint=(1, None), height=150)
    scroll.add_widget(layout)

    self.dialog = MDDialog(
        title=f"Asignar familia a '{self.producto}'",
        type="custom",
        content_cls=scroll,
        buttons=[
            MDFlatButton(text="CANCELAR", on_release=self.cerrar),
            MDFlatButton(text="GUARDAR", on_release=self.guardar)
        ]
    )
    self.dialog.open()
```

La primera de ellas es la encargada de mostrar un “PopUp” cuando se detectan nuevos productos. Primero de todo se crea el PopUp mediante la importación de “factory”, además, se crea una etiqueta donde se mostrará el nombre del producto y un botón encargado de mostrar las diferentes familias a las que puede pertenecer un producto. También, se crea un “scroll” ya que será necesario por la cantidad de familias que se dan a elegir. Finalmente se crea el diálogo para que el usuario pueda ver el “PopUp”.

La otra función clave que se ha desarrollado es la siguiente:

```
def abrir_menu(self, instance):
    if not self.menu:
        self.menu = MDDropdownMenu(
            caller=self.button_select,
            items=[
                {
                    "viewclass": "OneLineListItem",
                    "text": familia,
                    "on_release": lambda x=familia: self.seleccionar_familia(x),
                }
                for familia in FAMILIAS_FIJAS
            ],
            width_mult=3,
        )
    self.button_select.parent.do_layout()
    Clock.schedule_once(lambda dt: self.menu.open(), 0.3)
```

Esta función se encarga de abrir el menú con todas las familias disponibles para asignar a un productos. Primero hace una comprobación para ver si ya hay un menú abierto para evitar abrir múltiples menús. Tras la comprobación, se crea el menú y por cada elemento de un listado asignado en otro archivo, se crea una nueva línea en el menú con el nombre de cada una de las familias. Finalmente, se añade un poco de retraso para que se abra el menú ya que si no se da tiempo, el menú puede fallar.

4.2.4 Fase de pruebas y depuración

A continuación se comentarán las distintas pruebas que se han realizado para la prueba de la aplicación y los errores que se hayan encontrado. También se comentará cómo se han solucionado estos fallos.

4.2.4.1 Pruebas realizadas

4.2.4.1.1 Importación de tickets

- **Importación de tickets correctos:**
  Para la importación de tickets, en este caso tickets correctos, se han importado el máximo número de tickets que se disponía (31) y se ha comprobado que durante la inserción no haya ningún error y se importe todo correctamente.

- **Importación de tickets erróneos:**
  Se han creado varios tickets erróneos e importado distintos PDFs que no eran tickets para comprobar que no se realizaba la importación y que en vez de que la aplicación ‘crashee’ que notifique al usuario con un mensaje de error.

- **Importación de otros archivos que no sean .pdf:**
  Se ha intentado importar archivos de distintos formatos como .png, .jpg, .mp3, .mp4 para comprobar que la aplicación controla estos errores sin que esta deje de funcionar.

4.2.4.1.2 Extracción de datos

- **Extracción correcta de productos y precios:**
  De cada ticket, se ha comprobado que el nombre de los productos se extraía correctamente al igual que los precios de cada uno

- **Control de duplicados:**
  Puede darse la ocasión de que en un mismo ticket exista el nombre de un producto más de una vez como se muestra en la imagen de abajo, en este caso, solo se extraerá un producto con ese nombre para evitar duplicados y fallos en la extracción del PDF.

1	SAN JACOBO S/G	2,37
1	SAN JACOBO S/G	2,41

En este caso, se guardará “San Jacobo S/G” y “2,41”.

4.2.4.1.3 Organización de productos en familias y subcategorías

- **Navegación por la aplicación:**
  Tras haberse importado el ticket, cada producto debería haberse clasificado en su respectiva familia, para ello, se ha navegado por las distintas familias y subcategorías y se ha comprobado la correcta visualización de los productos en sus respectivas familias.

- **¿Qué pasa si se inserta un producto nuevo?:**
  Como se ha mencionado anteriormente, la aplicación viene con una clasificación de familias inicial en un diccionario pero este no contiene todos los productos del Mercadona. Si se inserta un producto que no se encuentra en el diccionario, tras haberse importado el ticket, aparecerá un “pop up” preguntando al usuario a qué familia pertenece el producto nuevo y se le proporcionará un listado con todas las familias para que el usuario clasifique el producto donde quiera. Una vez insertado, se comprobará que, efectivamente, el producto aparece en la familia asignada.

4.2.4.1.4 Base de datos

- **Control de duplicados:**
  Se ha comprobado que en el caso de que el nombre de un producto se repita, este no se vuelva a guardar en la base de datos ya que es totalmente innecesario. Sin embargo, cuando se repite el nombre de un producto, el precio si que se actualiza y se guarda.

- **Correcta visualización de estadísticas:**
  Tras insertar un ticket, se ha navegado hasta la pantalla del listado de los productos para ver que se han guardado los precios correctamente. Una vez comprobado, se vuelve a insertar otro ticket que contenga un producto que ya esté en base de datos y con un precio distinto. Tras ello, se vuelve a ir al listado de los productos y se comprueba que: el precio sea el del ticket con la fecha más cercana al actual, que el máximo sea el precio máximo registrado de el producto, lo mismo con el precio mínimo y por último, que se calcule la media correctamente.

- **Selector de fecha:**
  Para el seguimiento de productos por fecha, en el listado con todos los productos, se ha implementado un selector de fecha donde se elige una fecha y se cargan los productos y sus precios hasta esa fecha incluída. Para probar esto se han metido varios tickets con distintas fechas y se han ido filtrando distintas fechas para comprobar su correcto funcionamiento.

4.2.4.1.5 Interfaz

- **Diseño adaptable:**
  La aplicación se ha instalado en distintos dispositivos android para comprobar que la interfaz no se ve mal en ninguno de ellos y por ende afirmar que el diseño es adaptable al dispositivo en el que se utiliza. Gran parte de esto ha sido gracias al “Scroll View” ya que sin él, se intentaban meter todas las imágenes de las familias en la pantalla y no había forma de que quedara ni bien, ni adaptable a distintos dispositivos.

4.2.4.2 Errores detectados y soluciones

- **Pantalla negra inicial:**
  Cuando se iniciaba la aplicación en android, la pantalla se quedaba en negro y hasta que no se interactuaba con un botón de la pantalla inicial, esta no cargaba. Este error se ha arreglado pasando la carga de las vistas del “build” a un método a parte y en el build entonces añadir la siguiente línea de código:

```
Clock.schedule_once(self.load_remaining_screens, 0)
```

  “Clock.schedule_once” permite ejecutar una función para que se ejecute una vez después del tiempo que se indique, entonces, se ha puesto que se ejecute en 0 segundos así que nada más iniciar la aplicación se va a ejecutar y de esta forma la pantalla negra desaparecerá.

- **Tamaños pequeños:**
  La primera vez que se ha abierto la aplicación en un dispositivo android, ha saltado a la vista que tanto todas las letras como la tabla donde se listan los productos se veían demasiado pequeñas complicando la lectura al usuario. Aunque en la ejecución en el entorno de desarrollo que se ha ejecutado se veía todo con los tamaños perfectos, en los dispositivos android no era así. Para solucionar este problema solamente se ha tenido que aumentar los tamaños en una función que se encarga de hacer adaptable lo mencionado anteriormente.

- **"Crash" al volver atrás:**
  Para volver a la pantalla anterior, la aplicación cuenta con un botón en la parte superior a la izquierda con la que se vuelve hacia atrás. El problema es que se suele tener la costumbre de volver hacia atrás o bien haciendo el gesto de volver hacia atrás o dándole al botón de atrás del mismo móvil:

  Entonces, si se vuelve hacia atrás de alguna de esas 2 maneras, la aplicación "crasheaba". Esto se ha solucionado añadiendo una función que captura las teclas del teclado. La tecla correspondiente en android para volver atrás es la 27 por lo que si la función detecta que se ha pulsado el botón de atrás, entonces ya no "crashea".

- **Errores en la compilación:**
  Aunque el sistema operativo linux facilitase la compilación de la aplicación a un archivo .apk, no ha sido tan fácil como se esperaba debido a la gran cantidad de errores que se producían durante la compilación.

  Primero de todo, no ha sido un proceso fácil toda la configuración de la máquina virtual de Ubuntu ya que había que instalar muchas librerías, dependencias, etc y no se ha encontrado ni un solo vídeo donde siguiendo todos los pasos se consiguiera una compilación exitosa. Cada vez que se intentaba compilar, daba errores porque siempre faltaba algo por instalar, además, los errores que generaba la terminal eran extensos y difícil de encontrar donde estaba el fallo así que después de largas conversaciones con distintas inteligencias artificiales finalmente se ha conseguido compilar sin que de fallos por falta de instalaciones.

  Estos no han sido los únicos errores que se han dado a la hora de compilar. La compilación se ha hecho con buildozer y para ello, el proyecto tiene que tener un archivo llamado "buildozer.spec". Este archivo es una de las claves para que la aplicación compile por lo que tiene que estar perfectamente hecho ya que sino, la compilación va a fallar. Hasta que no se consiguió dar con el "buildozer.spec" perfecto, no había manera de conseguir que la aplicación compilase pero finalmente se consiguió.

- **“Crasheo” inicial al abrir la aplicación:**
  Una vez que se consiguió que la aplicación compilase, lo primero que ocurre al abrirse en android es un “crasheo” nada más iniciarse por lo que no se podía hacer nada con la aplicación. A pesar de que en el entorno de desarrollo, la aplicación funcionara perfectamente, en android no se podía ni abrir. Esto se debía a que algunas de las funcionalidades que se habían desarrollado, solo funcionaban en PC y no eran compatibles con android. Para solucionar este error se tuvo que ir buscando minuciosamente donde estaban aquellas funcionalidades que generaban este fallo y una vez encontradas, adaptarlas para que se pudieran utilizar en android.

- **Base de datos no detectada en android:**
  Aunque en la ejecución en el PC la aplicación detectase a la perfección la base de datos, en android, si se navegaba al listado de productos estaba siempre vacío porque no la encontraba. Para solucionar este problema se han tenido que configurar bien las rutas a la base de datos.

- **Error al importar el ticket 1:**
  Al igual que en varios de los problemas, en PC todo funcionaba a la perfección, sin embargo, en android ya no funcionaba. Lo mismo ocurrió con la funcionalidad de importar el ticket. Inicialmente la clase para leer los PDFs se desarrolló con “pdfminer.six” y cuando se quería insertar un ticket en android, siempre daba error. Para ello, se hizo “logcat” y de esta manera se podría ver el error que causaba el fallo. Tras consultarse el error con varias inteligencias artificiales, resulta que “pdfminer.six” necesita unas dependencias que solo se pueden tener en PC por lo que el lector de PDFs que se desarrolló no servía. Para solucionar este problema, se usó “pdfplumber” en vez de la anterior librería y tras adaptar la clase entera, ya se consiguió la importación correcta de los tickets

- **Error al importar el ticket 2:**
  Tras finalmente conseguir que el ticket se importe correctamente, los productos del ticket importado no se ven reflejados después en el listado de productos. Además, cuando se abre el “PopUp” de los nuevos productos detectados, la aplicación empieza a ralentizarse y depende de si hay varios productos para importar nuevos, el seleccionador de familias empieza a fallar y el “PopUp” empieza a fallar. Desgraciadamente, para este error no se ha encontrado la solución en los dispositivos android.

4.2.5 Fase de lanzamiento

Aunque PriceList se ha pensado para una distribución pública y que todo el mundo la pueda usar, el primer lanzamiento de la aplicación va a ser una distribución privada y controlada seleccionando a los usuarios para que la utilicen. De esta manera, cuando los usuarios seleccionados lleven un tiempo

usando la aplicación, es probable que se vayan encontrando fallos, aspectos a mejorar, nuevas funcionalidades que se podrían implementar, etc. De esta manera y tras consultarse con los usuarios de prueba, cuando la aplicación ya se haya actualizado con las propuestas de los usuarios ya estará lista para publicarla en Google Play.

Uno de los requisitos para poder publicar una aplicación en Play Store es tener el proyecto compilado en un archivo con la extensión .apk. Para conseguir este archivo se ha utilizado la herramienta Buildozer que permite empaquetar aplicaciones desarrolladas con Python y Kivy.

Una vez conseguido el archivo .apk será necesario crearse una cuenta como desarrollador en Google Play Console, para ello se necesita una cuenta de Google y pagar una tasa única de 25$. Tras haberse creado la cuenta ya se podrán gestionar las publicaciones, actualizaciones, estadísticas de la aplicación, etc.

Tras haberse creado la cuenta, Google necesita que la aplicación esté firmada digitalmente. Para ello, se debe crear una clave de firma y firmar la aplicación para poder subirla. Esta firma se necesita para identificar al desarrollador y de esta manera garantizar que la aplicación no ha sido modificada por otras personas. Para firmar la aplicación, Buildozer facilita el proceso ya que con configurar el archivo “buildozer.spec” correctamente, la aplicación ya quedará firmada.

Después de que se haya firmado la aplicación, en la consola de Google Play se deberán especificar campos como el nombre de la aplicación, descripción, categoría, etc, tras ello ya se podrá subir la aplicación en formato .apk.

Finalmente, la aplicación quedará en revisión y cuando Google dé el visto bueno, esta se publicará y a partir de ahí se podrá actualizar la aplicación, ver las estadísticas de descargas, valoraciones, etc.

5. Conclusiones

5.1 Reflexión razonada acerca del proyecto

Para concluir, estoy muy orgulloso y satisfecho de la aplicación que he conseguido. Además, estoy contento ya que es una aplicación que me ha pedido una persona y sé que le va a ser muy útil en su día a día y que no va a ser una aplicación que se va a quedar en el olvido. Aparte de eso, estoy muy contento ya que he cumplido todas las cosas propuestas incluso añadiendo algo más de lo propuesto en la propuesta inicial.

Este proyecto me ha enseñado la realidad de enfrentarte tú solo ante el desarrollo desde 0 de una aplicación y todo lo que conlleva, desde crear la idea inicial, diseñar los bocetos, etc hasta conseguir finalmente el resultado esperado. Me gustaría comentar la gran cantidad de tiempo que se necesita

para desarrollar una aplicación ya que cuando crees que está ya todo listo aparecen millones de errores que en un principio no sabes ni por qué pasan ni cómo arreglarlos. Aparte de la resolución de errores, que es una gran cantidad de tiempo durante el desarrollo del proyecto, obviamente también está la parte del desarrollo de la aplicación. Desde fuera todo parece más sencillo hasta que te pones a programar y lo que tenías pensado no es tan fácil de hacer, lo mismo pasa con la interfaz, que quieres hacer muchas cosas pero luego se complica y acabas haciendo menos, pero a pesar de ello, he logrado una aplicación funcional y bonita, además de útil.

Como ya se ha mencionado durante el resto de la memoria, PriceList es una aplicación que va a llevar un seguimiento de los precios por fechas y esto, aunque a priori no parezca gran cosa, va a servir para darnos cuenta de la realidad y de los aumentos de precios que estamos sufriendo los compradores. Durante el desarrollo de la aplicación, iba metiendo tickets para comprobar el funcionamiento, y además, me iba fijando en como iban subiendo los precios y hay productos que han llegado a subir casi 1€:

1 AGUACATE
0,990 kg 4,59 €/kg

1 AGUACATE
0,884 kg 5,30 €/kg

Igual hay gente que piensa que 1€ no es gran diferencia, pero a la larga, ir gastándote cada compra 1€ más va a salir muy caro. Esto es un problema real que se está viviendo, y la aplicación que he desarrollado, a mi parecer, pienso que va a ser bastante útil para darnos cuenta de la inflación que se está viviendo hoy en día.

Dejando de lado el tema de la inflación, me gustaría destacar todos los conocimientos adquiridos durante el desarrollo de la aplicación ya que durante el grado, hemos hecho bastante poco de Python por lo que me ha tocado aprender a usar de verdad este lenguaje. Para el aprendizaje de Python, ha sido clave que en mi empresa de las FCT es todo con Python por lo que iba cogiendo ideas de los códigos que tenían desarrollados y luego las iba adaptando a las necesidades de mi proyecto.

Por otro lado, toda la interfaz está hecha con el framework Kivy. Antes de empezar el proyecto no sabía ni que existía este lenguaje pero leí que era bastante útil para desarrollar aplicaciones multiplataforma con Python y decidí utilizarlo. Al tratarse de algo nuevo, he tenido que ir siguiendo bastantes tutoriales y preguntándole a las inteligencias artificiales pero la realidad es que es un lenguaje bastante sencillo ya que las etiquetas que utiliza son bastante sencillas y además, era verdad que es muy útil para aplicaciones multiplataforma ya que con ir jugando con los distintos “layouts” se consigue una interfaz adaptable para todos los dispositivos.

Para acabar, yo creo que va a ser una aplicación que voy a seguir desarrollando ya que me gustaría ir añadiendo nuevos supermercados, nuevas funcionalidades e ir mejorando aquellos aspectos que durante el uso de la aplicación vaya viendo que se podrían mejorar. Además, también me gustaría sacar la aplicación para distintas plataformas como puede ser iOS para que los usuarios con ese sistema operativo también puedan utilizar mi aplicación.

5.2 Puntos débiles y fuertes del proyecto

5.2.1 Puntos débiles

- **Cambio del formato del ticket:**
  Uno de los puntos más débiles de PriceList es que si algún día se cambia el formato del ticket, el lector de tickets dejará de funcionar y no se podrán importar tickets. Esto se debe a que la clase encargada de esta funcionalidad está desarrollada única y exclusivamente al formato de los tickets actuales.

- **Exclusiva para un supermercado:**
  La aplicación se ha diseñado para que solamente lea los tickets del supermercado Mercadona. Esto es un gran problema si se espera sacar beneficio de la aplicación ya que si se publica, es probable que solo la descargue la gente que compre habitualmente en Mercadona y por ende, perder la oportunidad de que muchísimas más personas utilicen PriceList.

- **Nombres de los productos:**
  Un punto débil de la aplicación es que los nombres que se guardan en la base de datos son los mismos que se ven en el ticket y estos no se normalizan por lo que puede haber ciertos productos que no se sepan que son.

5.2.2 Puntos fuertes

- **Gran escalabilidad:**
  Un gran punto fuerte de la aplicación y que cubre los 2 primeros puntos débiles mencionados anteriormente es la gran escalabilidad que tiene PriceList. Para solucionar el tema del formato de los tickets, en caso de que este cambie, simplemente se tendrán que adaptar las funciones de la clase encargada de la lectura de los tickets y ya se podrán volver a leer los tickets a la perfección. Además, esta clase no cuenta con muchos métodos por lo que no será una tarea de mucho trabajo.

  Por otro lado, para arreglar el problema de que la aplicación está dedicada solamente al Mercadona, si se quieren añadir nuevos supermercados a la aplicación, simplemente se

tendrán que seguir los mismos pasos que para el caso anterior ya que solamente se tendrán que adaptar las funciones a este nuevo supermercado, el resto del programa se quedaría intacto y funcionaría a la perfección.

- **Buena organización:**
  PriceList se ha desarrollado detalladamente para que esté muy bien organizada. Se podría haber optado por la manera fácil de poner todos los productos en una misma tabla sin clasificación alguna, pero se ha decidido crear distintas familias y si es necesario dentro de las familias subcategorías para facilitar al máximo la experiencia al usuario. Además de mejorar la experiencia al usuario, esto supone un ahorro de tiempo ya que no se tiene que poner a buscar el producto en una tabla enorme sino que con ir a la familia correspondiente lo podrá encontrar con facilidad.

- **Gran utilidad:**
  Con esta aplicación se permite llevar un control a tiempo real y detallado sobre los productos que se compran habitualmente. De esta manera, no se tiene que estar buscando constantemente en distintos sitios la subida de los precios sino que se puede ver en la aplicación y además totalmente enfocado en los productos habituales que se compran.

5.3 Reflexión de posibles mejoras

- **Aumentar listado de productos:**
  Para la clasificación de productos en familias, se utiliza un diccionario donde inicialmente tiene unos productos asignados. Estos productos son los productos de los tickets que se han conseguido durante el proceso de desarrollo y se tratan de tickets de la compra semanal. Este diccionario no contiene muchos de los productos existentes en Mercadona por lo que es posible que una nueva persona que utilice la aplicación, los productos de sus compras habituales no coincidan con los productos existentes en el diccionario y las primeras compras tenga que ir clasificando manualmente varios productos. Con el uso de la aplicación esto ya no será un problema ya que la mayoría de productos ya se habrán guardado.

  Para arreglar este problema, se pensó en crear una base de datos en la nube en vez de una local y que todas las personas suban sus productos junto a su familia a la nube, pero, al final, se optó por la base de datos local ya que puede que haya personas que piensen en distintas formas de clasificar los productos.

  En conclusión, para poder mejorar en este aspecto se necesita añadir todos los productos posibles al diccionario inicial.

- **Compatibilidad con iOS:**
  Aunque conseguir que la aplicación esté disponible para iOS es una tarea complicada, en un futuro estaría bien que la aplicación se actualice a una versión disponible para iOS.

- **Nuevos supermercados:**
  Actualmente, PriceList solo está enfocada al supermercado Mercadona pero en un futuro, se adaptará la clase encargada de la lectura de tickets para que sea capaz de leer ticket de nuevos supermercados. Los supermercados que se quieren añadir son los principales en España como: Carrefour, Eroski, Aldi, DIA, etc.

6. Bibliografía y referencias

- ➔ Extract PDF Content with Python
- ➔ Python GUI's With Kivy - YouTube
- ➔ DeepSeek
- ➔ ChatGPT
- ➔ Stack Overflow
- ➔ Inflación en los supermercados
- ➔ Tutorial selector de fecha
- ➔ W3Schools
- ➔ Aprendizaje en las FCT