Grado en Ingeniería Informática

Trabajo Final de Grado

App para reconocimiento de imágenes Version iOS

Autor:
Sergi Estellés Jovaní

Supervisor:
Joan Fabregat Bellido
Tutor académico:
Raúl Montoliu Colás

Fecha de lectura: 28 de Agosto de 2014
Curso académico 2013/2014

Resumen

En esta memoria de Trabajo Final de Grado se detalla el procedimiento seguido para la realización de una aplicación móvil para *iPhone* para reconocimiento de imágenes en un entorno real. En ella, se explica la metodología seguida, así como las diferentes partes de este.

La aplicación, implementada para *iPhone 4* y *iPhone 5*, consiste en una herramienta que permita capturar una fotografía e identificar una imagen contenida dentro de esta para un subconjunto amplio de imágenes (mediante un servidor externo). Para cada imagen reconocida, existe un contenido de diferente tipo (web, vídeo, audio, galería de imágenes,...) que el usuario puede visualizar. Toda esta funcionalidad se implementa sin salir de la aplicación. Para ello ha sido necesario implementar funcionalidades para utilizar cámara, reproductor de audio y vídeo, galería, entre otros, adaptando la interfaz de usuario para que su diseño concuerde con el del resto de la aplicación móvil y sea lo más intuitivo posible.

Palabras clave

Reconocimiento, imágenes, iPhone, iOS, fotografía.

Keywords

Recognizement, images, iPhone, iOS, photography.

Índice general

Índice de figuras

Índice de tablas

Índice de bloques de código

Capítulo 1

Introducción

1.1. Contexto del proyecto

Rubycon Information Technologies es una *Start-Up* de base tecnológica, situada en Castellón de la Plana, cuya principal actividad es desarrollar soluciones tecnológicas para llevar las Tecnologías de la Información a nuevas áreas, a través del diseño de sistemas y aplicaciones que permitan potenciar, agilizar y optimizar los métodos convencionales, tal y como se explica en su página web [27]. A esta empresa se le ha presentado un proyecto que le ha resultado de sumo interés por su potencial, y que han decidido llevar a cabo enfocándolo como un proyecto final de carrera para el Grado en Ingeniería Informática.

El proyecto consiste en crear un sistema que permita, desde un teléfono móvil (de los sistemas operativos *Android* e *iOS*), realizar una foto a una imagen y obtener una información asociada a dicha imagen. Puesto que el alcance del proyecto excedía con creces el alcance de un proyecto final de carrera, se decidió dividir el proyecto en dos. Una parte del proyecto es la que se expone en este documento, mientras que la otra forma parte del proyecto final de carrera de Javier Agut Barreda. Cabe destacar que este ha sido un proyecto desarrollado con metodologías ágiles y, por lo tanto, hay partes de este proyecto que se han realizado de forma conjunta entre ambos estudiantes, utilizando técnicas como el *Pair Programming*, tal y como se detalla más adelante.

Este proyecto(*App para reconocimiento de imágenes Versión iOS*) podemos dividirlo, por lo tanto, en dos partes:

- Creación de una aplicación móvil sobre la plataforma iOS que se comunique con un servidor. Esto es, enviar una fotografía, recibir el contenido asociado y mostrarlo al usuario, almacenando en local los resultados positivos.

- Implementación de un servidor para reconocimiento de imágenes. Esta parte es la realizada en equipo, junto con el estudiante responsable del proyecto *App para reconocimiento de imágenes Versión Android.*

Figura 1.1: En la Figura 1.1 se muestra el diagrama general del funcionamiento del sistema.

El sistema se ha realizado teniendo en mente las posibilidades del producto a obtener. Este tipo de aplicación puede aprovecharse en diversos ámbitos. Museos, librerías, videoclubs, publicidad, entre otros, son algunos ejemplos de posibles aplicaciones que se podrían atribuir a la app desarrollada. Para ello, se ha diseñado el sistema siguiendo el paradigma cliente-servidor [30]. La parte del cliente se ubica en el dispositivo móvil, que realiza peticiones a uno de los servidores de la empresa Rubycon-IT, que atiende dichas peticiones y responde de nuevo al cliente.

1.2. Motivación del proyecto

Vivimos en un mundo totalmente diferente al de hace 10 años. Estamos totalmente conectados a Internet las 24 horas del día. La inmensa mayoría de la sociedad hace uso del *smartphone* para mandar mensajes, compartir fotografías con sus amigos y buscar información, entre otros. La gente vive con el teléfono inteligente en la mano, y esto no ocurría hace unos pocos años.

Los teléfonos inteligentes son una evolución tecnológica muy significativa que viene de los teléfonos móviles y que ofrece un sinfín de oportunidades más, y que debemos explotar. Gracias a los *smartphones*, llevamos en el bolsillo un dispositivo con una gran capacidad de cálculo y

procesamiento, una cámara de fotos que toma fotografías a gran resolución, un reproductor de video en High Definition, además de muchas otras herramientas. Si unimos esto al salto funcional que ha representado el hecho de que la forma de interactuar con este dispositivo sea una pantalla táctil, que facilita en gran medida el tiempo de aprendizaje y el uso de estos dispositivos, nos percatamos de la potencia que tienen estos dispositivos.Y lo mejor es que esta potencia está al alcance de nuestras manos. Podemos aprovecharla.

La motivación del proyecto es precisamente esta. Construir una aplicación que facilite al usuario el acceso a la información a través de su *smartphone* mediante la captura de una imagen. Que haciendo una foto a un cuadro pudieras conocer en qué estaba pensando el autor cuando lo pintó. Que fotografiando la carátula de un DVD pudieras ver el trailer de dicha película. Que tomando una foto de un cartel pudieras saber la localización del restaurante que en él aparece. En una frase, la aplicación que se ha desarrollado es un puente, una autovía, entre el usuario y la información que quiere conocer.

Los teléfonos inteligentes nos brindan la oportunidad idónea y sirven como herramienta portadora de la tecnología a la que nos estamos refiriendo. *Apple* nos ofrece unas características tales como fiabilidad, eficiencia, usabilidad, entre otras, que nos dan pie a escogerlo como sistema operativo en el que desarrollar la aplicación. Además, cabe destacar que el posicionamiento de la marca en el mercado la sitúa como un referente en modernidad, diseño y tecnología, cosa que añade atractivo al hecho de desarrollar la aplicación para esta plataforma. Aunque, al mismo tiempo, también constituye un reto el realizar una aplicación que no desentone en la linea general de excelencia de las aplicaciones de este sistema operativo.

Además, llevar a cabo este proyecto supone dar un pequeño paso introductorio en el mundo del análisis de imágenes, generando *know-how* para la *Rubycon-IT*, y trabajando en equipo en un escenario real.

1.3. Necesidades del proyecto

1.3.1. Antecedentes

Las empresas demandan la interacción entre consumidores y marcas. Existen, desde hace décadas, diversas formas de acceder rápidamente a una información concreta. Todo el mundo conoce los códigos de barras, que utilizan los comercios para enlazar un producto concreto con información referente a este: nombre, descripción, precio. Este método es ampliamente utilizado, aunque con un fin puramente comercial. No es el usuario final el que lee el código para obtener la información.

Figura 1.2: La Figura 1.2 muestra un ejemplo de código de barras y código QR.

Por otra parte, en los últimos años ha aparecido el código QR [31]. Este tipo de códigos sí se han utilizado como una forma que llegar al destinatario de la información, al usuario final. Estos códigos son una tecnología que está integrándose en la sociedad y que, tal vez en el futuro, se utilicen de forma masiva.

1.3.2. Solución

Tanto el código de barras como los QR son estándares. Lo que se busca con este proyecto es crear una alternativa a ellos. Una vía más atractiva para el usuario de acceder a la información. Este novedoso método implica una complejidad y un coste temporal superior al de los códigos anteriormente expuestos. No obstante, se va a trabajar para tratar de minimizar estos puntos flacos y para obtener una herramienta que haga más accesible para el gran público el uso de esta tecnología y mejore el acceso a la información.

Seguramente, el reconocer un objeto dentro de una fotografía sea una tarea demasiado pretenciosa para un proyecto final de grado. No obstante, podemos simplificar el problema reduciéndolo a encontrar una imagen dentro de una fotografía.

Justo esto es lo que se propone en este proyecto. Mediante un dispositivo iPhone, capturar una fotografía que contenga una de las imágenes a las que se le ha asociado cierta información. Esta fotografía, desde el móvil, ha de ser enviada a un servidor. Este se encargará de buscar si en la fotografía aparece alguna de las imágenes registradas en el sistema y, en caso afirmativo, reconocerla, para devolver al dispositivo la información asociada a esta imagen en concreto. Una forma amable y al alcance de cualquier persona de obtener información sobre los productos que nos rodean.

1.4. Objetivos del proyecto

Mediante la realización de este Proyecto Final de Grado, se busca la consecución de diferentes objetivos, de diferente índole. Entre ellos se encuentran:

- **Aprender a programar para el sistema operativo iOS** para poder implementar todas las funcionalidades necesarias para obtener un sistema satisfactorio.

- **Diseño de un interfaz de usuario amigable y útil** para un amplio conjunto de usuarios finales. De otra forma no se conseguirá que la aplicación sea utilizada.
- **Adaptar algoritmos de reconocimiento de imágenes.** No es un objetivo *inventar* un nuevo algoritmo para reconocer imágenes. No obstante, para implementar un servidor capaz de reconocer imágenes es necesario estudiar, conocer, entender y adaptar los algoritmos de reconocimiento de imágenes que más se adecuen a las necesidades del proyecto.
- **Adquirir experiencia real en el trabajo en equipo con metodologías ágiles.** Durante el grado se adquiere una gran cantidad de conocimiento teórico sobre estas materias, pero es fundamental respaldar dicho conocimiento teórico con experiencia. Y en la empresa *Rubycon-IT* se nos brindan las condiciones idóneas para obtenerla.

1.5. Naturaleza del proyecto

En cuanto a la naturaleza del proyecto, cabe destacar que la realización de este amplio proyecto se ha repartido al 50 % entre dos alumnos de Grado en Ingeniería Informática de la Universitat Jaume I. Es decir, el proyecto que la empresa Rubycon IT planteó se ha realizado conjuntamente con entre Javier Agut Barreda y el redactor de esta memoria técnica.

Guiados por el *know-how*[12] de la empresa Rubycon Information Technologies, toda la sección referente al servidor REST[32], reconocimiento de imágenes, base de datos del servidor y diseño de la aplicación ha sido un trabajo realizado codo con codo, siguiendo con la línea de desarrollo ágil inherente a la empresa, destacando el Pair Programming[29] como una técnica que favorece el desarrollo dando como resultado un código de mayor calidad y dando estabilidad al grupo de trabajo, tal y como se detalla en el apartado 4.1. En la Tabla 1.1 se detalla el reparto del proyecto entre los miembros del equipo de desarrollo.

Tabla 1.1: Distribución de las partes del proyecto.

	Javier Agut	Sergi Estellés
Servidor REST	X	X
Base de datos MongoDB	X	X
Base de datos Android	X	
Base de datos iOS		X
Reconocimiento de imágenes	X	X
Reproductores Android	X	
Reproductores iOS		X
Cliente móvil Android	X	
Cliente móvil iOS		X
Diseño interfaces	X	X
Gestión del proyecto	X	X

Capítulo 2

Descripción del proyecto

El proyecto consiste en la realización de una aplicación para reconocer imágenes y mostrar al usuario una información concreta asociada a la imagen reconocida. Dicho sistema se compone, a su vez, de diversos subsistemas. Tal y como se muestra (a grandes rasgos) en la Figura 2.1, el sistema está diseñado, de una parte, siguiendo el paradigma cliente - servidor, teniendo en cuenta que la parte del cliente se ha desarrollado de forma nativa para el sistema operativo iOS.

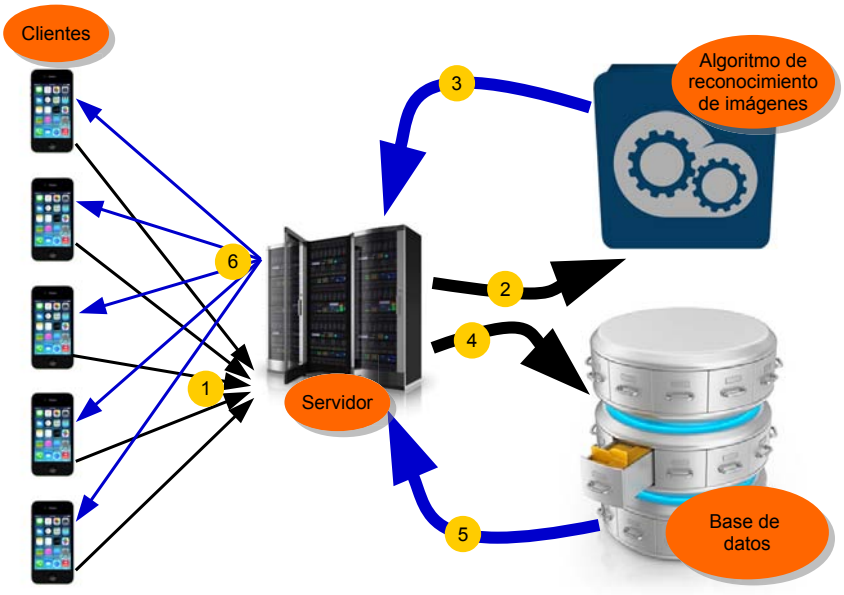


Figura 2.1: La Figura 2.1 muestra un diagrama extendido del funcionamiento del sistema.

El funcionamiento del sistema es el que sigue. Desde el iPhone se realizan peticiones al servidor (1), que recibe la fotografía que contiene la imagen a identificar. El servidor le pide al algoritmo de reconocimiento de imágenes (2) que encuentre la imagen (de las que tiene que reconocer el sistema) de dentro de la fotografía. Este algoritmo devuelve, si es que la encuentra, cuál ha sido la imagen reconocida (3). Cuando el servidor conoce cuál es la imagen

contenida en la fotografía, accede a la base de datos (4) para que esta le devuelva la información asociada a dicha imagen (5). Finalmente, el servidor responde a la petición realizada desde el *iPhone* retornando dicha información (6), para que este la muestre al usuario. En los siguientes subapartados se detalla un nivel más el funcionamiento de cada parte.

2.1. Cliente en el dispositivo móvil

Se ha implementado para dispositivos **iOS** (en Objective-C) la parte cliente del sistema. Desde el dispositivo, y mediante una interfaz sencilla y plana (como aconsejan sobre usabilidad desde Apple actualmente [6]), intentando que el grado de similitud entre dicha interfaz y la interfaz diseñada para la otra parte de la aplicación, en Android, sea el máximo posible (teniendo en cuenta las restricciones de cada lenguaje de programación y de cada sistema operativo) el usuario puede enviar una fotografía al servidor. Más tarde, cuando el servidor devuelve el resultado asociado a la imagen enviada, en el dispositivo se muestra al usuario esta información sin salir de la aplicación. Esto implica la implementación de funcionalidades que utilizan reproductores de vídeo y audio, galería de imágenes y vistas para mostrar una página web. En el apartado 7.1 se detallan los pormenores de ambas implementaciones.

2.2. Base de datos

La base de datos consiste en dos entidades: una para almacenar las imágenes y su contenido asociado, y otra para registrar los accesos que se realizan al servidor. Ambas tablas no están relacionadas entre ellas, pero cabe apuntar que los elementos a guardar en la entidad referente a las imágenes tienen una estructura muy variable, ya que, dependiendo del tipo de información que se quiera devolver al usuario, muchos de sus atributos son totalmente diferentes. Esta es una de las razones por las cuales nos decantamos por utilizar una base de datos no relacional: **MongoDB** [20]. Para ver la información referente a la base de datos al completo, acceda al apartado 7.2.2.

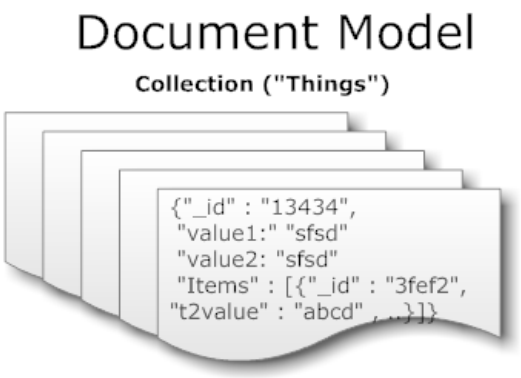


Figura 2.2: La Figura 2.2 muestra la organización que sigue una base de datos no-relacional.

2.3. Servidor

El servidor de este sistema se va a encargar de recibir peticiones con fotografías, ejecutar el reconocimiento de la imagen, con dicho reconocimiento acceder a la base de datos, obtener el contenido que hay que devolver al cliente y devolver el mismo. Teniendo en cuenta esto, se decidió implementar un servidor **REST** [32] en Python. La elección de dicho lenguaje de programación reside en la facilidad del mismo para acoplarse tanto a la base de datos MongoDB, como a las librerías de OpenCV que utilizará el algoritmo de reconocimiento de imágenes.

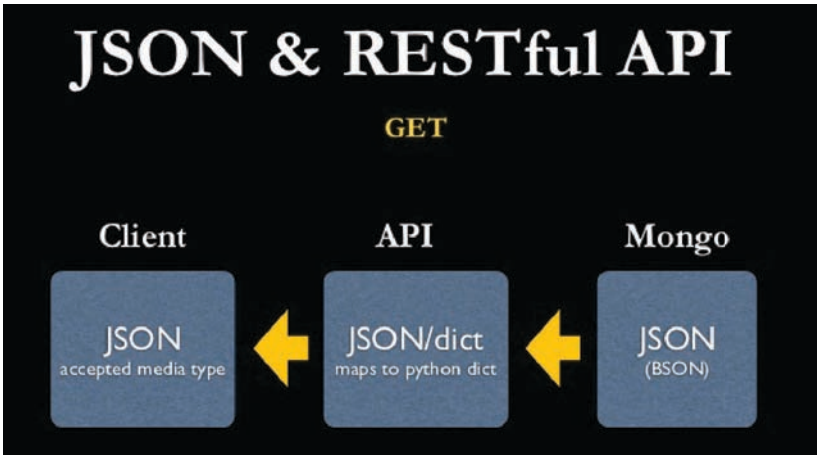


Figura 2.3: La Figura 2.3 muestra la relación que existe entre MongoDB, JSON y servicios REST.

La respuesta devuelta al cliente consiste en un **JSON** en el que se indican los campos necesarios para que el dispositivo móvil cliente pueda interpretar y mostrar la respuesta de la forma deseada. Para conocer la forma concreta en que se realiza dicha implementación y los problemas surgidos en ella, puede consultar el apartado 7.2.

2.4. Algoritmo de reconocimiento de imágenes

Esta parte fue, desde el principio, la que supuso una mayor incertidumbre para el equipo de desarrollo, ya que es la parte menos habitual del proyecto. El paradigma cliente - servidor y la conexión con bases de datos es bastante común en un proyecto de complejidad media, pero el reconocimiento de imágenes es un ámbito mucho más concreto y en el que, además, no teníamos demasiada experiencia. No obstante, contar con el respaldo del INIT (Instituto de Nuevas Tecnologías de la Imagen) y la experiencia de *Rubycon-IT* fueron factores determinantes.

El algoritmo, a grandes rasgos, recibe una fotografía. Dentro de dicha fotografía se incluye una de las imágenes de la base de datos. Dicho de otra forma, la fotografía que recibe no es únicamente la imagen que hay que reconocer, si no que contiene todo lo que la cámara haya captado, como por ejemplo, la pared detrás de la imagen, sombras, luces, entre otros. Por lo tanto, para saber si la fotografía contiene una imagen de las de la base de datos, hay que recorrer todas ellas y, una a una, buscarlas dentro de la fotografía. Para ello, y después de analizar otros

métodos (tal y como se refleja en la sección 3.1), se ha decidido utilizar el detector SURF y el comparador FLANN.

Capítulo 3

Estudio previo

Debido al amplio alcance del proyecto, el apartado de estudio previo ha representado un volumen de trabajo considerable dentro del proyecto. El trabajo referente al estudio del arte abarcó, dedicado exclusivamente a la búsqueda y comprensión de información, los primeros quince días del proyecto, como se indica más adelante, en el apartado Planificación 4. No obstante, con este tiempo se obtuvieron conocimientos con el fin de discernir qué tecnologías se decidía utilizar y conocimientos muy básicos sobre el funcionamiento de dichas tecnologías, que luego fue necesario ampliar, a la hora de ponerse a implementar.

Una de las tareas más importantes dentro del estudio previo fue, de todas las tecnologías que los supervisores de la empresa propusieron para cada uno de los ámbitos del proyecto (servidor, base de datos, frameworks de tratamiento de imágenes, etcétera), dedicir cuál se iba a emplear. En las secciones siguientes se explican los motivos que nos llevaron a decantarnos por unas y no por otras.

3.1. Algoritmos de reconocimiento de imágenes

Observando proyectos o aplicaciones que utilizan algún tipo de reconocimiento de imágenes, se advierte una tendencia, en los últimos tiempos. Prácticamente todos los algoritmos se estructuran en dos partes. Primero hace falta tener una disposición de los datos que nos permita comparar entre dos imágenes. Hay diversas razones que nos impiden, por poner un ejemplo, utilizar la comparación píxel a píxel. Por ejemplo, hay que tener en cuenta que no todas las imágenes tienen el mismo tamaño, y también que la fotografía no contiene únicamente la imagen a reconocer, si no que puede incluir elementos como detalles de la pared, una parte de la mano que sujeta la imagen a reconocer o otros muchos elementos. Las aplicaciones que necesitaban un reconocimiento similar al de este proyecto, que se podría resumir en encontrar una imagen dentro de otra, utilizaban un algoritmo dividido en dos partes.

La primera parte consiste en obtener, de una imagen, los puntos clave, características o descriptores (la terminología para referirse a estos datos es variada). Los procedimientos que se encargan de obtener estos *key points* se denominan detectores y descriptores. De lo que se trata

es de obtener, computacionalmente, una abstracción de la imagen, encontrando los puntos en los que se encuentra alguna característica de la imagen. Esta información se almacena en un array, de modo que, obteniendo los descriptores de dos imágenes, se puede comparar qué descriptores se repiten en ambas, de modo que podemos conocer cómo de parecidas son. Tony Lindenberg, en su artículo *Feature Detection with Automatic Scale Selection* [18], nos habla de la detección de características de una imagen. Como Lindemberg apunta, la obtención de estos descriptores no se ve alterada por la escala o rotación de la imagen, cosa que resulta esencial para el correcto funcionamiento de esta aplicación, ya que las fotografías que los usuarios capturen harán que la imagen a reconocer varíe en tamaño y posición.

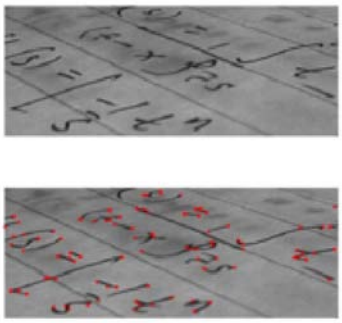


Figura 3.1: En la Figura 3.1 aparece un ejemplo de detección de puntos clave en una imagen.

La segunda parte del algoritmo es la que se encarga de encontrar coincidencias entre los descriptores de dos imágenes. Los procesos que se encargan de realizar dicha tarea reciben el nombre de *matchers* o comparadores. Tienen que realizar una búsqueda muchos a muchos, en los que comprueban si cada descriptor de la imagen que contiene la imagen a reconocer coincide con alguno de los descriptores de la imagen que se quiere encontrar dentro de la fotografía. Por ejemplo, en la figura 3.2, se observa como el *matcher* busca y encuentra algunos puntos clave de la imagen a encontrar (izquierda) comparando con los puntos clave de la fotografía que contiene la imagen a reconocer (derecha).

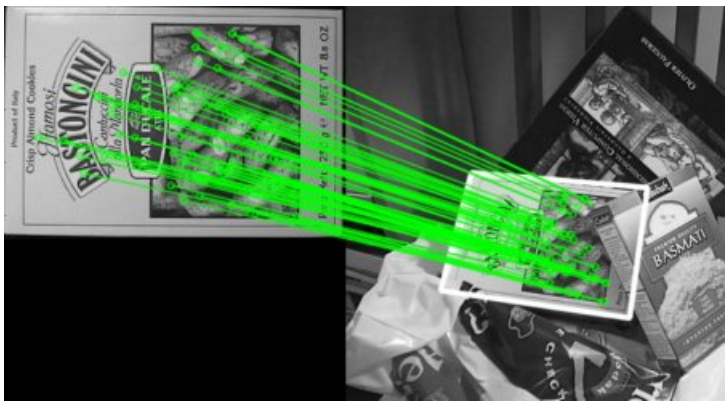


Figura 3.2: Ejemplo de comparación entre puntos clave de dos imágenes, extraído de la página oficial de *OpenCV*.

Una vez comprendida la forma en que se reconoce una imagen dentro de otra, falta encontrar las librerías idóneas para hacerlo. OpenCV [15] nos ofrece diversos algoritmos, tanto para extraer los descriptores de una imagen, como para luego comparar los *key points* de diversas imágenes. Se decide implementar diversos algoritmos para, tras la realización de exhaustivas pruebas, determinar cuál es el que mejor se adapta al proyecto. Para hacerlo, utilizamos un programa llamado *Find-Object* [17]. Puesto que *SURF*[33] es la herramienta más rápida, hemos decidido hacer una batería de pruebas combinando *SURF* como descriptor con diferentes detectores y viceversa. Estos son los resultados obtenidos:

Tabla 3.1: Resultados obtenidos utilizando un procesador Core2Duo a 2,33GHz.

Detector	Descriptor	Núm. caract. encontradas	Tiempo detección caract.(ms)	Tiempo procesamiento descriptores (ms)	Tiempo total(ms)
BRISK	SURF	70	1300	45	1345
SIFT	SURF	130	150	55	205
Star	SURF	93	51	30	81
SURF	**SURF**	**160**	**260**	**140**	**400**
SURF	SIFT	137	250	680	930
SURF	BRISK	140	280	1400	1680
SURF	FREAK	100	270	230	500

Después de leer la documentación y de realizar pruebas con algunos algoritmos (los disponibles en el programa *Find-Object*), se decide que los algoritmos que finalmente se implementen en la aplicación sean :

- Para **Feature description** (detección de características):
  1. SURF (Speeded Up Robust Features)
  2. SIFT (Scale-Invariant Feature Transform)
  3. ORB (Oriented FAST and Rotated BRIEF)
- Para **Feature matching** (comparación de descriptores):
  1. FLANN (Fast Approximate Nearest Neighbor Search Library)
  2. BFMatcher (Brute-Force Matcher)

Puntualizar el hecho de que el detector ORB es compatible únicamente con el comparador BFMatcher, mientras que SIFT y SURF funcionan con FLANN.

3.2. Frameworks para la implementación del servidor REST

En cuanto a la elección de *framework* para implementar los servicios REST, desde la empresa se propusieron varios, que a continuación se analizan, dejando libertad para decidir cuál utilizar. Los *frameworks* propuestos por los expertos de Rubycon-IT son:

- *Flask*
- *Python Eve*
- *Django*
- *NodeJS*
- *Spring*

3.2.1. Flask

Flask [22] es un *microframework* que destaca por su sencillez y su facilidad de uso. Se encuentra bajo la licencia BSD-License [26] y opera para los sistemas de gestión de bases de datos: *PostgreSQL*, *MariaDB*, *MySQL*, *MongoDB*, *CouchDB*, *Oracle* y *SQLite*. Aparte de la cantidad de SGBD con los que es compatible, otro punto a favor es que sigue el paradigma de programación orientado a objetos (que hemos estudiado ampliamente durante el grado) y en el lenguaje de programación *Python*, que también se puede utilizar en la parte de reconocimiento de imágenes.

3.2.2. Python Eve

Python Eve [14] es un *framework* motorizado por *Flask*, *MongoDB* y *Redis*. Mediante *Python*, permite crear servicios web RESTful con todas las funciones de los *frameworks* citados que lo motorizan. Aunque a primera vista parece una opción aconsejable por ser una abstracción de alto nivel que incluye diferentes tecnologías que se van a utilizar, desde la empresa se indica que no supone una reducción sustancial del tiempo de aprendizaje, pero si puede mermar la libertad del programador a la hora de implementar ciertas cosas si escapan de los cánones más habituales.

3.2.3. Django

Django [11] es otro *framework* de desarrollo web *open source* escrito en *Python*, cuya principal meta es facilitar la creación de sitios web complejos, basándose en el principio de desarrollo rápido y de no repetición. Se encuentra, al igual que *Flask*, y también utiliza el paradigma de programación orientado a objetos.

3.2.4. NodeJS

NodeJS [16] no está escrito en *Python*, si no que se basa en el lenguaje *Javascript*. Sigue una arquitectura orientada a eventos, mediante la cual construye programas de red estables, como servidores web. No obstante, también utiliza el paradigma de programación orientada a objetos y la programación funcional y, además de *Javascript*, también emplea *Ruby* como lenguaje de programación. Además, trabaja con los *SGBD* de *MongoDB* y *MySQL*. También es de código abierto y se encuentra bajo la licencia *MIT License*.

3.2.5. Spring

Bajo los lenguajes *Ruby*, *C#*, *Python* y *Java*, *Spring* [24] proporciona un software de desarrollo de aplicaciones y contenedor de inversión de control para la plataforma Java. Es de código abierto y varias extensiones para la conocida plataforma *Java EE* le han proporcionado popularidad hasta el punto de ser considerado una alternativa a *Enterprise JavaBeans* [23]. Se encuentra bajo la licencia *Apache License GPLv2*.

3.2.6. Resumen

En la Tabla 3.2 se resumen las principales características de cada uno de los *frameworks* analizados:

Tabla 3.2: Tabla resumen de las características de los *frameworks* analizados para decidir con cuál implementar el servidor.

	Flask	**Python Eve**	**Django**	**NodeJS**	**Spring**
Licencia	BSD-Licence	BSD-Licence	BSD-Licence	MIT Licence	Apache Licence GPLv2
Bases de Datos	PostgreSQL, MariaDB, MySQL, MongoDB, CouchDB, SQLite	MongoDB, MySQL, NoSQL	PostgreSQL, MySQL, SQLite	MongoDB, MySQL	Relacionales y no relacionales
Paradigmas de programación	Orientado a objetos	Orientado a objetos	Orientado a objetos	Orientado a objetos, orientado a eventos, funcional	Orientado a aspectos
Lenguaje de programación	Python	Python	Python	JavaScript, ruby	Java, ruby, C#, Python

Teniendo en cuenta los datos resumidos en la Tabla 3.2, se advierte que las diferencias entre los distintos *frameworks* no suponen una variación determinante. No obstante, la experiencia de los supervisores de *Rubycon-IT* hizo que la balanza se decantara por utilizar *Flask* para la implementación del servidor.

3.3. Base de datos

Tal y como se ha explicado en la sección 2.2, la base de datos que necesitamos no tiene una complejidad demasiado alta en lo que al número de tablas y relaciones entre ellas se refiere, pero sí en la diversidad de los campos de una misma tabla. Existen diferentes tipos de información que se devolverá al cliente (vídeo, audio, galería, web,...) y esto nos obliga a contar con campos muy diversos. Se han analizado 3 *SGBD* con el fin de obtener una idea aproximada de cuál es más beneficioso para nuestro sistema.

Además de esta base de datos, que reside en el servidor, se implementará otra utilizando *Core Data* (el *SGBD* de *Apple*) para almacenar, dentro del dispositivo, los resultados favoritos.

3.3.1. SQLite

De aprendizaje y uso rápido, y aunque es muy útil para realizar test, empiezan a aparecer problemas cuando la base de datos empieza a escalarse. Esta es una base de datos relacional interna. Si se almacenan los datos en la memoria del teléfono, disminuyen las peticiones al servidor para obtener información, pero disminuye la memoria del teléfono.

3.3.2. MySQL

Su curva de aprendizaje es bastante menos pronunciada que la de *SQLite*, pero, en contraposición, cuenta con muchas más opciones para personalizar la base de datos, cosa que ocasiona que la base de datos se pueda escalar adecuadamente. Entre otros, cuenta con gestión de usuarios y permisos. También es una base de datos relacional, como *SQLite*, pero se almacena en el servidor, aligerando así la memoria utilizada en el dispositivo, pero aumentando el coste de comunicaciones entre el cliente y el servidor.

3.3.3. MongoDB

MongoDB es un Sistema de Gestión de Base de Datos *noSQL*. No contiene tablas, si no colecciones. Los diferentes elementos contenidos en una colección pueden tener diferentes campos, cosa que nos podría beneficiar a la hora de extender funcionalidad en la aplicación. Otra ventaja con la que cuenta MongoDB es que trabaja con JSON, que es el formato que se utiliza para el intercambio de mensajes entre cliente y servidor.

3.3.4. Resumen

Tras analizar los diferentes *SGBD*, cabe destacar que el diseño de la aplicación difiere ampliamente dependiendo de si se utiliza uno u otro. Tras discutirlo con el equipo de la empresa, se decide que, también con el objetivo de aprender a trabajar con una tecnología con la que no hemos utilizado, el Sistema de Gestión de Base de Datos sea *MongoDB*.

Tabla 3.3: Tabla resumen de las características de los *SGBD* analizados para decidir con cuál implementar la base de datos.

	SQLite	**MySQL**	**MongoDB**
Almacenamiento	Dispositivo	Servidor	Servidor
Tipo	Relacional	Relacional	No relacional
Configurable	No	Sí	Sí

3.4. Dónde realizar el análisis de la imagen

El reconocimiento de imágenes es la parte más importante del proyecto, y las demás giran alrededor de ella. Por lo tanto, es de vital importancia determinar, lo más temprano posible, si el reconocimiento se iba a realizar en el mismo dispositivo móvil o en el servidor. Es por esto que esta cuestión se aborda en este apartado de estudio previo.

A priori, ambas alternativas cuentan con ventajas y puntos flacos. Si se realiza el reconocimiento de la imagen en el mismo *iPhone*, se ahorra el coste en tiempos de enviar la imagen desde el dispositivo al servidor. Pero hay que consumir recursos del *smartphone*, tanto computacionales como de memoria.

Por otra parte, el realizar el reconocimiento de la imagen en el servidor tiene como desventaja que es necesario enviar la fotografía desde el móvil a el servidor. No obstante, esto no puede significar un gasto exagerado, ya que el enviar imágenes desde un dispositivo es una funcionalidad que miles de aplicaciones utilizan.

Figura 3.3: En la Figura 3.3 aparecen los procesadores *iPhone* para las últimas generaciones.

No obstante, al tratar las imágenes en el servidor, podemos controlar la potencia de la que se dispone para dicho procesamiento. Y es que, aún sin salir de los *iPhone*, no se sabe hacia donde puede evolucionar la tecnología de procesadores. Además, de esta forma, la aplicación tardaría, más o menos, el mismo tiempo en reconocer la imagen independientemente de si se está ejecutando la aplicación en un *iPhone 3* o en un *iPhone 5s*, ya que solamente variaría el tiempo que se tarda en mandar la imagen. Y, esto último, depende mucho más de la señal de *Wi-Fi* o de *3g*, que de la potencia del dispositivo en sí.

3.5. El sistema *iOS*

Apple ha desarrollado un sistema operativo movil propio: *iOS* [9]. Este sistema está escrito en *Objective-C* [10] y las aplicaciones que se implementan en nativo para este sistema operativo también se codifican utilizando este lenguaje. Para desarrollar aplicaciones para *iOS*, existen herramientas de diseño de interfaces propias y patrones de programación propios, tal y como se explica en los siguientes apartados.

Figura 3.4: En la Figura 3.4 se muestra la evolución de la pantalla de inicio de *iOS*.

Dentro de las primeras semanas dedicadas al estudio del estado del arte, dediqué bastante tiempo a conocer cómo funciona *iOS*, cómo se estructuran las aplicaciones y cómo utilizar el Xcode IDE. Para ello, el Jefe de Desarrollo de *iOS* de la empresa transmitió las nociones más básicas y proporcionó una serie de tutoriales disponibles *online* para empezar a familiarizarse con todas estas cosas que, para un estudiante de cuarto de Grado en Ingeniería Informática, son totalmente nuevas y desconocidas. En las siguientes secciones se explican las ideas más interesantes que aprendí en dicha experiencia. Aunque una vez terminado el periodo de estudio previo se conocen las características básicas del lenguaje y se pueden empezar a desarrollar aplicaciones, es cierto que los conocimientos sobre el sistema se han ido ampliando a lo largo de todo el proyecto.

3.5.1. *Storyboard*

En *iOS 5* se introdujo el sistema de *Storyboard*. El *storyboard* es una herramienta que ahorra mucho tiempo al desarrollador en lo que a diseño de interfaces de usuario se refiere, si se sabe

utilizar adecuadamente. Mediante él, de un sólo vistazo, se pueden comprobar las diferentes bifurcaciones que puede tomar la interfaz, los saltos entre ventanas, y el aspecto de cada una de ellas. Gráficamente, es posible añadir componentes (*labels*, botones, *views*,...) y modificar sus características, así como especificar qué botones abren otras vistas, tablas en las que organizar la información, y un sinfín de cosas más. Mucha información muy compacta.

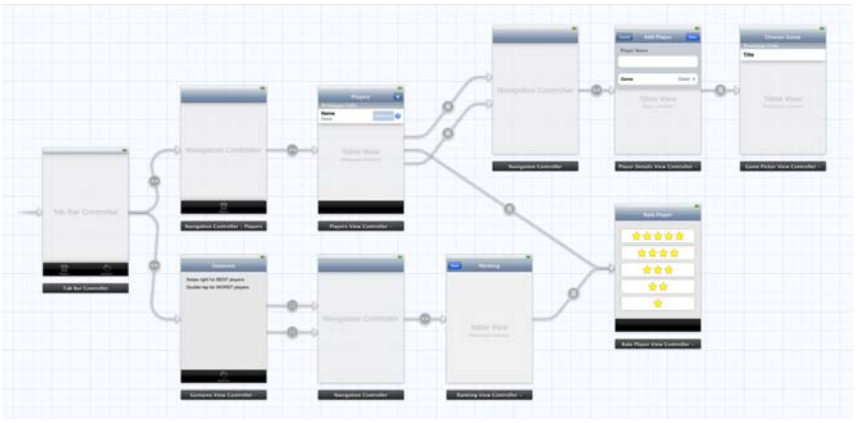


Figura 3.5: La figura 3.5 muestra un ejemplo de *storyboard*.

No obstante, el *storyboard* también tiene algunos puntos débiles, de los que hay que ser consciente. Por ejemplo, es poco útil cuando tu aplicación tiene un número de vistas que se cuente por docenas (como en la Figura 3.6), o cuando necesitas realizar acciones muy complejas en el salto de una *view* a otra, como, por ejemplo, realizar una fotografía.

Figura 3.6: La figura 3.6 muestra un mal ejemplo de *storyboard*.

3.5.2. *AFNetworking*

Una de las cosas que más habituales son en una aplicación móvil es, por ejemplo, de una lista, ir cargando las aplicaciones de forma asíncrona, sin que la vista que muestra la lista deje de ser responsiva, sin que se bloquee mientras carga. *Apple* ha desarrollado un par de librerías propias que cubren esta necesidad bastante bien (de las que destaca *Grand Central Dispatch* [7]), aunque lo cierto es que existe una librería de terceros, llamada *AFNetworking* [28], que cuenta con un mayor número de opciones configurables y mayor fiabilidad, por lo que, aunque sea externa a *iOS*, es la que se utiliza en esta aplicación.

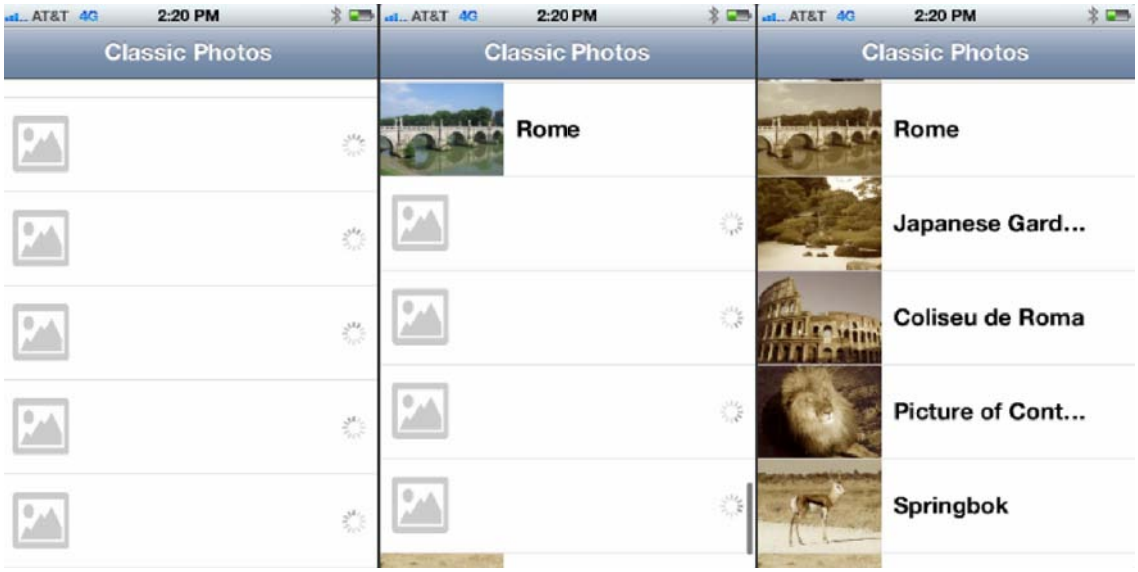


Figura 3.7: La Figura 3.7 es un ejemplo de lista reactiva mientras carga imágenes.

Esta librería se puede utilizar para diversas funcionalidades que se requieren en este proyecto, como son:

- Descargar imágenes desde una *url*.
- Descargar vídeo desde una *url*.
- Descargar audio desde una *url*.
- Realizar peticiones a un servidor REST y obtener una respuesta.

Y además, ayuda a gestionar errores en las peticiones, si el contenido no se encuentra disponible en el momento de realizar la petición, etcétera.

3.5.3. *Blocks*

Los *Blocks* [4] son una extensión bastante potente de los lenguajes *C* y *Objective-C*, añadida por *Apple*. Permiten al programador mantener varias lineas de código como unidades autónomas,

decidiendo el orden en el que se van a ejecutar y pudiendo pasarlas como argumentos a otras funciones, como si de funciones *lambda* se tratara. En un primer instante, los *blocks* fueron creados para dotar de una mayor potencia a la *Grand Central Dispatch* [7], de la que ya se ha hablado en este documento, en la sección 3.5.2. Pero, cuando los desarrolladores vieron las posibilidades que ofrecía, poco a poco los fueron utilizando en código de todo tipo.

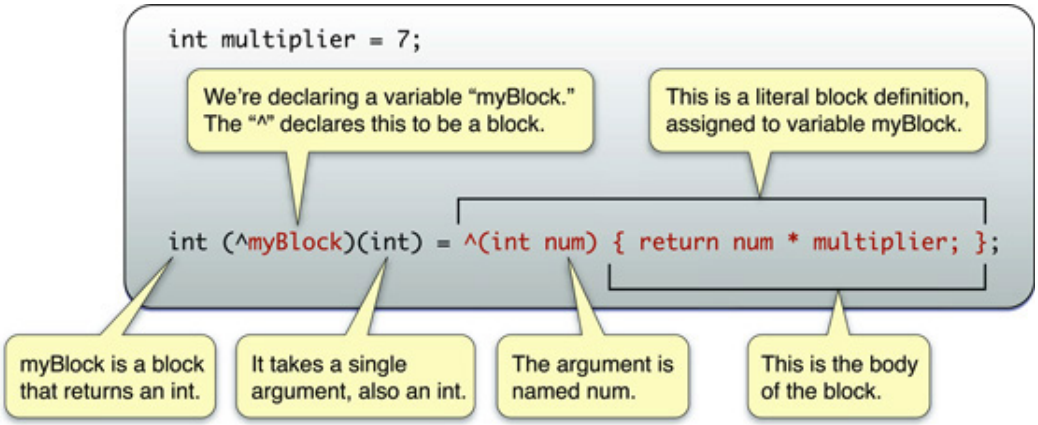


Figura 3.8: En la Figura 3.8 se muestra la sintaxis de los *blocks*.

Siguiendo la sintaxis que aparece en la Figura 3.8, especificamos los parámetros que toma el *block*, los que devuelve y el cuerpo (el código que encapsula). De esta forma, podemos pasar este código a otra función, proporcionando así al lenguaje la posibilidad de dar un salto cualitativo para ponerse al nivel de otros lenguajes que utilizan *lambdas* [19], como pueden ser *Python*, *C#* o *Java 8*.

3.5.4. Delegados

Un delegado es un objeto [5] que responde a un evento en un programa. Es un objeto que actúa en coordinación con otro objeto cuando este encuentran un evento producido por un usuario. Así es como funciona *iOS* por dentro.

No obstante, en el desarrollo de este proyecto, los delegados son objetos que satisfacen una funcionalidad por defecto y que, si quieres adaptarla para que se comporten de una forma determinada, es necesario sobreescribir sus métodos. De esta forma, en esta aplicación se sobreescriben métodos de los delegados de:

- *UIGestureRecognizerDelegate*: para detectar gestos del usuario sobre la pantalla.
- *UITabBarDelegate*: para implementar el cambio de vistas como pestañas.
- *UIImagePickerControllerDelegate*: para tomar fotografías y acceder a la galería del dispositivo.
- *UITableViewDelegate*: para mostrar ítems en forma de lista.

- *UITableViewDataSource*: para establecer una clase como fuente de datos de una estructura.
- *AVAudioPlayerDelegate*: para reproducir audio.
- *UIWebViewDelegate*: para visualizar páginas web.

Capítulo 4

Planificación

En este capítulo se expone la estimación que se realizó en la fase inicial del proyecto. Su objetivo es definir los costes temporales y los recursos materiales y humanos necesarios para concluir el proyecto correctamente, satisfaciendo las necesidades especificadas en la toma de requisitos, y en el tiempo establecido.

4.1. Metodología

Como ya se ha explicado en el apartado 1.5, una parte de este proyecto (la referente a la implementación del servidor) se realiza conjuntamente entre dos alumnos, mientras que las demás partes se realizan de forma individual. Este hecho hace, si cabe, más necesario, establecer la metodología de trabajo que se va a utilizar.

Guiados por la forma de trabajar de la empresa *Rubycon-IT*, se ha llevado a cabo el proyecto siguiendo la metodología ágil ***SCRUM***. Y esto se debe a que, siguiendo la filosofía *Lean Startup* [25], las metodologías tradicionales no se adaptan en absoluto a las necesidades cambiantes del mercado, y más aún en la situación económica global en la que nos encontramos actualmente. *Lean Startup* busca eliminar prácticas ineficientes y incrementar el valor de la producción durante la fase de desarrollo. De esta forma, en esta empresa se busca desarrollar, en cada fase del proyecto, el producto mínimo viable. Esto es, desarrollar en el mínimo código posible, una funcionalidad básica, pero que funcione. Es decir, que al terminar una fase del proyecto, se obtenga un producto que el cliente ya pueda utilizar y del que el cliente pueda devolver a la empresa un *feedback* para mejorar los aspectos que sea necesario mejorar. Esto se debe a que, si en un momento determinado, el cliente decide que no puede dedicar más capital al desarrollo de la aplicación que ha contratado, se detiene el transcurso del proyecto, pero al menos el cliente recibe un producto que puede utilizar, a cambio del dinero que ha aportado hasta el momento.

En lo que a la planificación del proyecto se refiere, esto se traduce en que el proyecto se va a dividir en *sprints*. Habrá *sprints* que se dedicarán a trabajo individual, otros se dedicarán a trabajo en equipo, y otros contendrán tareas de los dos tipos.

4.1.1. SCRUM

SCRUM es una metodología que se basa en:

- Hacer grupos pequeños, interdisciplinares y auto-organizados.
- Dividir el trabajo en entregables concretos y estimar cada parte.
- Dividir el tiempo en iteraciones cortas.
- Optimizar el proceso.

Dentro de esta metodología, se establecen varios roles con los que se identifican los diferentes miembros del equipo. Por una parte, el propietario del producto es quien establece la visión del producto y las prioridades. Por otra, el equipo de desarrollo se ocupa de implementar el producto. Por último, es *Scrum Master* es una persona que se encarga de liderar el proceso y solucionar los problemas que surjan en el mismo.

En Scrum se definen unas iteraciones de tiempo que deben ser, por regla general, constantes. Al inicio de cada iteración, el equipo selecciona un subconjunto de la pila de producto (todas las tareas a realizar para que el producto resultante esté completo) teniendo en cuenta la prioridad del propietario del producto y el tiempo que requerirá. Al final de cada iteración, el equipo muestra cómo opera la funcionalidad implementada en el mismo y reflexiona sobre el proceso.

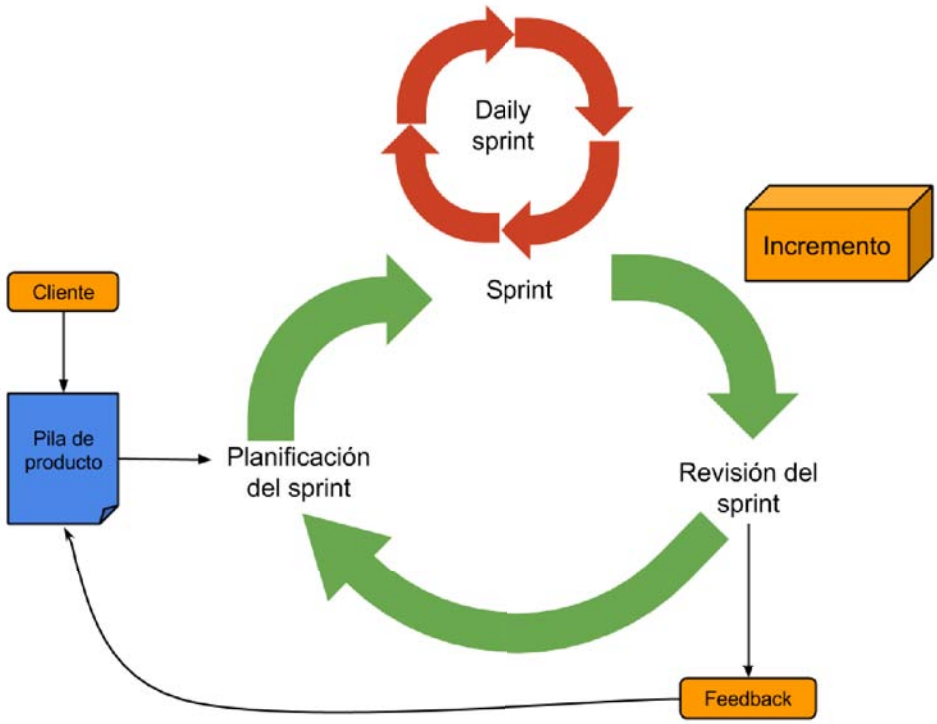


Figura 4.1: La Figura 4.1 es un diagrama sobre las iteraciones de *SCRUM*.

4.1.2. SCRUM adaptado

Tras la incorporación de los miembros del equipo en la empresa *Rubycon Information Technologies*, se indicó que la metodología a seguir para la realización del proyecto sería la metodología ágil *SCRUM*. No obstante, desde el primer momento, se explicó que las metodologías ágiles no están diseñadas como un conjunto de mandamientos que hay que seguir al pie de la letra, si no que, la utilidad de dichas metodologías reside en la capacidad de adaptación de las mismas al proyecto y a las circunstancias concretas. Es por ello que, siempre aconsejados por los supervisores experimentados en la materia, se han permitido ciertas licencias o relajación de ciertas normas. Por poner un ejemplo, algunos *sprints* han resultado en la práctica uno o dos días más de lo que se refleja en la planificación, debido a la ausencia de alguno de los supervisores el día del cierre del sprint o situaciones similares.

No obstante, sí que se han seguido la mayoría de indicaciones recogidas en la metodología *SCRUM*. Y es que, de otra forma, hubiera sido imposible realizar este proyecto de la forma en que ha transcurrido. Entre los aspectos más relevantes, cabe destacar la distribución de las historias de usuario y tareas en *sprints* y las continuas reuniones, tanto entre los miembros del equipo como entre el equipo y el cliente.

Descomposición de las historias de usuario en Sprints

Al principio del proyecto se realizó una reunión con los clientes. De ella, se extrajeron los requisitos del sistema y las historias de usuario, que se convirtieron en el *Backlog* de la aplicación. Una vez establecido exactamente el *Backlog* y las pruebas de aceptación que validan la correcta implementación de las historias de usuario, se pudo hacer una primera distribución de las tareas a realizar entre los diferentes *sprints*, sin perder de vista que, al finalizar cada *sprint*, se revisa tanto el estado (finalizado o no) de las tareas de dicho *sprint*, como qué tareas se han seleccionado para el siguiente. Tras este análisis, se pueden cambiar las tareas asignadas a cada *sprint*.

Reuniones

Las reuniones son fundamentales en un proyecto *SCRUM*. Es verdad que suele haber menos documentación, pero para compensar hay que llevar a cabo una comunicación efectiva entre todo el equipo. Esto se consigue mediante las reuniones:

Daily Scrum Cada día, al comenzar la jornada, los miembros del equipo se reúnen para recordar el estado de las tareas al finalizar el día anterior y para revisar si la planificación del *sprint* se está cumpliendo. De esta forma, cada miembro del equipo es consciente del estado del proyecto y se pueden redistribuir los esfuerzos para que el proyecto avance de una forma controlada.

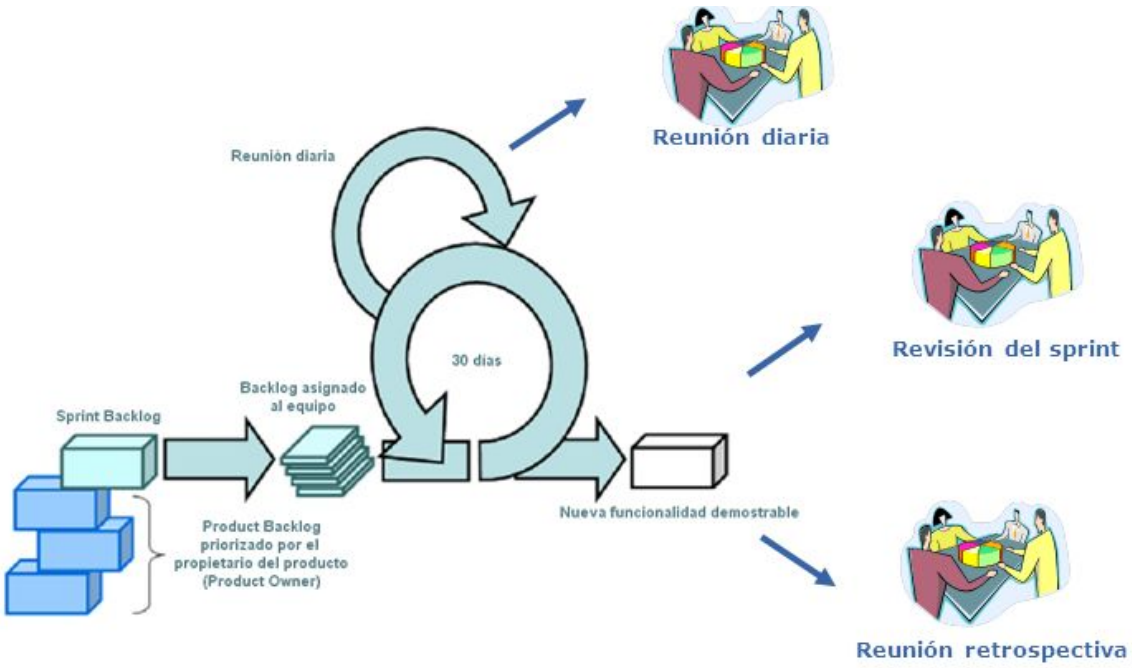


Figura 4.2: La Figura 4.2 es un diagrama en el que aparece cuándo se realizan reuniones en *SCRUM*.

Planificación del *sprint* Son reuniones en las que se detalla de qué forma va a transcurrir el siguiente *sprint*. En ella, se seleccionan las historias de usuario de la pila de producto o *backlog* que se van a implementar, cuanto va a durar la implementación de cada una y qué pasos o tareas hay que seguir.

Revisión del sprint A la finalización de cada *sprint*, el equipo se reúne con el cliente para comprobar que las pruebas de aceptación se cumplen y que el resultado del trabajo es satisfactorio para el cliente. En el mejor de los casos, se mencionan detalles que se modificarán durante el siguiente *sprint* para acercar el resultado lo máximo posible a lo que el cliente tiene en mente.

Retrospectiva del sprint Este tipo de reuniones se realizan a continuación de las de revisión del *sprint*. Su objetivo es la reflexión sobre el proceso realizado para mejorar la productividad del equipo y la calidad del producto desarrollado. Entre otras cosas, se analiza por qué o por qué no se están cumpliendo los plazos, o si el incremento de producto ha resultado ser el que el cliente esperaba o no.

4.2. Planificación temporal

4.2.1. Definición de tareas

Para poder estimar el tiempo necesario para la realización del proyecto, es necesario dividirlo en historias de usuario y, si estas no son lo suficientemente pequeñas como para poder estimarlas, es necesario dividirlas en tareas. De los requisitos que se extrajeron de la reunión con el cliente, tal y como se explica en la sección 5.1, se definieron las siguientes historias de usuario:

Tabla 4.1: Historias de usuario del proyecto.

Ident.	**Historia**
HU01	[D] Familiarizarse con el entorno de trabajo.
HU02	[D] Conceptos de programación móvil.
HU03	[D] Investigación tecnologías servidor.
HU04	[D] Investigación algoritmos reconocimiento de imágenes.
HU05	[U] Realizar una fotografía con mi dispositivo móvil.
HU06	[U] Enviar la imagen al servidor.
HU07	[U] Recibir la imagen en el servidor.
HU08	[D] Cotejar imagen con la base de datos.
HU09	[U] Ver tutorial al iniciar la aplicación por primera vez, pudiéndolo omitir.
HU10	[U] Acceder al tutorial en cualquier momento.
HU11	[U] Ver el contenido multimedia sin salir de la aplicación.
HU12	[U] En caso de error al enviar una imagen, se me permite reintentar o volver a empezar.
HU13	[U] Compartir el resultado de la búsqueda en Facebook y Twitter.
HU14	[U] Asignar una puntuación a los resultados obtenidos.
HU15	[U] Marcar como favorito un resultado obtenido.
HU16	[U] Acceder a mis favoritos.
HU17	[U] Apartado de ?acerca de? donde aparezcan datos sobre la empresa.
HU18	[U] Apartado de licencias se especifiquen las licencias utilizadas para el desarrollo de la aplicación.
HU19	[U] Navegar por la aplicación mediante un *action bar*.
HU20	[A] Añadir imágenes con un contenido asociado al sistema.
HU21	[A] Añadir campañas al sistema a las que asignar imágenes.
HU22	[C] Almacenar las fotos realizadas, la imagen a la que corresponden y desde que móvil se han enviado.
HU23	[C] Integrar la aplicación con Google Analytics.
HU24	[E] Redacción del manual de usuario.
HU25	[E] Redacción de la Memoria.
HU26	[E] Preparación de la presentación.

En la Tabla 4.1, las historias tienen una clave entre corchetes. Estas claves se refieren al actor que requiere la historia de usuario:

- [**D**]: como desarrollador. Es quien implementa el código que genera el sistema que satisface ciertas necesidades.

- [**U**]: como usuario final. Es quien hará uso de la aplicación para obtener beneficio propio.

- [**A**]: como administrador del sistema. Es quien, cuando el sistema esté en funcionamiento, añadirá nuevas imágenes a la base de datos y nuevo contenido asociado a dichas imágenes para que los usuarios finales puedan reconocerlas.

- [**C**]: como cliente. Es el propietario del *software* resultante del proceso.

- [**E**]: como estudiante. En este caso, la misma persona es al mismo tiempo estudiante y desarrollador.

T	Clave	Resumen	Informador	Pr	Estado
	PFCARI-1	[D] Familiarizarse con el entorno de trabajo.	Sergi Estelles	↑	RESUELTA
	PFCARI-2	[D] Conceptos de programación móvil.	Sergi Estelles	↑	RESUELTA
	PFCARI-3	[D] Investigación tecnologías servidor.	Sergi Estelles	↑	RESUELTA
	PFCARI-4	[D] Investigación algortimos reconocimiento de imágenes.	Sergi Estelles	↑	CERRADA
	PFCARI-5	[U] Realizar una fotografía con mi dispositivo móvil.	Sergi Estelles	↑	RESUELTA
	PFCARI-6	[U] Enviar la imagen al servidor.	Sergi Estelles	↑	RESUELTA
	PFCARI-7	[U] Recibir la imagen en el servidor.	Sergi Estelles	↑	CERRADA

Figura 4.3: La Figura 4.3 es una captura de las historias de usuario en el *Jira*.

Algunas de estas historias de usuario son demasiado extensas (como por ejemplo, cotejar una imagen con la base de datos), y es verdaderamente complicado estimar el tiempo que se va a necesitar para llevarlas a cabo. Por lo tanto, estas historias hay que dividirlas en tareas. También se dividen en tareas las historias que se refieren (en el proyecto general de la empresa) a la implementación de la misma funcionalidad para *Android* y para *iOS*. En este documento no aparecen las tareas referentes al sistema operativo *Android* por no pertenecer a este proyecto.

Tabla 4.2: División en tareas de las historias de usuario del proyecto.

Ident.	Historia	Tarea
TA01	**HU01**	Aprender funcionamiento del Confluence y Jira.
TA03	**HU01**	Aprender funcionamiento XCode.
TA05	**HU02**	Estudio fundamentos iOS.
-	**HU03**	No es necesario fragmentar esta historia de usuario.
-	**HU04**	No es necesario fragmentar esta historia de usuario.
TA09	**HU05**	Implementar cámara iOS.
TA11	**HU06**	Realizar petición iOS.
TA12	**HU07**	Implementar servicio REST para recibir una imagen.
TA13	**HU08**	Construir base de datos.
TA14	**HU08**	Implementar algoritmo para obtener los descriptores de la imagen.
TA15	**HU08**	Implementar algoritmo para matchear los descriptores contra la base de datos.
-	**HU09**	No es necesario fragmentar esta historia de usuario.
-	**HU10**	No es necesario fragmentar esta historia de usuario.
TA19	**HU11**	Implementar reproductor de video iOS.
TA21	**HU11**	Implementar reproductor de audio iOS.
TA23	**HU11**	Implementar galería iOS.
-	**HU12**	No es necesario fragmentar esta historia de usuario.
TA26	**HU13**	Implementar función compartir Facebook iOS.
TA28	**HU13**	Implementar función compartir Twitter iOS.
TA29	**HU14**	Implementar servicio puntuación y almacenar en la base de datos.
TA31	**HU15**	Crear base de datos interna iOS.
TA33	**HU15**	Añadir funcionalidad favoritos a la interfaz iOS.
-	**HU16**	No es necesario fragmentar esta historia de usuario.
-	**HU17**	No es necesario fragmentar esta historia de usuario.
-	**HU18**	No es necesario fragmentar esta historia de usuario.
-	**HU19**	No es necesario fragmentar esta historia de usuario.
-	**HU20**	No es necesario fragmentar esta historia de usuario.
-	**HU21**	No es necesario fragmentar esta historia de usuario.
-	**HU22**	No es necesario fragmentar esta historia de usuario.
-	**HU23**	No es necesario fragmentar esta historia de usuario.

4.2.2. Estimación temporal

Antes de la reunión inicial con los clientes, durante los primeros días de la estancia en la empresa, se realizó un diagrama de *Gantt* para distribuir las tareas entre el periodo que dura la estancia. No obstante, puesto que el proyecto se ha llevado a cabo siguiendo la metodología *Scrum*, simplemente ha servido como orientación.

Nombre de tarea	Duración	Comienzo	Fin
Proyecto App Reconocimiento de imágenes	**100 días**	**lun 03/03/14**	**vie 18/07/14**
1º Sprint	**10 días**	**lun 03/03/14**	**vie 14/03/14**
Estudio previo	**10 días**	**lun 03/03/14**	**vie 14/03/14**
Conceptos de programación en iOS	10 días	lun 03/03/14	vie 14/03/14
Conceptos de programación en Android	10 días	lun 03/03/14	vie 14/03/14
Comparación algoritmos reconocimiento de imágenes	10 días	lun 03/03/14	vie 14/03/14
Estudio de tecnologias para implementar servicios REST	10 días	lun 03/03/14	vie 14/03/14
Definición de la pila de producto	3 días	mié 12/03/14	vie 14/03/14
Estimación de la pila de producto	3 días	mié 12/03/14	vie 14/03/14
Entrega propuesta técnica borrador	0 días	vie 14/03/14	vie 14/03/14
2º Sprint	**15 días**	**lun 17/03/14**	**vie 04/04/14**
Desarrollo cliente beta (Android)	8 días	lun 17/03/14	mié 26/03/14
Desarrollo cliente beta (iOS)	8 días	lun 17/03/14	mié 26/03/14
Primera versión del servicio REST	7 días	jue 27/03/14	vie 04/04/14
Entrega propuesta técnica definitiva	0 días	vie 04/04/14	vie 04/04/14
Pruebas de aceptación con el cliente	0 días	vie 04/04/14	vie 04/04/14
3º Sprint	**15 días**	**lun 07/04/14**	**vie 25/04/14**
Comunicación cliente Android-servidor	15 días	lun 07/04/14	vie 25/04/14
Comunicación cliente iOS-servidor	15 días	lun 07/04/14	vie 25/04/14
Implementación base de datos	7 días	lun 07/04/14	mar 15/04/14
Rectificar indicaciones cliente anterior Sprint	4 días	lun 07/04/14	jue 10/04/14
Pruebas de aceptación con el cliente	0 días	vie 25/04/14	vie 25/04/14
4º Sprint	**15 días**	**lun 28/04/14**	**vie 16/05/14**
Algoritmo de reconocimiento de imágenes	15 días	lun 28/04/14	vie 16/05/14
Rectificar indicaciones cliente anterior Sprint	4 días	lun 28/04/14	jue 01/05/14
Pruebas de aceptación con el cliente	0 días	vie 16/05/14	vie 16/05/14
5º Sprint	**15 días**	**lun 19/05/14**	**vie 06/06/14**
Integración del sistema	15 días	lun 19/05/14	vie 06/06/14
Rectificar indicaciones cliente anterior Sprint	4 días	lun 19/05/14	jue 22/05/14
Pruebas de aceptación con el cliente	0 días	vie 06/06/14	vie 06/06/14
6º Sprint	**15 días**	**vie 06/06/14**	**vie 27/06/14**
Rectificar indicaciones cliente anterior Sprint	8 días	lun 09/06/14	mié 18/06/14
Inicialización del servidor y base de datos	8 días	lun 09/06/14	mié 18/06/14
Testeo	7 días	jue 19/06/14	vie 27/06/14
Pruebas de aceptación con el cliente	0 días	vie 06/06/14	vie 06/06/14
7º Sprint	**15 días**	**lun 30/06/14**	**vie 18/07/14**
Redacción del manual de usuario	2 días	lun 30/06/14	mar 01/07/14
Redacción del trabajo final de grado	12 días	lun 30/06/14	mar 15/07/14
Preparación de la presentación	3 días	mié 16/07/14	vie 18/07/14

Figura 4.4: La Figura 4.4 muestra las tareas del proyecto general de la empresa antes de la reunión con los clientes.

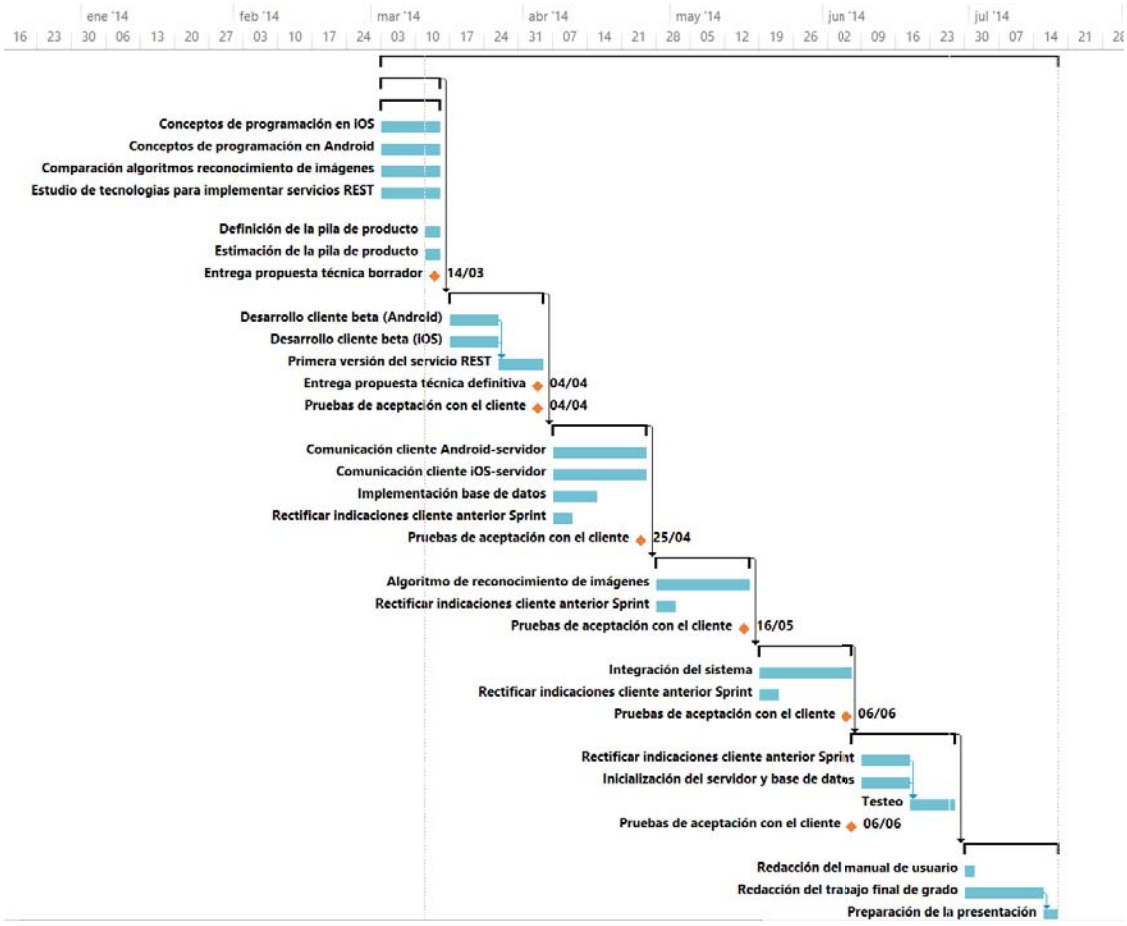


Figura 4.5: La Figura 4.5 es el diagrama de *Gantt* del proyecto general de la empresa antes de la reunión con los clientes.

No obstante, como ya he dicho, este diagrama de *Gantt* solamente sirvió como una forma de ver el proyecto en global, ya que siguiente una metodología *SCRUM*, los *sprints* fueron variando dependiendo de circunstancias puntuales. No obstante, no se puede decir que el realizar el diagrama de Gantt no haya servido para nada, ya que, al hacerlo, se puede controlar, *sprint* a *sprint*, que el proyecto no tenga desviaciones temporales demasiado acusadas y que se vaya a terminar más o menos en el tiempo establecido.

Planificación temporal con SCRUM

Para planificar el desarrollo del proyecto mediante *SCRUM*, es necesario establecer los puntos de historia de cada historia de usuario, la duración de los *sprints* y la velocidad del equipo.

Puntos de historia. En *Rubycon-IT* se estimas las historias de usuario según puntos de historia. Cada punto de historia equivale a 2 horas de trabajo. Teniendo esto en mente, se asigna a cada historia de usuario la puntuación que se piensa que se va a requerir. Puesto que el proyecto general se llevó a cabo en equipo, se estimaron las historias de usuario siguiendo la

técnica de *Planning Poker* [21]. Esta consiste en los siguientes pasos:

1. Se selecciona una historia de usuario.
2. Cada miembro del equipo saca una carta que representa los puntos de historia que le atribuye a dicha historia de usuario.
3. Si hay diferencia entre las estimaciones de los miembros del equipo, cada uno expone el porqué de su puntuación y se vuelve a puntuar hasta que esta diferencia se reduzca.
4. Cuando no hay diferencia o es mínima, se atribuye a la historia de usuario dicho valor o una media de ellos y se pasa a la siguiente historia.

Tabla 4.3: Estimación en puntos de historia de las historias de usuario del proyecto.

Id.	**Historia**	**Puntos**
HU01	[D] Familiarizarse con el entorno de trabajo.	8
HU02	[D] Conceptos de programación móvil.	8
HU03	[D] Investigación tecnologías servidor.	4
HU04	[D] Investigación algoritmos reconocimiento de imágenes.	8
HU05	[U] Realizar una fotografía con mi dispositivo móvil.	10
HU06	[U] Enviar la imagen al servidor.	5
HU07	[U] Recibir la imagen en el servidor.	5
HU08	[D] Cotejar imagen con la base de datos.	40
HU09	[U] Ver tutorial al iniciar la aplicación por primera vez, pudiéndolo omitir.	5
HU10	[U] Acceder al tutorial en cualquier momento.	1
HU11	[U] Ver el contenido multimedia sin salir de la aplicación.	16
HU12	[U] En caso de error al enviar una imagen, se me permite reintentar o volver a empezar.	12
HU13	[U] Compartir el resultado de la búsqueda en Facebook y Twitter.	8
HU14	[U] Asignar una puntuación a los resultados obtenidos.	8
HU15	[U] Marcar como favorito un resultado obtenido.	4
HU16	[U] Acceder a mis favoritos.	2
HU17	[U] Apartado de *acerca de* donde aparezcan datos sobre la empresa.	2
HU18	[U] Apartado de licencias se especifiquen las licencias utilizadas para el desarrollo de la aplicación.	2
HU19	[U] Navegar por la aplicación mediante un *action bar*.	4
HU20	[A] Añadir imágenes con un contenido asociado al sistema.	8
HU21	[A] Añadir campañas al sistema a las que asignar imágenes.	2
HU22	[C] Almacenar las fotos realizadas, la imagen a la que corresponden y desde que móvil se han enviado.	2
HU23	[C] Integrar la aplicación con Google Analytics.	2
HU24	[E] Redacción del manual de usuario.	4
HU25	[E] Redacción de la Memoria.	40
HU26	[E] Preparación de la presentación.	6

Duración del *sprint*. Lo primero para planificar la duración de cada *sprint* es tener en cuenta los días festivos dentro del periodo de duración del proyecto. Este proyecto empieza el día 3 de marzo de 2014 y concluye el 30 de junio del mismo año. Entre estos días encontramos los siguientes festivos no laborables:

- 19 de Marzo: San José
- 24 de Marzo, Fiestas de la Magdalena.
- 28 de Marzo: Fiesta local.
- 18 de Abril: Viernes Santo.
- 21 de Abril: Lunes de Pascua.
- 1 de Mayo: Fiesta del Trabajo.

Teniendo en cuenta esto, se decide dividir el proyecto en iteraciones de 6 *sprints* durante la estancia en la empresa, más un séptimo, para la realización de la memoria y preparación de la presentación para el Trabajo Final de Grado. Estos se distribuyen en las siguientes fechas:

Tabla 4.4: Duración de los *sprints* del proyecto.

Sprint	**Fecha**
1	Del 3 de Marzo al 14 de Marzo (10 días).
2	Del 14 de Marzo al 17 de Abril (12 días).
3	Del 7 de Abril al 28 de Abril (13 días).
4	Del 28 de Abril al 19 de Mayo (14 días).
5	Del 19 de Mayo al 9 de Junio (15 días).
6	Del 9 de Junio al 30 de Junio (15 días).
7	Del 30 de Junio al 14 de Julio (15 días).

Velocidad del equipo. El término velocidad del equipo se refiere a la cantidad de puntos de historia que el equipo puede abordar en un *sprint*. Puesto que para esta empresa, un punto de historia son 2 horas y los *sprints* duran 15 días de 4 horas de jornada laboral, obtenemos 60 horas de trabajo cada *sprint*. Si dividimos estas 60 horas de trabajo entre 2 (que son las horas que se atribuyen a cada punto de historia), obtenemos una velocidad de 30 puntos de historia por cada *sprint*. No obstante, esta velocidad puede ser recalculada durante el proyecto si se ve que el equipo no dispone de capacidad suficiente para sacar adelante las historias estimadas para cada *sprint*.

4.3. Recursos

Para la realización de este proyecto se han utilizado recursos *hardware* y *software* proporcionados por la empresa *Rubycon-IT*. Entre ellos se encuentran:

- MacBook Pro 2.3 GHz (2410M) Intel Core i5 con 3 MiB on-chip L3 cache:
  - Sistema operativo OSX.
  - Xcode 5.
  - Simulador iOS.
- Acer Intel Core i7-3612QM 2.1GHz con Turbo Boost up a 3.1GHz:
  - Sistema operativo Windows 8.
  - Software para diseño de imágenes (para el diseño de iconos, botones, etcétera).
- Servidor Intel Xeon E3 1245v2 4 cores / 8 threads a 3.4 GHz, con 32 GB RAM:
  - Sistema operativo Gentoo.
- iPhone 4.
- iPhone 5.

4.4. Seguimiento del proyecto

4.4.1. Herramienta para seguimiento y control de la gestión temporal: *JIRA*

En *Rubycon-IT* están concienciados con la necesidad de hacer un seguimiento preciso y efectivo de la situación de las tareas en relación con el resto del proyecto. Para ello, disponen de una herramienta de gestión de proyectos realizados con *SCRUM* llamada *JIRA* [2], desarrollada por *Atlassian* [1].

La finalidad de este *software* es proporcionar a un equipo de desarrollo ágil todas las herramientas necesarias para gestionar errores, funciones, historias y tareas dentro de un proyecto concreto. En los siguientes subapartados se explican ejemplos de la funcionalidad que *Jira* ofrece aplicada a este proyecto.

Gestión de historias de usuario

Mediante JIRA, se asignan las tareas y historias de usuario a los *sprints* a los que corresponden. De esta forma, es posible visualizar en cada momento el estado del *sprint* actual y de las tareas restantes que aún no se han asignado a ningún *sprint* y que esperan en la pila del producto tal y como se muestra en la Figura 4.6.

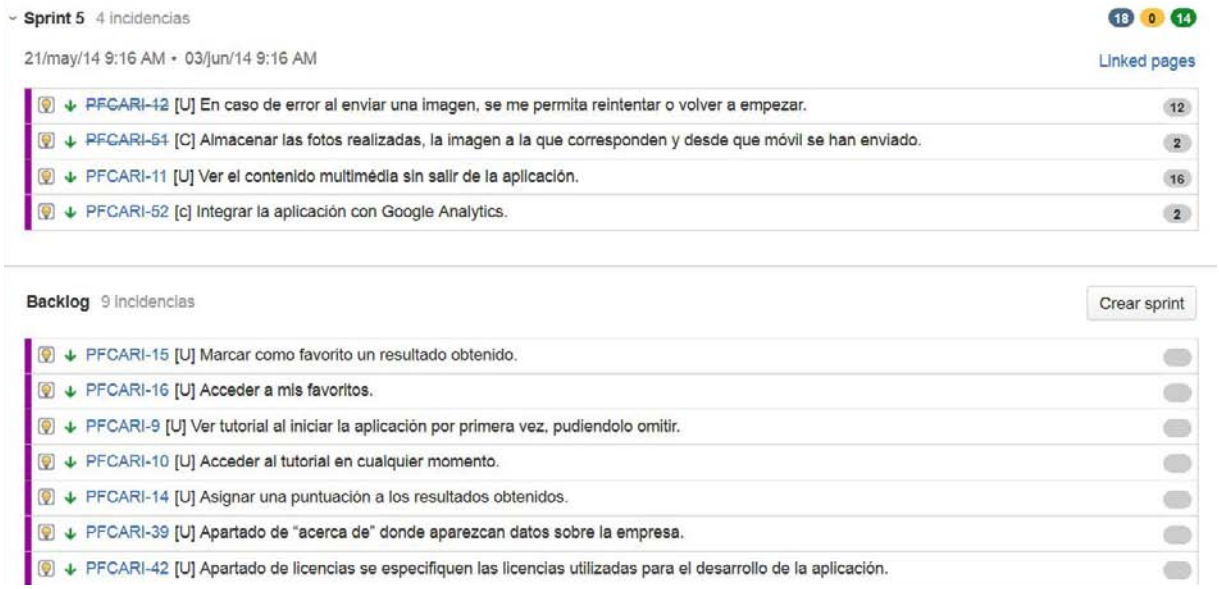


Figura 4.6: La Figura 4.6 muestra el panel de historias de usuario asignadas al *sprint* en proceso y *backlog* restante.

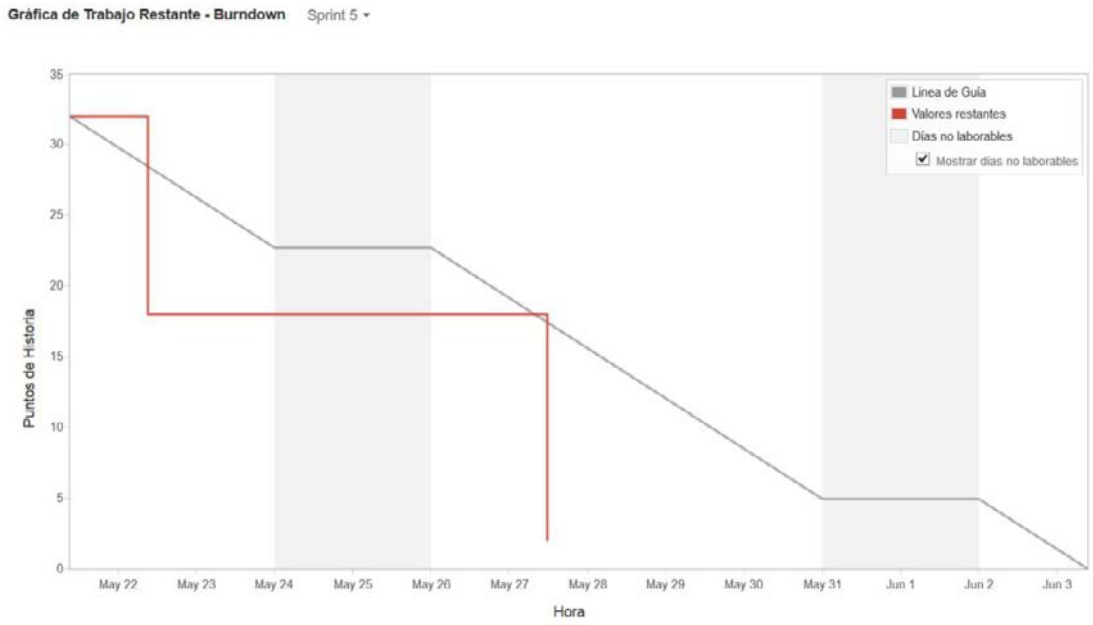


Figura 4.7: La Figura 4.7 muestra una gráfica temporal sobre el desarrollo y la finalización de las tareas.

Además, en un instante concreto, se pueden conocer el estado actual del *sprint* en el que se encuentra el proyecto. En la Figura 4.7 aparece un gráfico. La linea roja disminuye cuando se marcan como terminadas las historias de usuario o tareas asignadas a dicho *sprint*, mientras que la linea gris muestra el ritmo al que debería ir disminuyendo si el trabajo sigue un ritmo constante, teniendo en cuenta incluso los días no laborables.

Reportes con estadísticas de los *sprints*

Al finalizar cada *sprint*, *JIRA* ofrece reportes, informando sobre cuándo se han ido cerrando las historias de usuario y en qué estado se encuentran al finalizar el *sprint*. En el gráfico de la Figura 4.8 se muestra una historia de usuario que no ha sido cerrada al finalizar el *sprint* en el que nos encontramos (por eso aparece como *abierta*), si no que finaliza en el *sprint* siguiente.

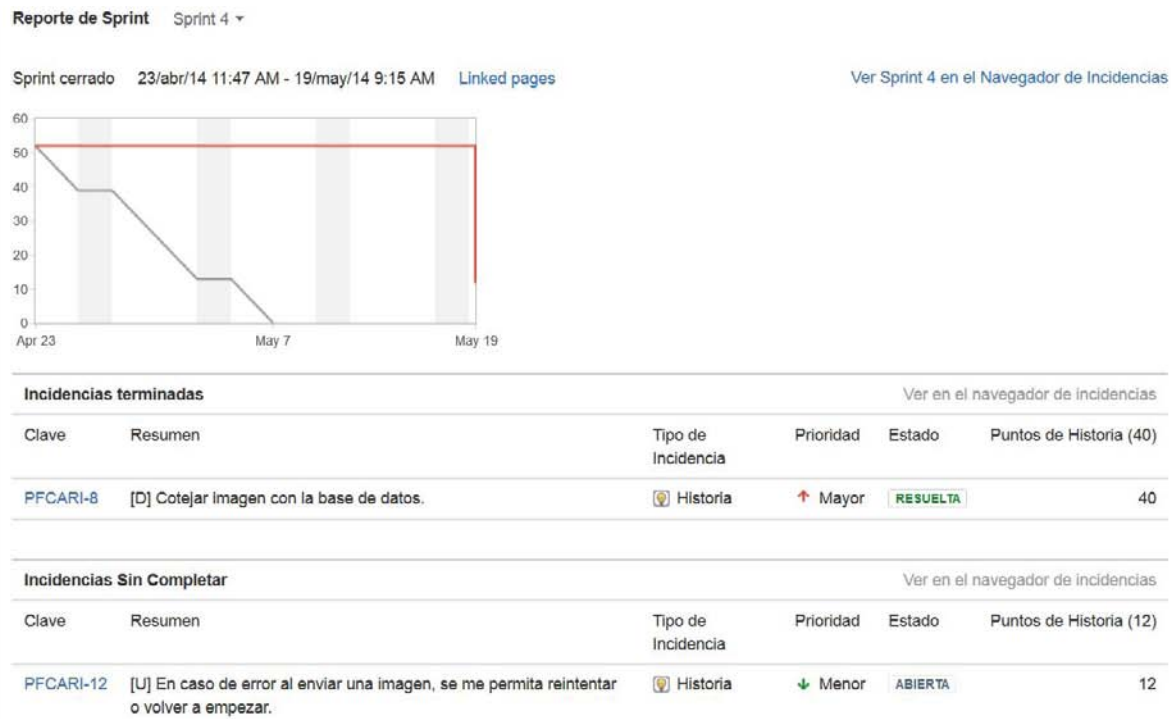


Figura 4.8: La Figura 4.8 muestra el reporte final con estadísticas sobre el resultado del *Sprint 4*.

Tablero *Kanban*

En la Figura 4.9 se muestra el tablero *Kanban* del estado de un *sprint* en el que se muestran, de las tareas asignadas a dicho *sprint*, cuáles de ellas están por hacer, cuáles en progreso y cuales terminadas. Además, para cambiar el estado de una tarea, es suficiente con arrastrarla de una columna a otra, de modo que la herramienta gestiona todo lo que el cambio de estado de la tarea implica, como por ejemplo, la terminación de un *sprint*.

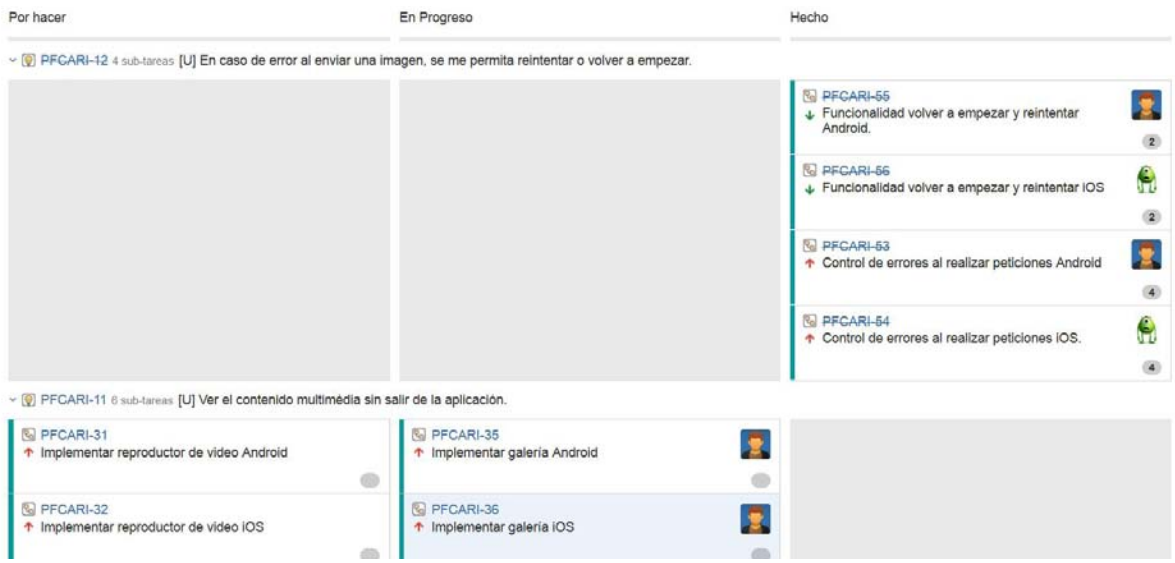


Figura 4.9: En la Figura 4.9 aparece un tablero tipo *Kanban*.

Flujos de actividad

Figura 4.10: La Figura 4.10 muestra un gráfico de creación y resolución de incidencias.

Las Figuras 4.10 y 4.11 muestran la forma en que *JIRA* nos muestra los periodos de tiempo en los que se ha producido una mayor actividad. En este caso se muestran un gráfico sobre la creación y resolución de incidencias (Figura 4.10) y una tabla sobre las incidencias más recientes

acontecidas (Figura 4.11).

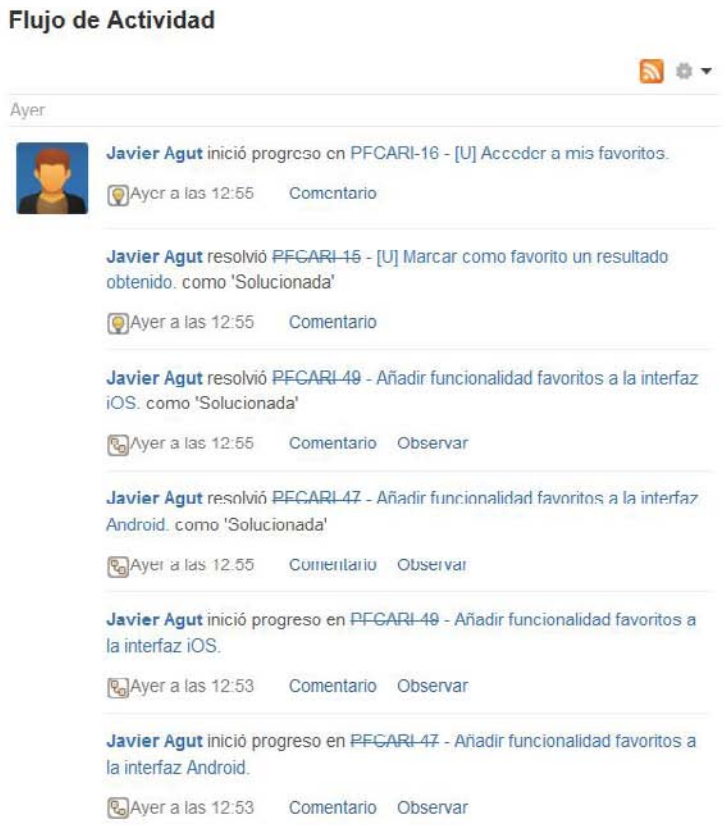


Figura 4.11: La Figura 4.11 muestra el flujo de actividad sobre la herramienta *Jira*.

4.4.2. Comparación entre lo estimado y lo acontecido

En un proyecto llevado a cabo siguiendo la metodología *SCRUM*, al final de cada *sprint* se hace una reunión con el cliente y se definen las historias de usuario que se realizarán en el siguiente. Es necesario apuntar que, además de las historias de usuario que aparecen en la tabla, en cada *sprint* se ha dedicado un tiempo (no despreciable) a rehacer o modificar cosas del *sprint* anterior para ajustar el resultado a los deseos del cliente.

El desglose del *backlog* entre los diferentes *sprints* ha terminado siendo el que sigue:

Tabla 4.5: Distribución del *backlog* entre los *sprints* del proyecto.

Sprint	Pila del sprint
1	[D] Familiarizarse con el entorno de trabajo. [D] Conceptos de programación móvil. [D] Investigación tecnologías servidor. [D] Investigación algortimos reconocimiento de imágenes.
2	[U] Realizar una fotografía con mi dispositivo móvil. [U] Enviar la imagen al servidor. [U] Recibir la imagen en el servidor.
3	[U] En caso de error al enviar una imagen, se me permite reintentar o volver a empezar (No finalizada). [D] Cotejar imagen con la base de datos (No finalizada). [U] Navegar por la aplicación mediante un *action bar*. [U] Compartir el resultado de la búsqueda en Facebook y Twitter.
4	[U] En caso de error al enviar una imagen, se me permite reintentar o volver a empezar. [D] Cotejar imagen con la base de datos.
5	[U] Ver el contenido multimedia sin salir de la aplicación. [U] Marcar como favorito un resultado obtenido. [U] Acceder a mis favoritos. [C] Almacenar las fotos realizadas, la imagen a la que corresponden y desde que móvil se han enviado.
6	[U] Apartado de *acerca de* donde aparezcan datos sobre la empresa. [U] Apartado de licencias se especifiquen las licencias utilizadas para el desarrollo de la aplicación. [U] Ver tutorial al iniciar la aplicación por primera vez, pudiendolo omitir. [U] Acceder al tutorial en cualquier momento.
7	[E] Redacción del manual de usuario. [E] Redacción de la Memoria. [E] Preparación de la presentación.

Una vez finalizado el proyecto, podemos hacer una comparación entre los puntos de historia que estimamos para cada historia de usuario, y los que realmente consumimos realizándola.

Tabla 4.6: Puntos de historia estimados y puntos de historia reales utilizados.

Id.	Historia	P. Estimados	P. Real
HU01	[D] Familiarizarse con el entorno de trabajo.	8	6
HU02	[D] Conceptos de programación móvil.	8	7+7
HU03	[D] Investigación tecnologías servidor.	4	2
HU04	[D] Investigación algoritmos reconocimiento de imágenes.	8	5+5
HU05	[U] Realizar una fotografía con mi dispositivo móvil.	10	12
HU06	[U] Enviar la imagen al servidor.	5	6
HU07	[U] Recibir la imagen en el servidor.	5	6
HU08	[D] Cotejar imagen con la base de datos.	40	14+26
HU09	[U] Ver tutorial al iniciar la aplicación por primera vez, pudiéndolo omitir.	5	6
HU10	[U] Acceder al tutorial en cualquier momento.	1	1
HU11	[U] Ver el contenido multimedia sin salir de la aplicación.	16	18
HU12	[U] En caso de error al enviar una imagen, se me permite reintentar o volver a empezar.	12	6+2
HU13	[U] Compartir el resultado de la búsqueda en Facebook y Twitter.	8	2
HU14	[U] Asignar una puntuación a los resultados obtenidos.	8	-
HU15	[U] Marcar como favorito un resultado obtenido.	4	5
HU16	[U] Acceder a mis favoritos.	2	2
HU17	[U] Apartado de *acerca de* donde aparezcan datos sobre la empresa.	2	2
HU18	[U] Apartado de licencias se especifiquen las licencias utilizadas para el desarrollo de la aplicación.	2	1
HU19	[U] Navegar por la aplicación mediante un ?action bar?.	4	4
HU20	[A] Añadir imágenes con un contenido asociado al sistema.	8	-
HU21	[A] Añadir campañas al sistema a las que asignar imágenes.	2	-
HU22	[C] Almacenar las fotos realizadas, la imagen a la que corresponden y desde que móvil se han enviado.	2	4
HU23	[C] Integrar la aplicación con Google Analytics.	2	-
HU24	[E] Redacción del manual de usuario.	4	
HU25	[E] Redacción de la Memoria.	40	
HU26	[E] Preparación de la presentación.	6	

Haciendo el cálculo, el total de puntos estimados (sin contar las historias de usuario que finalmente no se han realizado) es de 146 puntos de historia, mientras que los puntos de historia reales utilizados ascienden hasta los 149. Entre los dos hay una diferencia de 3 puntos de historia. Teniendo en cuenta que se han estimado tareas a realizar en lenguajes de programación y tecnologías que el equipo no había utilizado antes, los supervisores de la empresa han calificado una diferencia de 3 puntos de historia como una diferencia más que aceptable.

4.5. Estimación de costes

Una vez definidos los recursos necesarios, en cuanto a tiempo y herramientas necesarias, podemos estimar el coste del proyecto. El tiempo invertido por el estudiante en implementar el sistema es de media jornada durante 4 meses, a 20 días trabajados al mes durante 4 horas, hacen un total de: 320 horas. Cobrando un programador junio 15€/hora, el total es de: 4.800,00€-6.495,14$.

Capítulo 5

Análisis

En esta apartado se describe todo el estudio referente al análisis de la aplicación. Partiendo por los requisitos del sistema, se describen todas las partes que hacen que la aplicación funcione correctamente.

5.1. Requisitos del sistema

El desarrollo de la aplicación no ha estado motivado por la contratación de la empresa por parte de un cliente, si no que ha sido la propia empresa *Rubycon-IT* la que ha creído interesante la realización de esta aplicación, dadas sus muchas posibles aplicaciones al ámbito comercial. Es por esto que, para la toma de requisitos, los supervisores de la empresa han tomado el papel de clientes. Al principio del proyecto, se llevó a cabo una reunión en la que se establecieron los requisitos del sistema. Se definieron requisitos de tres tipo de requisitos:

- **Requisitos de datos**: expresan una necesidad referente a la información que debe almacenar el sistema.
- **Requisitos funcionales**: expresan una necesidad que se cubre añadiendo una funcionalidad al *software*.
- **Requisitos no funcionales**: expresan una necesidad o restricción que no tiene cabida en los anteriores apartados.

5.1.1. Requisitos de datos

El sistema tiene que devolver una información específica para cada imagen reconocida. No obstante, hay diferentes tipos de información devuelta, y el sistema debe contener la información para poder devolver correctamente resultados de los tipos:

- Web.

- Vídeo.

- Sonido MP3.

- Audioguía (MP3 con texto).

- Imágenes (modo galería: hay un conjunto de imágenes y una de ellas se muestra, pudiendo cambiar la imagen que se esta visualizando).

Para poder mostrar, en el dispositivo del usuario, todos estos tipos de información, es necesario tener una estructura que le llegue al dispositivo con los campos adecuados para poder obtener los diferentes tipos de resultado. A continuación se detalla la información que se necesita almacenar en el sistema para devolver cada tipo de resultado:

Tabla 5.1: Tipos de resultados.

Resultado	**Necesidades**
Web	Existen dos formas de mostrar una *web*: mediante contenido *HTML* o mediante un enlace *URL*. Por lo tanto, dentro de los resultados *Web* habrá que diferencia de que tipo son y, dependiendo de este, adjuntar el contenido *HTML* o la dirección *URL*.
Vídeo	Para poder descargar un vídeo es necesario conocer la dirección (*URL*) en la que éste se almacena.
Sonido MP3	Para poder descargar un MP3 es necesario conocer la dirección (*URL*) en la que éste se almacena.
Audioguía	Es el mismo formato que el *Sonido MP3*, pero, además, añadiendo texto complementario al sonido
Imágenes	Es necesario aportar las direcciones *URL* de cada imagen que se quiera mostrar al usuario.

Otro requisito funcional que el cliente demanda para su aplicación es la creación de un registro en el que se almacene, para cada petición que se realice al servidor, los datos referentes a dicho acceso. En la Tabla 5.2 se detallan los campos a almacenar.

Tabla 5.2: Campos que a guardar de cada acceso.

	Campos
Acceso	• Imagen enviada. • Si la imagen ha sido reconocida o no. • Imagen reconocida. • Fecha. • Hora. • Sistema operativo desde el que se realiza la petición. • Identificador del dispositivo.

Por último, el cliente nos pidió que los administradores del sistema pudieran crear *Campañas*. Dentro de cada campaña puede haber una o varias imágenes y, a cada imagen, se le podrá asignar un contenido de los tipos especificados en la Tabla 5.1.

Tabla 5.3: Campañas e imágenes.

	Campos
Campaña	Lista de imágenes
Imagen	Un único contenido asociado.

5.1.2. Requisitos funcionales

Los requisitos funcionales son los más numerosos. A continuación se define cada uno de ellos y, posteriormente, se incluye un diagrama de casos de uso. No obstante, no se incluyen los detalles de los casos de uso, ya que no ha sido necesario realizarlos, puesto que el proyecto se ha realizado mediante la metodología ágil *SCRUM*, en la que se genera otro tipo de documentación.

Capturar fotografías El cliente desea que aparezca una cámara dentro de la aplicación, que no sea necesario abandonarla para capturar una fotografía. Además, debe ser posible capturar una imagen en el momento y poder seleccionar una desde la galería del dispositivo.

Reconocer imágenes Es necesario que cuando se realice una fotografía en la que aparezca una imagen de las incluidas en el sistema, este debe reconocerla.

Reintentar y reenviar En el caso de que se produzca un fallo a la hora de reconocer una imagen, en la aplicación se debe proporcionar al usuario de una forma sencilla la posibilidad de reenviar la imagen a reconocer o de volver a la cámara para tomar otra fotografía.

Reproductores multimedia Una petición muy clara del cliente es que todos los tipos de resultado deben poder visualizarse sin salir de la aplicación. Es decir, no reproducir vídeo y audio con las aplicaciones por defecto de *iPhone*, si no mediante un reproductor implementado dentro de la misma aplicación desarrollada.

Compartir resultado El usuario debe poder compartir en las principales redes sociales el resultado devuelto como contenido asociado a la imagen reconocida, pudiendo, además, añadir un comentario personalizado.

Gestión de resultado El usuario debe poder asignar un resultado como favorito para, más tarde, poder acceder a él rápidamente, así como puntuar cada resultado devuelto.

Tutorial Se debe proporcionar al usuario un tutorial que le ayude a saber cómo utilizar la aplicación, además de la posibilidad, tanto de volver a verlo en cualquier momento, como de omitirlo.

Apartado *Acerca de* Debe existir en la aplicación un apartado *Acerca de* donde aparezcan datos sobre la empresa, datos sobre las licencias utilizadas en el desarrollo de la aplicación y un acceso al tutorial.

Action bar La forma de navegar entre diferentes ventanas de la aplicación y de acceder a funcionalidad diversa se realizará mediante un *action bar* (barra superior de menú que contiene accesos a diversa funcionalidad), desde donde el usuario encuentre un acceso rápido, fácil e intuitivo a la funcionalidad más habitual de la aplicación

Backend Los administradores del sistema deberán poder crear *Campañas* y, dentro de estas, añadir una o diversas *Imágenes*, cada una con su contenido asociado.

De los anteriores requisitos se deriva un conjunto de Casos de Uso, de los que se adjunta, en la Figura 5.1 su diagrama.

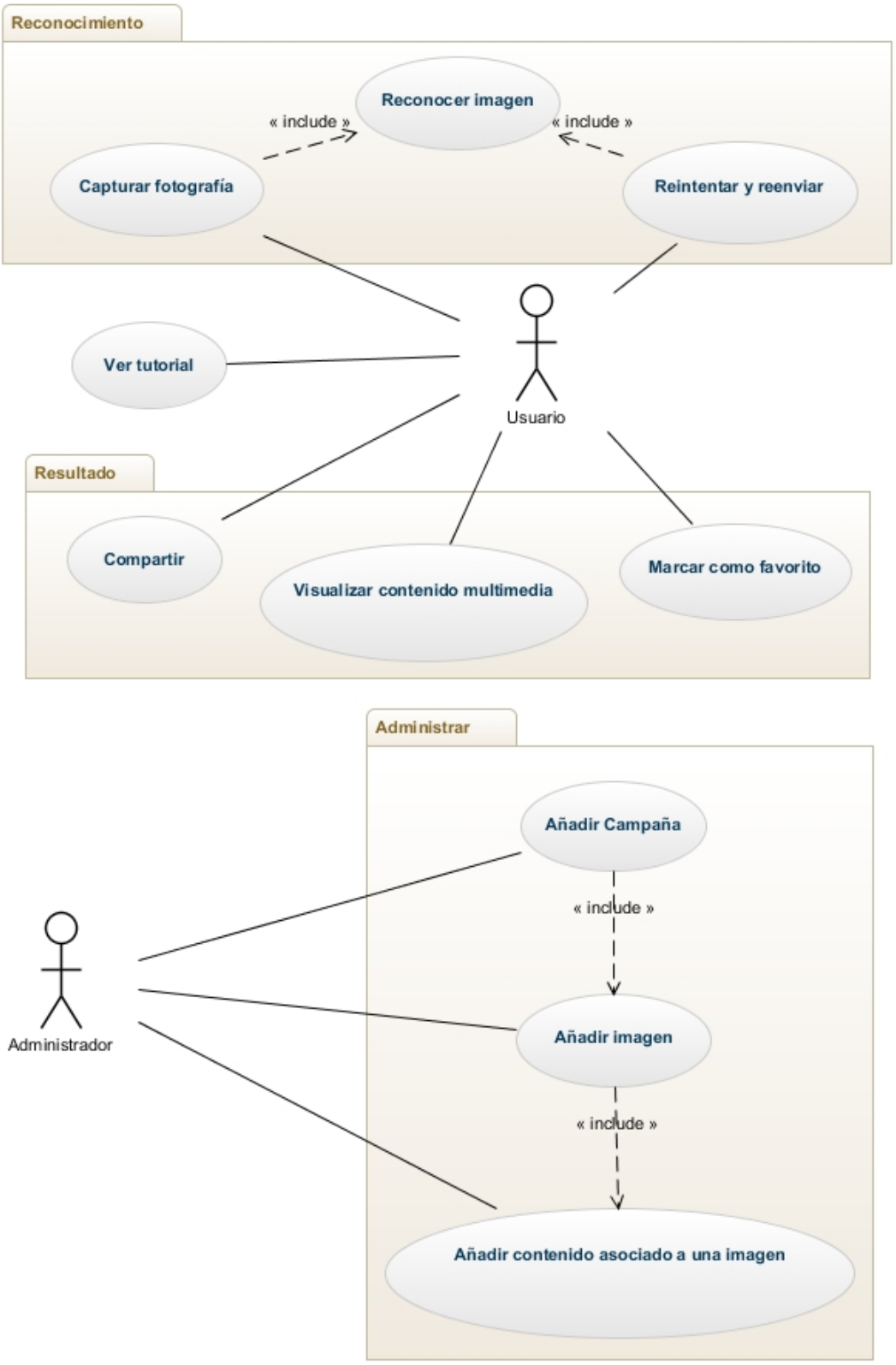


Figura 5.1: La Figura 5.1 es el diagrama de casos de uso de la aplicación.

5.1.3. Requisitos no funcionales

Los requisitos no funcionales son, muchas veces, los más difíciles de reflejar, puesto que pueden ser complicados de expresar y de documentar. No obstante, en nuestra aplicación aparecen algunos.

Tiempo de reconocimiento El tiempo que el sistema emplee en reconocer la imagen debe ser un tiempo razonable. Este tiempo va a depender de la cantidad de imágenes que contenga la base de datos, con las que se va a tener que comparar la fotografía capturada en el dispositivo. No obstante, con un número de imágenes razonable (entre 200 y 500) el sistema debe tardar en reconocer una imagen menos de 20 segundos. En este tiempo no está incluido el tiempo que tarda el dispositivo en enviar la imagen al servidor, ya que este tiempo depende en gran medida de la conexión de la que se disponga en el momento del envío.

Resultado no reconocido El sistema no puede ser exacto, teniendo en cuenta las condiciones variables de luz, posición, resolución, entre otros, que este proyecto asume. No obstante, si se envía una imagen a reconocer que no pertenece a las contenidas en la base de datos, el resultado debe ser *"La imagen no se ha reconocido"*.

Google Analytics La aplicación debe estar integrada con *Google Analytics*.

Capítulo 6

Diseño

En este apartado se revisa el diseño de la aplicación. Primero se muestran los diseños sobre el interfaz gráfico de usuario y, en el siguiente apartado, se detallan los detalles en cuanto al diseño del comportamiento de la aplicación y su funcionalidad. Por último, se aborda el diseño de *software* realizado previo a la implementación de esta aplicación

6.1. Interfaz de usuario

En los últimos meses, el diseño de las interfaces de usuario para dispositivos móviles ha dado una vuelta más de tuerca. Y este cambio ha sido producido, principalmente, por *Apple*, liderando una migración de las interfaces más metafóricas (basadas en representaciones de objetos de la vida real, con el fin de que el usuario sepa cómo utilizar cada componente), intentando representar lo más fielmente posible la realidad, hacia interfaces más planas, icónicas, con pocos colores, etcétera.

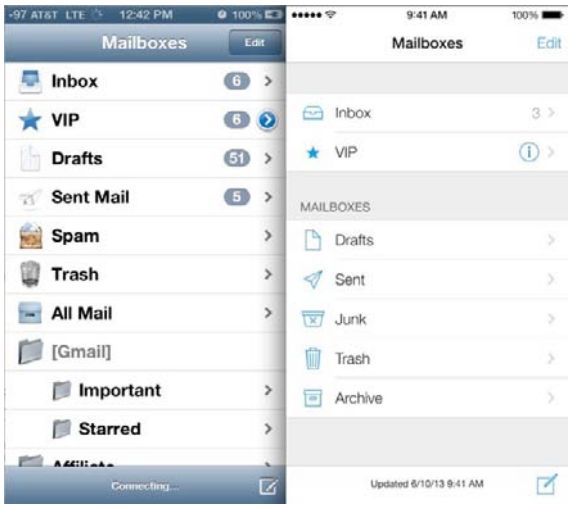


Figura 6.1: La Figara 6.1 muestra una comparación de las interfaces de *iOS 6* y *iOS 7*.

Por poner un ejemplo, la Figura 6.1 muestra la evolución de las interfaces gráficas de *iOS 6* a *iOS 7*. Y a esta tendencia se han adherido las aplicaciones más importantes del mercado, tanto de *iOS* como de *Android*. Por lo tanto, es interesante intentar acercarse lo más posible a estos modelos.

Para ello, *Apple* ha creado documentos cuyo fin es ayudar a los desarrolladores a mejorar el aspecto y la usabilidad de sus interfaces de usuario. Estos documentos se encuentran disponibles en la red (*Designing for iOS 7* [6]), y se ha intentado seguirlos para obtener una interfaz de usuario satisfactoria. A continuación se describen las principales pantallas de la aplicación.

Una de las primeras cuestiones que se sugieren en esta guía es la elección de los colores. *Apple* aboga por crear interfaces que no contengan un número demasiado elevado de colores, así que para esta aplicación se tomó la misma decisión. Toda la aplicación está diseñada con un color principal azul, un color secundario blanco, y ayudándose de los grises cuando es necesario. En la Figura 6.2 se muestran dichos colores.

Figura 6.2: La Figura 6.2 contiene la paleta de colores utilizada en la aplicación.

6.1.1. Pantalla de inicio

La pantalla de inicio debe de ser una pantalla introductoria, que de la bienvenida al usuario y que no ofrezca demasiadas opciones. En un primer momento, se diseñó un *mockup* demasiado sencillo, al que posteriormente fue necesario añadir una barra de pestañas, para que el usuario pudiera desplazarse entre la vista de *inicio*, la de *favoritos* y la de *acerca de*.

Figura 6.3: *Mockup* inicial de la *view* de inicio.

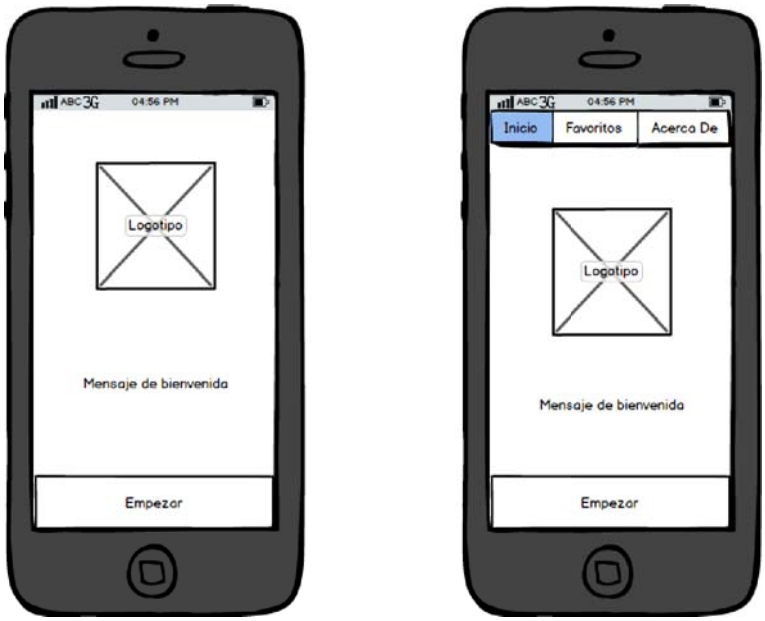


Figura 6.4: *Mockups* de la *view* de inicio.

El botón de *Empezar* que aparece abajo muestra la vista de la cámara cuando es presionado. Los botones de la barra de pestañas o *tab bar* actúan como botones, es decir, la vista cambia a la seleccionada cuando uno de estos botones es clicado. No obstante, también se ha implementado la funcionalidad de deslizar la pantalla a izquierda o derecha para cambiar entre las pestañas.

6.1.2. Pantalla de cámara

La pantalla de cámara no tiene demasiada complejidad, en lo que al diseño se refiere. Simplemente cuenta con un botón para capturar una fotografía, otro para acceder a la galería y otro para cerrar la cámara, aunque es cierto que este último fue añadido cuando el proyecto ya estaba bastante avanzado. Mencionar que los botones se han diseñado manualmente para adaptarlos a los colores de la aplicación previamente escogidos y para dotarlos de cierta estética. En la Figura 6.5 se incluyen los *mockups* de esta view.

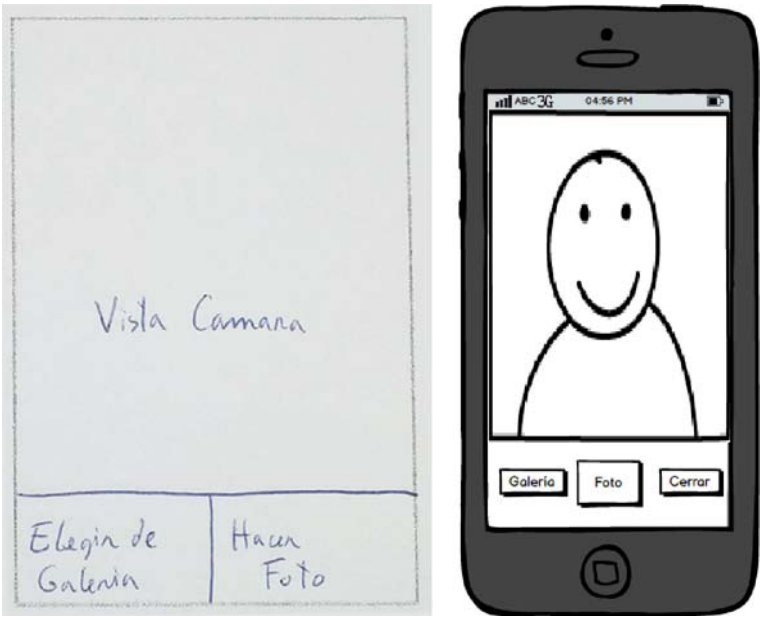


Figura 6.5: *Mockups* de la *view* de cámara.

6.1.3. Resultado Web

En esta vista se mostrarán los resultados que devuelvan una página web. Aparte de contener la barra superior, aparece en esta vista una *Web View* en la que se muestra el contenido de la web que corresponda. El usuario, mediante esta *Web View*, puede interactuar con la página web como si de un navegador se tratara. Sólo que, por motivos de seguridad de *Apple*, cuando se clica en un enlace, este tiene que abrirse en el navegador predeterminado del *iPhone*.

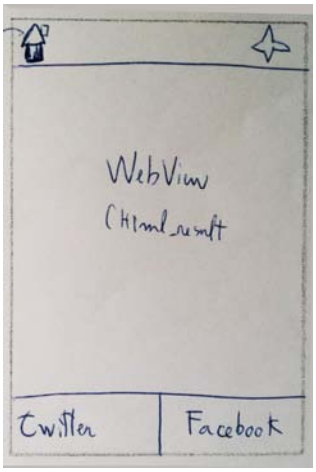


Figura 6.6: *Mockup* inicial de la *view* de resultado *web*.

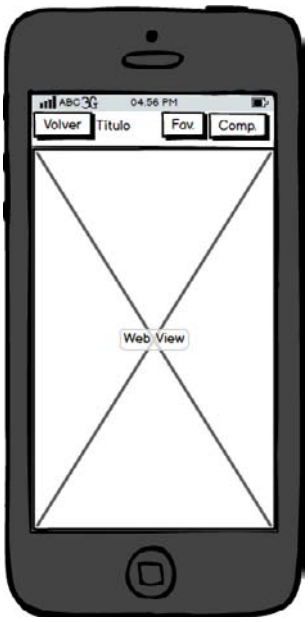


Figura 6.7: *Mockup* de la *view* de resultado *web*.

Como se puede observar, puesto que esta vista se diseñó en primer lugar, la diferencia más significativa entre el *mockup* inicial (Figura 6.6) y su evolución (figura 6.7) aparecen diferencias: los botones de compartir aparecen abajo en la pantalla y no arriba.

6.1.4. Resultado Galería

Esta vista sufrió algunas modificaciones desde los diseños iniciales hasta la vista final. En un primer instante, se decide mostrar la galería como un conjunto de miniaturas de imágenes:

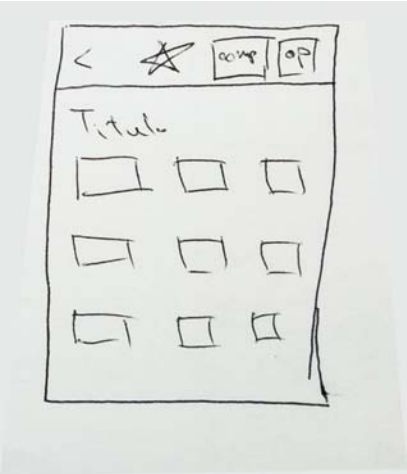


Figura 6.8: *Mockup* inicial de la *view* de resultado Galería.

No obstante, en la siguiente reunión, se redirige el diseño de esta vista en otra dirección. Se decide que la vista conste, aparte de la barra superior, de un *Image View* en el que se sitúan las imágenes, que irán cambiando. Aparte, contiene una label para que el usuario sepa qué imagen de la galería está visualizando y cuántas hay. Dicho esto, el *mockup* tampoco debe resultar demasiado complicado, tal y como se muestra en la Figura 6.9.

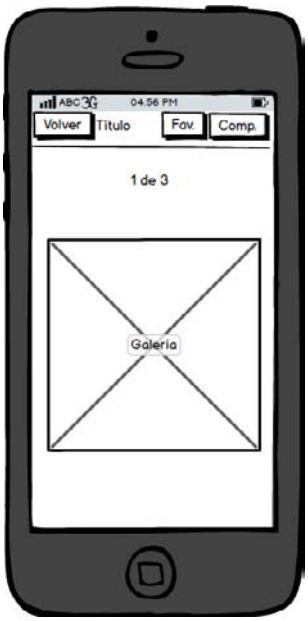


Figura 6.9: *Mockup* de la *view* de resultado Galería.

6.1.5. Resultado Audio

Para esta *view*, si ha sido necesario realizar algunas modificaciones en los *mockups*. En un primer instante, se pensó en no manejar el tiempo de la pista de audio. Para la siguiente reunión, se cambió de idea y se decidió que el tiempo se controlaría mediante los botones de *play*, *pause*, *backward* y *forward*. No obstante, antes de ponerse a implementar, se decidió que era más útil para los usuarios contar con un botón que cambiara de aspecto para la funcionalidad de *play/pause* y de un *slider* que se fuera moviendo tal y como el tiempo fuera pasando. Con esto, el usuario podría modificar el momento de la pista que quiere escuchar simplemente desplazando el *slider*. Por último, se pensó en añadir una *label* con el segundo concreto de la pista que se está reproduciendo. A continuación se muestran los *mockups* con sus modificaciones (Figuras 6.10 y 6.11).

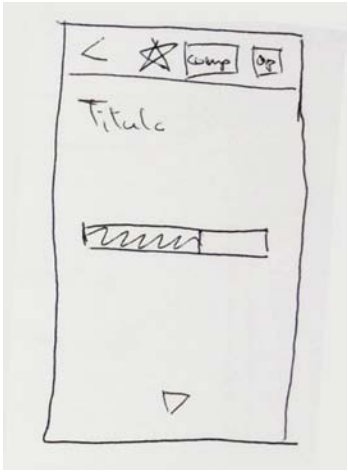


Figura 6.10: *Mockup* inicial de la *view* de resultado Audio.

Figura 6.11: *Mockups* de la *view* de resultado Audio.

6.1.6. Resultado Vídeo

En cuanto a la vista de resultado de vídeo, ocurrió lo mismo que con la de audio, en lo que a los botones para controlar el tiempo de reproducción se refiere. No obstante, el problema aquí es más grave, ya que esta vista debe mostrar el vídeo en pantalla completa al girar la pantalla. Estos son los *mockups* de esta vista:

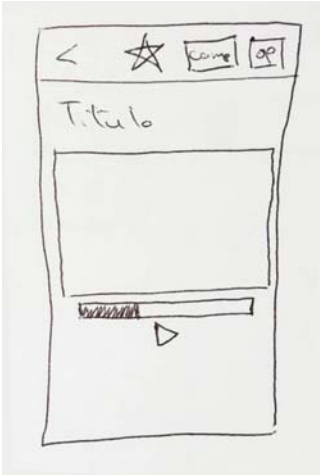


Figura 6.12: *Mockup* inicial de la *view* de resultado Vídeo.

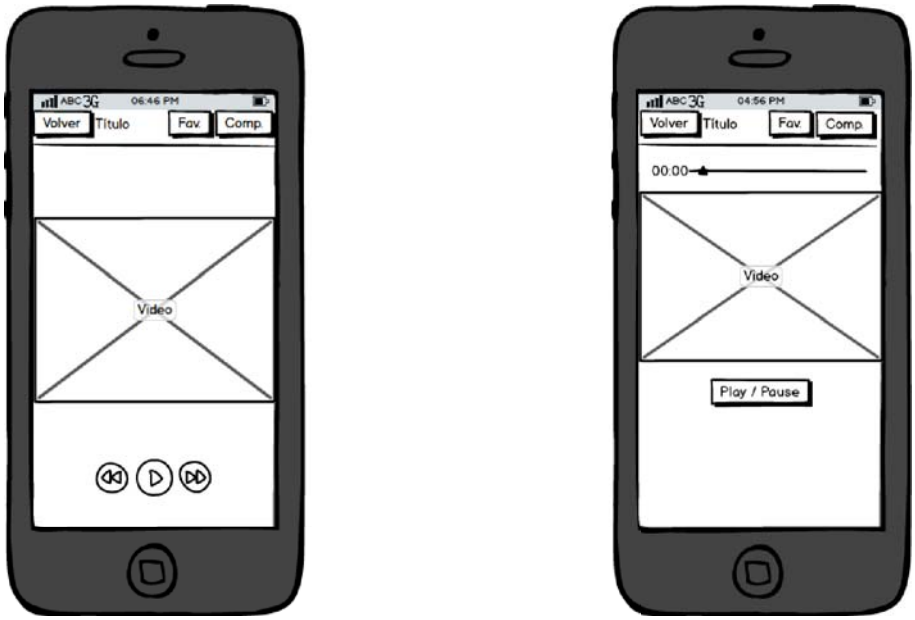


Figura 6.13: *Mockups* de la *view* de resultado Vídeo en vertical.

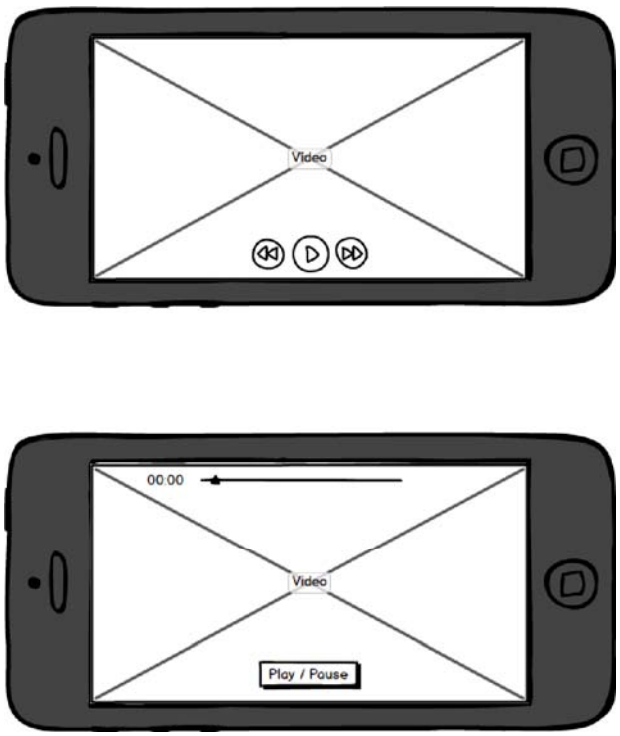


Figura 6.14: *Mockups* de la *view* de resultado Vídeo en horizontal.

6.1.7. Vista de Favoritos

Para la vista de favoritos, la idea que se tenía en mente era recrear una lista, tipo contactos del *whatsApp*, que se pudiera desplazar, en la que poder ver los favoritos que previamente se han marcado como tal. Además, al clicar en cada favorito, se abre una vista resultado mostrando el favorito seleccionado.

Figura 6.15: *Mockup* inicial de la *view* de Favoritos.

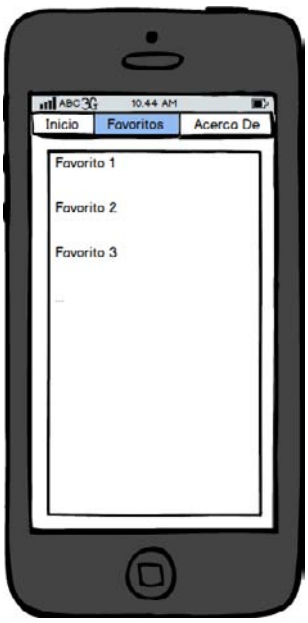


Figura 6.16: *Mockup* de la *view* de Favoritos.

6.2. Diseño de procesos

Esta aplicación no contiene procesos que tengan tantos caminos posibles como para hacerlos susceptibles de una representación en un diagrama de actividad, excepto por lo que a la funcionalidad de reconocer imagen se refiere. Desde que iniciamos la aplicación hasta que obtenemos un resultado del reconocimiento sí pueden acontecer diferentes eventos, hasta el nivel de que su explicación se simplifica mediante la representación de dicho proceso en un diagrama UML:

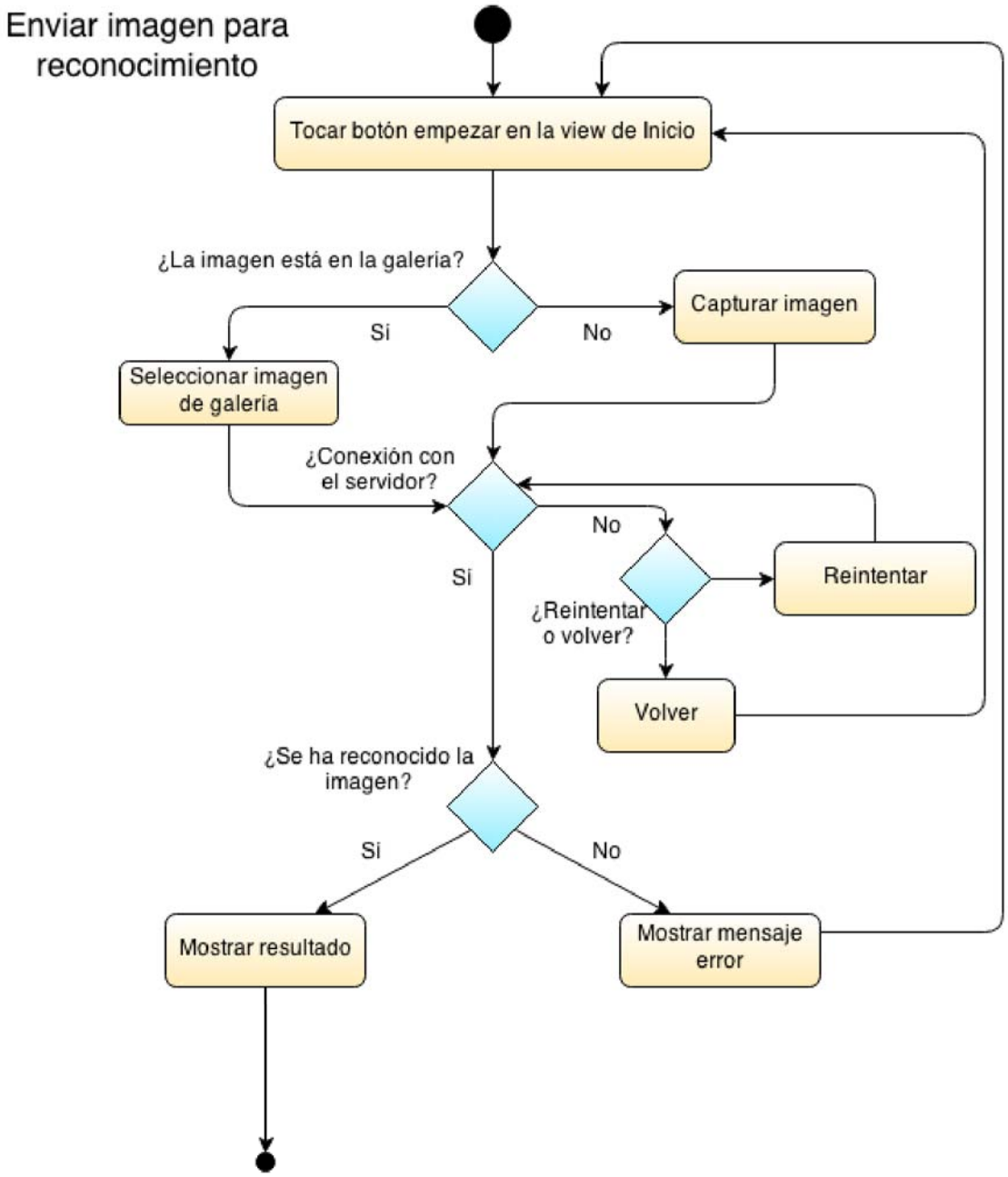


Figura 6.17: En la Figura 6.17 se muestra el diagrama UML de actividad del proceso *reconocer imagen* del dispositivo.

6.3. Diagrama de clases

Llegados a este punto, es necesario diferenciar entre las clases de la aplicación móvil y las clases del servidor. El objetivo de la aplicación móvil es simplemente mostrar la información recibida, por lo que no es necesario guardar información (excepto los favoritos, que se guardan en la base de datos interna), de modo que las clases del cliente móvil no guardan demasiada relación unas con otras, si no que simplemente son controladores y vistas. No obstante, en el servidor sí que existe una cantidad de clases amplia y relacionadas entre sí.

Para acceder a la base de datos, se utiliza la clase **BaseDeDatos**. Desde ella se gestionas tanto las inserciones como las consultas a la base de datos. Ésta contiene imágenes y accesos. Para añadir un acceso, se utiliza una instancia de la clase **Acceso**. Por otra parte, para insertar imágenes en la base de datos, puesto que existen diferentes tipos de imágenes, se ha diseñado una clase abstracta **Imagen**, la cual extienden las clases concretas **ImagenVideo**, **ImagenInfo**, **ImagenAudio**, **ImagenWeb**, **ImagenGaleria** e **ImagenUrl**.

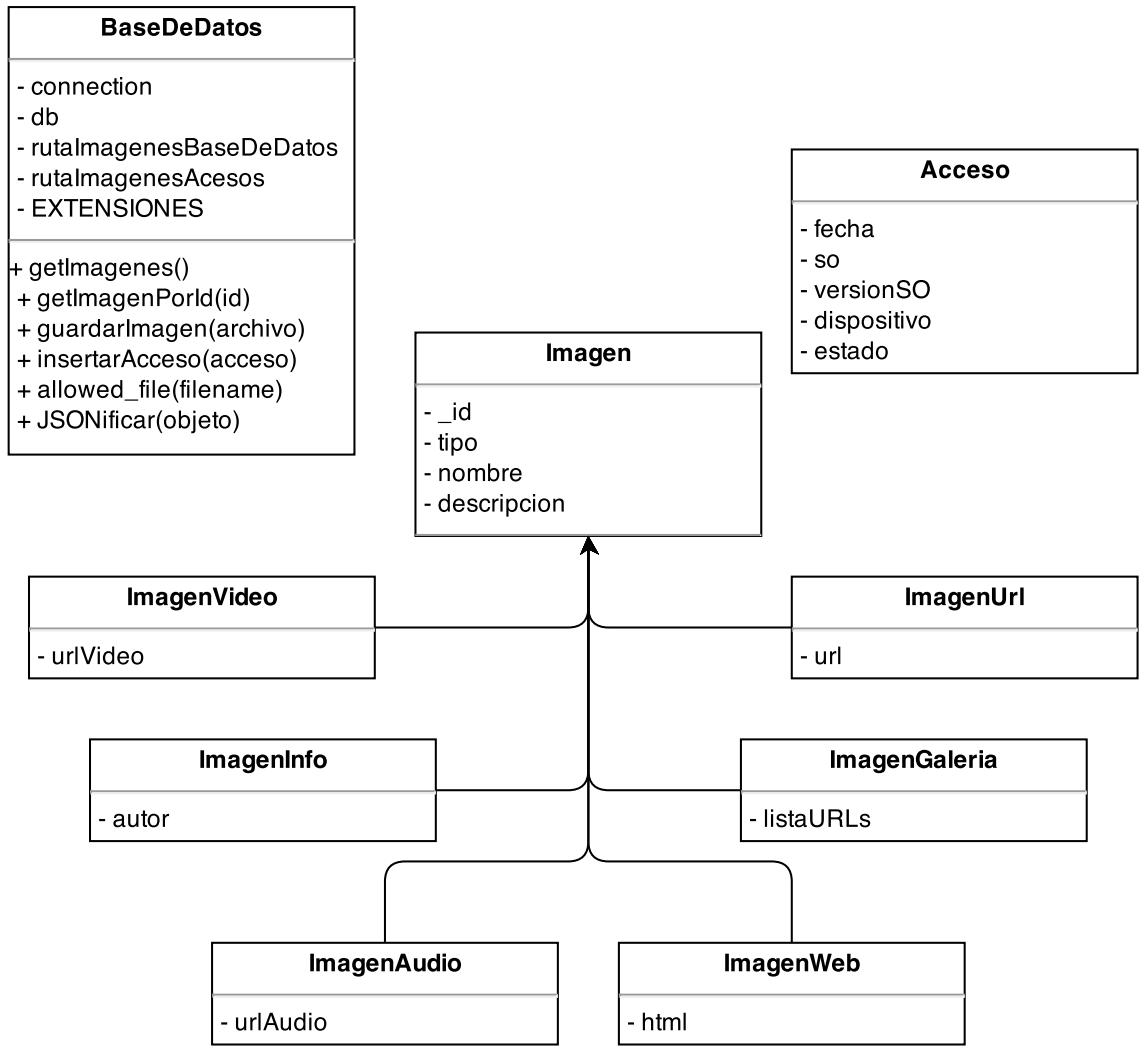


Figura 6.18: La Figura 6.18 es el diagrama de clases sobre las clases de la base de datos.

En cuanto al proceso de reconocimiento, aparte de las clases **Reconocedor**, **Detector** y **Comparador**, existe la clase **ImagenAnalizada**, que se utiliza para gestionar las diferentes imágenes que intervienen en el proceso de reconocimiento. También se ha creado la clase **Estado**, que se utiliza para devolver la información resultante del reconocimiento al cliente. En cuanto a la clase **Reconocedor**, sirve para conectar la base de datos con el proceso de reconocer imágenes, es decir, con el detector de descriptores y el comparador.

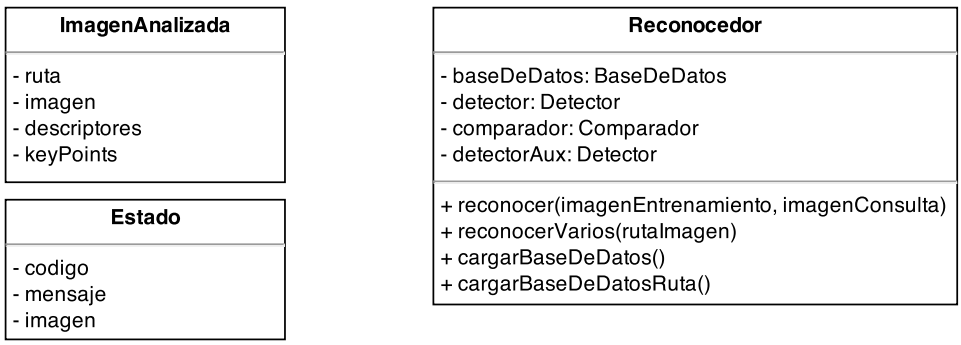


Figura 6.19: La Figura 6.19 contiene las clases **ImagenAnalizada**, **Estado** y **Reconocedor**.

Como existen diversos detectores y comparadores, pero todos se utilizan para lo mismo, se han creado dos clases abstractas: **Detector** y **Comparador**, que los detectores y comparadores concretos extienden, respectivamente.

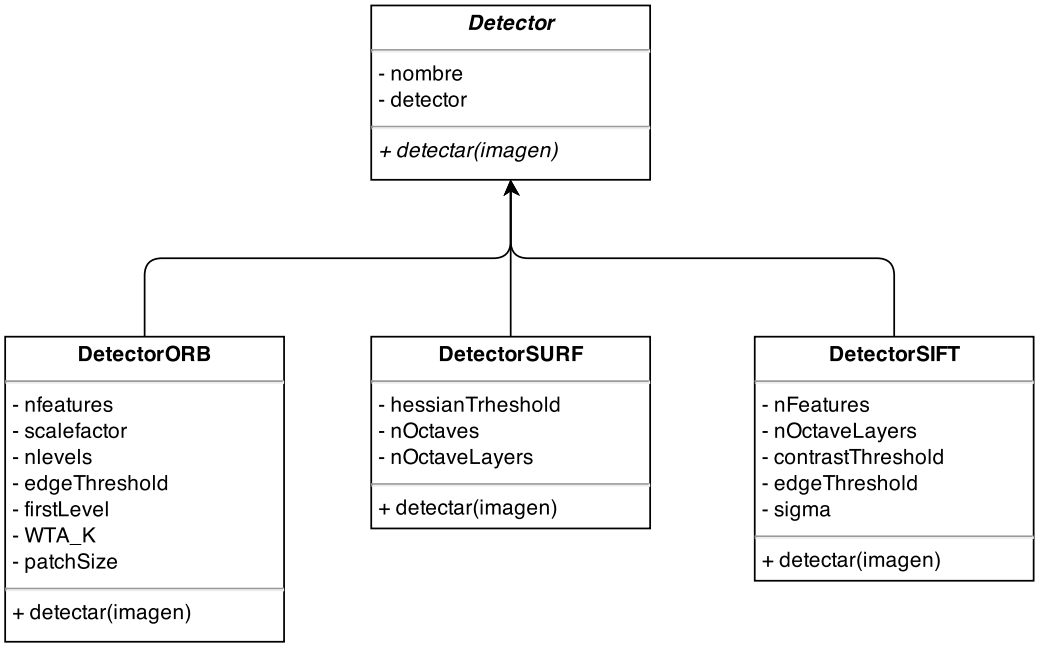


Figura 6.20: La Figura 6.20 muestra el diagrama de clases de la clase **Detector**.

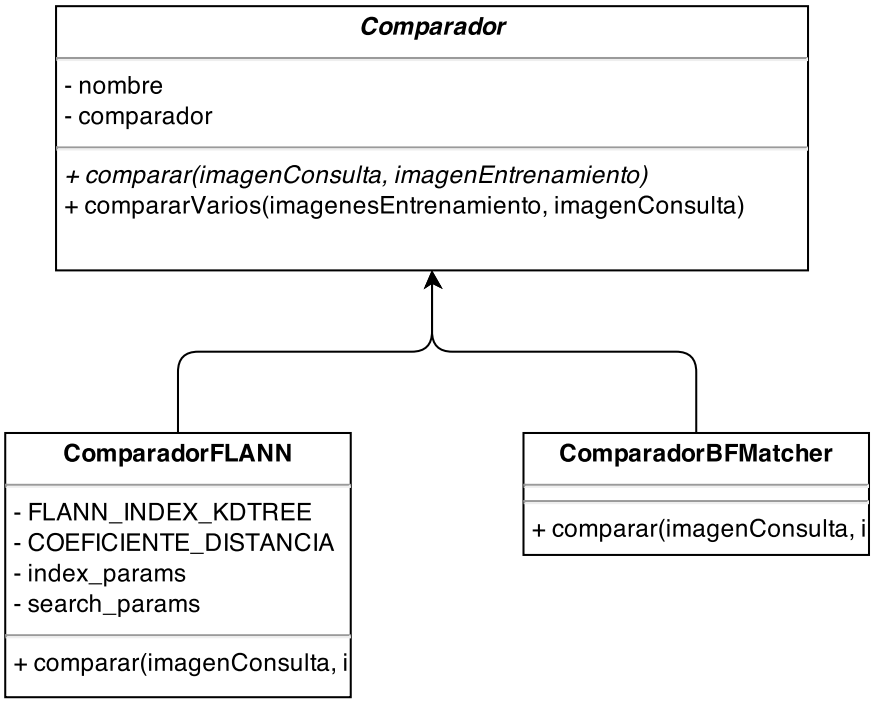


Figura 6.21: La Figura 6.21 muestra el diagrama de clases de la clase **Comparador**.

Capítulo 7

Implementación

Durante este capítulo se detallan los pormenores, en lo que a la implementación se refiere, sobre las cuestiones más extraordinarias del proyecto. Para hacerlo, se divide el código en partes. La primera división representa la división entre cliente y servidor. Aunque se comunican, se han desarrollado en lenguajes de programación diferentes y su implementación no guarda relación.

En el siguiente apartado se realiza un recorrido por los los diferentes objetivos que se ha buscado conseguir en el cliente para determinar un orden coherente a la hora de recorrer los ficheros. En cuanto a la parte del servidor, se abordará desde los ficheros que tienen menos que ver con el reconocimiento para terminar con la implementación del reconocimiento en sí, que es lo más complejo del proyecto.

7.1. Implementación del cliente *iOS*

Prácticamente la totalidad del cliente son controladores y vistas, emparejados de dos en dos. Cada controlador se encarga de gestionar una vista y de, cuando el usuario lo decida, ordenar al controlador correspondiente que se ejecute. No se ha implementado *modelo* para gestionar la información que se muestra, por ser relativamente sencillo. No obstante, sí existe un *modelo* para gestionar la base de datos interna, tal y como se explica en el subapartado 7.1.6.

El cliente de la aplicación tiene tres partes diferenciadas:

- Cámara.
- Mostrar información.
- Organización de las vistas y otros.

Comenzaremos por la parte de capturar una imagen para enviarla al servidor, continuando por las implementaciones de las vistas de resultado y favoritos, para terminar explicando cómo se han organizado la jerarquía de vistas.

7.1.1. Cámara

El controlador que se encarga de gestionar la vista de cámara ha resultado el más complicado de implementar. Esto es porque, tanto para capturar una imagen, como para seleccionar una imagen de galería, hay que utilizar una instancia de la clase *UIImagePickerController*. Puesto que lo que se busca es tener la cámara encendida (es decir, que se visualice en la pantalla lo que la cámara del *iPhone* capta) y desde ese estado, ofrecer al usuario la posibilidad de capturar lo que se está mostrando por pantalla como fotografía o acceder a la galería para seleccionar otra, es necesario crear dos controladores de la clase *UIImagePickerController*, uno para capturar y otro para ir a la galería. Por lo tanto, una vez la imagen ha sido seleccionada, es necesario tener en cuenta cuantos se han iniciado para poder continuar hacia la siguiente vista. Hasta que este problema se entiende, la implementación de la cámara es muy difícil de comprender, ya que no se sabe el porqué de ciertas acciones que, una vez comprendido como funciona el *UIImagePickerController*, se entiende su necesidad.

Para conocer en cada momento si se ha seleccionado la opción de acceder a galería, se han creado valores booleanos que nos lo indican. De esta forma, cuando se termina de capturar la imagen (ya sea mediante la cámara o mediante la galería), se comprueba desde dónde se ha capturado y se actúa en consecuencia.

Bloque de código 7.1: Captura de cámara o desde galería.

```
if (_newMedia){
          UIImageWriteToSavedPhotosAlbum(image,
                                         self,
                                         @selector(image:finishedSavingWithError:contextInfo:),
                                         nil);

          EnviarImagenViewController *enviarImagenViewController =
              [[EnviarImagenViewController alloc] initWithNibName:@"EnviarImagenView"
              bundle:nil];
          enviarImagenViewController.imagen = info[UIImagePickerControllerOriginalImage];
          [self presentViewController:enviarImagenViewController animated:YES completion:nil];
      }else{
          //Cuando venimos de la galería ponemos el imagePicker a null y
              volverAIniciarImagePicker a No
          imagePickerController = nil;
          volverAIniciarImagePicker = NO;

          //Esta view que estamos despresentando corresponde a la view de la galería
          [self dismissViewControllerAnimated:YES completion:^{
              EnviarImagenViewController *enviarImagenViewController =
                  [[EnviarImagenViewController alloc] initWithNibName:@"EnviarImagenView"
                  bundle:nil];
              enviarImagenViewController.imagen = info[UIImagePickerControllerOriginalImage];

              [self presentViewController:enviarImagenViewController animated:YES
                  completion:nil];
          }];
      }
```

Además de esto, para poder implementar la funcionalidad que requiere esta vista, es necesario nombrar a tu clase como *UIImagePickerControllerDelegate*, lo que implica que tiene que implementar los métodos:

- *-(void)imagePickerController:(UIImagePickerController *)picker didFinishPickingMediaWithInfo:(NSDictionary *)info*: este método es llamado cuando se selecciona una foto, ya sea mediante la cámara o mediante la galería, en nuestro caso, incluye el código que se muestra en el listing **??**.

- *-(void)image:(UIImage *)image finishedSavingWithError:(NSError *)error contextInfo:(void *)contextInfo*: este método se ejecuta cuando ocurre algún error a la hora de almacenar la nueva imagen capturada en la galería del dispositivo.

- *-(void)imagePickerControllerDidCancel:(UIImagePickerController *)picker*: contiene el código que se ejecuta cuando se cancela el controlador para capturar imágenes. Por ejemplo, es llamado cuando se pulsa el botón de cancelar desde la galería. En este caso, hay que indicar al controlador que debe volver a ejecutar el *UIImagePickerController* para que se siga mostrando en la pantalla lo que capta la cámara del dispositivo.

7.1.2. Resultado Audio

La dificultad de esta vista se la confiere el hecho de tener que manejar, mediante un *slider*, el tiempo de la pista de audio que se está reproduciendo. Para ello, se hace uso de una instancia del objeto *NSTimer*, que sirve para medir el tiempo. Ademas, se debe declarar el controlador como delegado de *AVAudioPlayer*, que es el reproductor de audio que utiliza.

Desde esta vista, se realiza una petición asíncrona para descargar el archivo de audio a reproducir. Cuando esto ocurre, se inicia el reproductor de audio, tal y como se muestra en el Bloque de código 7.2.

Bloque de código 7.2: Descarga asíncrona de un archivo de audio.

```
[operation setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
    responseObject) {
    NSLog(@"Successfully downloaded file to %@", path);
    urlAudio = path;

    [self abrirPlayer];
    [self liberarLoading];



} failure:^(AFHTTPRequestOperation *operation, NSError *error) {
    NSLog(@"Error: %@", error);
    [self liberarLoading];
    [self mostrarViewError];
}];

[operation start];
```

En el Bloque de código 7.2 podemos ver un ejemplo de los *blocks* de *Objective-C*. Dentro del objeto *operation* de declara un método que se ejecuta cuando el archivo se descarga correctamente (que para la animación de *loading* y ejecuta el reproductor de audio) y otro que se ejecuta cuando ocurre un fallo en la descarga (que, aparte de detener la animación de çargando", muestra una vista de error).

De la misma forma que ocurre en el controlador de la cámara, al declarar este controlador como *AVAudioPlayerDelegate*, debe implementar los siguientes métodos:

- - *(void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag*: este método es llamado cuando termina de reproducirse la pista de audio. En este caso, solamente es necesario detener el reloj para que no siga contando segundos si la pista de audio ya ha terminado.
- - *(void)audioPlayerDecodeErrorDidOccur:(AVAudioPlayer *)player error:(NSError *)error*: este método se ejecuta cuando ocurre algún error a la hora de decodificar el fichero de audio descargado.

Aparte de los métodos para controlar el audio, existen, dentro de este controlador, una serie de funciones que sirven para gestionar la interacción entre el usuario y el tiempo de la pista de audio. Estos métodos son:

- - *(void)updateDisplay*: que actualiza los componentes que ve el usuario a medida que la reproducción va transcurriendo.
- - *(IBAction)currentTimeSliderValueChanged:(id)sender*: este método se ejecuta cuando cambia el valor del *slider*. Para que funcione como se desea, ha de funcionar de forma conjunta con el siguiente método.
- - *(IBAction)currentTimeSliderTouchUpInside:(id)sender*: se ejecuta cuando el *slider* es tocado. Su código aparece en el Bloque de código 7.3.

Bloque de código 7.3: Método *currentTimeSliderTouchUpInside*.

```
- (IBAction)currentTimeSliderTouchUpInside:(id)sender {
    [self.reproductor stop];
    self.reproductor.currentTime = self.currentTimeSlider.value;
    [self.reproductor prepareToPlay];
    estaEnPlay = NO;
    [self playYPause:self];
}
```

7.1.3. Resultado Galería

Esta view tiene dos puntos clave en lo que a la implementación se refiere. La descarga de varias imágenes de forma asíncrona y la gestión de varios gestos táctiles simultáneamente.

En cuanto a la descarga de varias imágenes de forma asíncrona, es necesario que sea de esta forma ya que, de no ser así, si se quiere ver una galería que tenga 100 imágenes, no se podría ver ninguna (o tal vez sólo la primera) si esta descarga no se hiciera de forma asíncrona, ya que no se podría interactuar con la interfaz hasta que la descarga de todas las imágenes hubiera sido completada. En cambio, al realizar esta descarga de forma asíncrona, cuando la primera imagen

se ha descargado, se visualiza y se puede interactuar con ella: cambiar el *zoom* y moverla. Y esto, mientras las demás siguen descargándose. Para conseguir esto, lanzamos, utilizando *AFNetworking* 3.5.2, un hilo para cada imagen y, cuando se van completando las descargas, almacenamos en un vector las imágenes descargadas, tal y como aparece en el Bloque de código 7.4.

Bloque de código 7.4: Descarga asíncrona de varias imágenes.

```
    vectorImagenes = [NSMutableArray arrayWithCapacity:vectorURLs.count];
    for(int i = 0; i < vectorURLs.count; i++)
        [vectorImagenes addObject:[NSNull null]];
    for(int i = 0; i < vectorURLs.count; i++)
    {
        NSURL * myUrl = [NSURL URLWithString:[NSString stringWithFormat:@"%@",vectorURLs[i]]];

        NSMutableURLRequest* request = [[NSMutableURLRequest alloc] initWithURL:myUrl];
        AFHTTPRequestOperation* op = [[AFHTTPRequestOperation alloc] initWithRequest:request];
        op.responseSerializer = [AFImageResponseSerializer serializer];
        [op setCompletionBlockWithSuccess:^(AFHTTPRequestOperation* operation, id
            responseObject){
            UIImage* image = responseObject;
            [vectorImagenes replaceObjectAtIndex:i withObject:image];
        } failure:^(AFHTTPRequestOperation* operation, NSError* error) {
            NSLog(@"Fallo al descargar imágenes!");
        }];

        [op start];
    }
```

La otra dificultad de este controlador es, como se ha comentado anteriormente, la gestión simultánea de varios gestos táctiles. Esto conlleva trabajo por el hecho de que el gesto de mover la imagen hacia un lado y el gesto de deslizar hacia un lado para cambiar de imagen son el mismo para el dispositivo. Para solucionarlo, se han añadido algunas restricciones, como, por ejemplo, que solo se pueda mover una imagen cuando el *zoom* ha cambiado (si el *zoom* es el original, la imagen se ajusta a la pantalla, por lo que no hay motivo para moverla. Luego, para poder cambiar de imagen, se ha establecido como necesario el que la imagen visualizada esté a *zoom* 1, es decir, a tamaño original (se ha añadido al interfaz un botón que reinicia el estado de la imagen automáticamente. El Bloque de código 7.5 es el que se ejecuta cuando se desliza el dedo hacia la derecha. Dentro de este método, lo primero es comprobar si la imagen tiene el tamaño original. Además, se ha implementado una pequeña animación para el cambio de imagen.

Bloque de código 7.5: Cambio de imagen.

```
- (void)swipeRightAction:(id)ignored{
    if (tamOriginal) {
        if (numeroImagenActual == 0) {
            numeroImagenActual = [vectorImagenes count] - 1;
        }else{
            numeroImagenActual--;
        }

        [lblNumeroImagenActual setText: [NSString stringWithFormat:@"%d", numeroImagenActual +
            1]];

        [UIView transitionWithView:galeriaView
```

```
                          duration:0.5f
                           options:UIViewAnimationOptionTransitionFlipFromLeft
                        animations:^{
                            [galeriaView setImage:vectorImagenes[numeroImagenActual]];
                            heightOriginal = galeriaView.frame.size.height;
                            widhtOriginal = galeriaView.frame.size.width;
                            primerOrigen = galeriaView.center;
                        } completion:NULL];
    }

}
```

7.1.4. Resultado Vídeo

Este es de los controladores más largos del proyecto, ya que tiene mucha funcionalidad diferente. Aparte del reproductor de vídeo, se gestiona la rotación de la pantalla (que no es trivial) y el control de gestos (para que el usuario pueda tocar el reproductor de vídeo para que aparezcan los botones que permiten actuar sobre el vídeo).

Gestionar la descarga del archivo de vídeo, iniciar el reproductor de vídeo y controlar la sincronía entre el *slider* y el tiempo del vídeo, se realiza de la misma forma que en el reproductor de audio, sólo que ahora sobre un objeto del tipo *MPMoviePlayerController*, por lo que no es necesario entrar en más detalle.

La rotación de la pantalla sí que ha generado diversos problemas. Los clientes pidieron que, al voltear el *iPhone*, el vídeo apareciera en pantalla completa y el *slider* que representa en que momento del video nos encontramos y el botón de *play/pause* mantuvieran la misma funcionalidad que cuando el dispositivo está en vertical, pero ahora debían esconderse cuando el usuario estuviera unos segundos sin pulsar sobre el dispositivo, y volver a aparecer cuando el usuario pulsara.

Para solucionar el problema de la rotación, el primer paso fue permitir la orientación horizontal para todo el proyecto. No obstante, puesto que las demás vistas no debían de girar, se sobreescribió en los demás controladores el método *shouldAutorotate*, devolviendo no, para que solo se permitiera girar el dispositivo en la vista de vídeo. No obstante, la vista sólo debe rotar si el vídeo ya está cargado.

Bloque de código 7.6: Permitir rotación de la pantalla cuando el vídeo esté cargado.

```
- (BOOL) shouldAutorotate {
    if(hayVideo) return YES;
    else return NO;
}
```

Para implementar la desaparición de los componentes cuando el usuario no toca la pantalla durante algunos segundos y la aparición de estos, se pensó que sería suficiente con asignar un reconocedor de gesto tipo *tap* (pulsación) y detectar cuando el usuario pulsaba sobre la pantalla. No obstante, cuando se está reproduciendo un vídeo, esto no funciona así. Cuando se pulsa un reproductor de vídeo se ejecuta el método *touchesBegan* del delegado del reproductor de vídeo,

y cuando se levanta el dedo, el método *touchesEnded*. Son estos métodos los que hay que utilizar para gestionar las pulsaciones sobre el reproductor de vídeo. De la forma que se gestionan las pulsaciones en las demás vistas (sobre otros objetos) del proyecto, en este caso no funciona.

Bloque de código 7.7: Métodos *touchesBegan* y *touchesEnded*.

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{
    moviePlayerSiendoTocado = YES;
    [self handleTap];
}

-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    moviePlayerSiendoTocado = NO;
}
```

Por último, en el método *handleTap*, llamado desde *touchesBegan*, se realiza la acción de esconder los botones mediante una animación, para que el resultado sea estético. Y, por último, se vuelve a iniciar la cuenta atrás para que, al terminar, se escondan los componentes nuevamente.

Bloque de código 7.8: Animación y cuenta atrás.

```
    [UIView transitionWithView:btnPlayPause
                      duration:0.4
                       options:UIViewAnimationOptionTransitionCrossDissolve
                    animations:NULL
                    completion:NULL];

    btnPlayPause.hidden = NO;
    contadorTiempo = [NSTimer
                      scheduledTimerWithTimeInterval:5
                      target:self selector:@selector(esconderBotones)
                      userInfo:nil repeats:NO];
```

7.1.5. Resultado *Web*

Desde el mismo controlador, se gestiona una *web view* para mostrar tanto el contenido que obtiene desde una cadena de contenido *HTML*, como para cargar una página *web* de una *URL*. Simplemente, antes de abrir la view, el controlador que inicie este, debe decirle el tipo de información que le pasa para que este controlador, en función de esto, ejecute un código u otro, tal y como se muestra en el Bloque de código 7.9.

Bloque de código 7.9: Tipo *HTML* o *URL*.

```
- (void)viewDidAppear:(BOOL)animated {
    visorWeb.delegate = self;

    if ([tipo isEqual: @"HTML"]) {
        [MBProgressHUD showHUDAddedTo:self.view animated:YES]; //Activity Indicator loading
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

            [visorWeb loadHTMLString:html baseURL:nil];

        });
    }else if ([tipo isEqual:@"URL"]){
```

```
        [MBProgressHUD showHUDAddedTo:self.view animated:YES]; //Activity Indicator loading
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
            [visorWeb loadRequest: [NSURLRequest requestWithURL:[NSURL URLWithString:url]]];

        });
    }
}
```

Para que el usuario sepa cuando la *web* ha terminad de cargarse, indicamos que este controlador es *UIWebViewDelegate*. De esta forma, cuando se inicia mostramos una animación de cargando y, cuando termina de cargarse, gracias a haber identificado al controlador como delegado de *UIWebView*, se ejecuta el método *webViewDidFinishLoad*, que sobreescribimos en nuestro controlador, y en el que indicamos a la animación de cargando que desaparezca.

Bloque de código 7.10: Página *web* cargada.

```
- (void)webViewDidFinishLoad:(UIWebView *)webView{
    [self liberarLoading];
}

-(void)liberarLoading{
    dispatch_async(dispatch_get_main_queue(), ^{

        [MBProgressHUD hideHUDForView:self.view animated:YES]; //hide that after your Delay
            Loading finished fromDatabase
    });
}
```

7.1.6. Core data: base de datos interna

Para almacenar los favoritos de cada usuario, se ha determinado que lo mejor es, puesto que esta información no ocupa prácticamente espacio y es dependiente de cada dispositivo (cada dispositivo tendrá sus favoritos propios), diseñar una base de datos interna, para almacenar dicha información.

Para implementarla, existen varias alternativas. Se puede crear una base de datos *SQLite*, por ejemplo, o crear una base de datos utilizando *Core Data* [8], que es el *framework* para persistencia desarrollado por *Apple*. Puesto que el supervisor de la empresa experto en *iOS* advirtió de la potencia de dicho *framework*, esta fue la opción elegida.

Para crear una base de datos con *Core Data*, lo primero es crear un archivo con la extensión *xcdatamodel*. Desde él, mediante un asistente gráfico, se crean las entidades y atributos de la base de datos. En nuestro caso, es muy sencillo. Nuestra base de datos tiene la entidad *Favorito*, que contiene los atributos *id* y *json*, ambos cadenas.

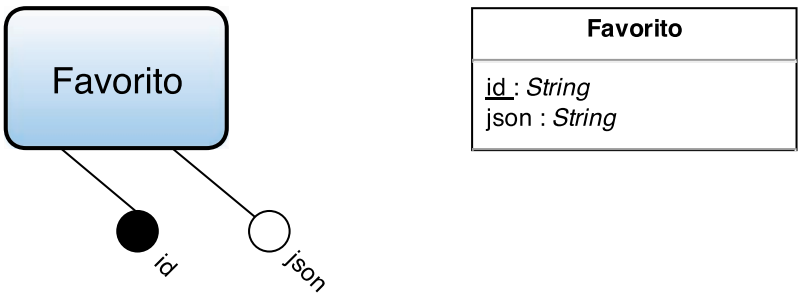


Figura 7.1: La Figura 7.1 incluye los diseños de la base de datos interna.

El siguiente paso es crear el modelo de nuestra base de datos. Esto son, en este caso, un fichero *.h* y otro *.m* que contengan la siguiente información.

Bloque de código 7.11: *Favorito.h*.

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>

@interface Favorito : NSManagedObject

@property (nonatomic, retain) NSString * id;
@property (nonatomic, retain) NSString * json;

@end
```

Bloque de código 7.12: *Favorito.m*.

```
#import "Favorito.h"

@implementation Favorito

@dynamic id;
@dynamic json;

@end
```

Una vez creado el modelo de la base de datos, en cada clase en la que sea necesario acceder a la base de datos es necesario tener, en el fichero *.h* de la clase, los objetos y el método:

Bloque de código 7.13: Objetos y método necesarios para operar con *Core Data*.

```
@property (readonly, strong, nonatomic) NSManagedObjectContext *managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
    *persistentStoreCoordinator;

- (NSURL *)applicationDocumentsDirectory;
```

Y, en los ficheros *.m* de estas clases, es necesario implementar los métodos *managedObject-*

Context, *managedObjectModel*, *persistentStoreCoordinator* y *applicationDocumentsDirectory*. No obstante, estos métodos no han sido implementados para este proyecto, si no que han sido proporcionados por terceros.

Llegados a este punto, se puede operar con la base de datos fácilmente. Por ejemplo, para insertar un favorito, es suficiente con crear una instancia de la clase *Favorito*, asignarle las propiedades y llamar al método *save* de la clase *NSManagedObjectContext*. No obstante, si no se puede insertar en la base de datos por incumplir una regla de integridad, se producirá una excepción que hay que controlar, tal y como se muestra en el Bloque de código 7.14, por ejemplo.

Bloque de código 7.14: Guardar favorito en *Core Data*.

```
Favorito *fav = [NSEntityDescription insertNewObjectForEntityForName:@"Favorito"
    inManagedObjectContext:context];
fav.id = [jsonImagen objectForKey:@"_id"];

NSData *jsonData = [NSJSONSerialization dataWithJSONObject:jsonImagen
                                                   options:(NSJSONWritingOptions)
                                                       (NSJSONWritingPrettyPrinted)
                                                     error:nil];
NSString *jsonString = [[NSString alloc] initWithData:jsonData
    encoding:NSUTF8StringEncoding];

fav.json = jsonString;

NSError *error;
if (![context save:&error]) {
    NSLog(@"Vaya, no se ha podido guardar: %@", [error localizedDescription]);
}
```

Para listar o borrar elementos de la base de datos, la forma de proceder es muy similar. De este modo, podemos acceder a la base de datos sin emplear lenguaje *SQL*, y de una forma sencilla y limpia.

7.1.7. Barra superior

La barra superior que comparten todas las vistas que muestran resultados tiene la forma y forma de funcionar de las *Action Bar* de *Android*. Esto se debe a que, como cabe recordar, este proyecto forma parte de otro más amplio, en el que se tiene que implementar la misma funcionalidad para el sistema operativo *Android*. Por lo tanto, una de las peticiones de los clientes es que las interfaces sean lo más parecidas posibles.

Figura 7.2: *Screen* de la barra superior.

Lograr que una *Navigation Bar* de *iOS* parezca una *Action Bar* de *Android* no es un trabajo instantáneo. El botón de la izquierda de la barra, que en *Android* es para volver atrás, en este caso se ha implementado indicando que, al ser clicado, se debe ejecutar el controlador de la vista de inicio. En la parte derecha, en cambio, aparecen varios botones. Para conseguir que los dos aparezcan en esa zona, es necesario añadirlos a un vector y asignar ese vector como botón de la *Navigation Bar*. Es decir, en realidad, el botón es un vector de botones.

Bloque de código 7.15: Vector de botones.

```
UIBarButtonItem *btnCompartir = [[UIBarButtonItem alloc] initWithTitle:@" "
    style:UIBarButtonItemStylePlain target:self action:@selector(compartir)];
[btnCompartir setTitle:[NSString stringWithFormat:@" %@ ", [NSString
    fontAwesomeIconStringForEnum:FAShare]]];
[btnCompartir setTitleTextAttributes:attributes forState:UIControlStateNormal];

btnFavoritos = [[UIBarButtonItem alloc] initWithTitle:@" " style:UIBarButtonItemStylePlain
    target:self action:@selector(gestionarFavoritoActual:)];
//Comprobamos si el resultado es favorito y actulizamos el botón de favoritos para que se
    pinte como toque
[self comprobarFavoritos];
[self actualizarBotonFavorito];
[btnFavoritos setTitleTextAttributes:attributes forState:UIControlStateNormal];

NSArray *myButtonArray = [[NSArray alloc] initWithObjects: btnCompartir, btnFavoritos, nil];

navigationBar.rightBarButtonItems = myButtonArray;
```

Estos dos botones sirven para gestionar favoritos y para la funcionalidad de compartir. Ya se ha comentado la implementación de la gestión de favoritos en el apartado 7.1.6. En cuanto a la funcionalidad de compartir, utilizando un objeto de la clase *UIActivityViewController*, se presenta una vista en la que aparecen las redes sociales en las que el usuario se ha registrado en el *iPhone* y, al pulsar sobre alguna, se abre la aplicación correspondiente, dando al usuario la posibilidad de compartir el resultado que estaba visualizando. Además, se inicializa el mensaje

que se compartirá con un mensaje por defecto de la aplicación, que el usuario puede cambiar si lo desea.

Bloque de código 7.16: Vector de botones.

```
- (IBAction)compartir {

    NSString *mensaje = [NSString stringWithFormat:@"¡Mirad como mola %@!¡Lo he descubierto
        gracias a CazaPromo!", self.nombre];

    self.activityViewController = [[UIActivityViewController alloc]
        initWithActivityItems:@[mensaje] applicationActivities:nil];
    [self presentViewController:self.activityViewController animated:YES completion:nil];

}
```

7.1.8. Vista de inicio, favoritos y acerca de

Cuando se inicia la aplicación, aparece la vista de inicio. En la parte superior de dicha vista, se encuentra una *tab bar* o barra de pestañas. Para desplazarse entre las diferentes pestañas, se puede tocar la pestaña a la que se quiere ir, o bien deslizar el dedo para pasar de pestaña. Otra vez, se ha intentado imitar en el *iPhone* la transición de cambio de pestaña que existe en el sistema *Android*. No obstante, este tipo de transición en el que se ven las dos ventanas y como una empuja a la otra para ocupar su lugar, no existe en *iOS*, así que ha sido necesario crearla.

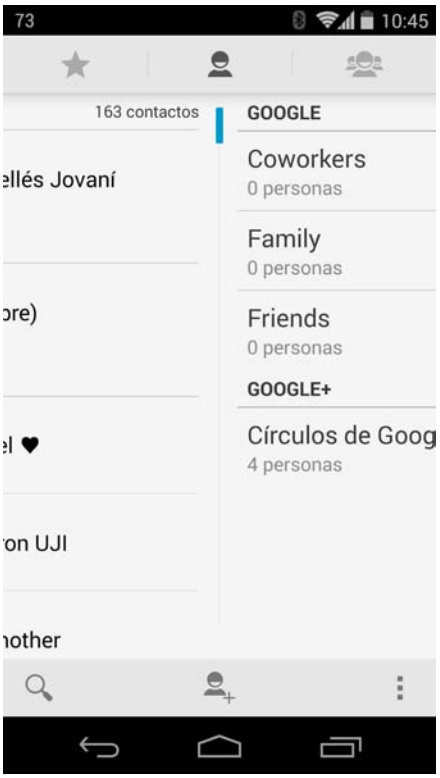


Figura 7.3: Transición de cambio de pestaña en *Android*.

Para simular esta transición en *iOS*, ha sido necesario crear una nueva animación en la que se deja de presentar un controlador para dar paso a otro, mientras las vistas de ambos están en pantalla y pasan a ocupar la nueva el sitio de la vieja, empujándola.

Bloque de código 7.17: Animación para *tab bar*.

```
- (void) presenta:(UIViewController *)modalViewController withPushDirection: (NSString *)
    direction {
    [CATransaction begin];

    CATransition *transition = [CATransition animation];
    transition.type = kCATransitionPush;
    transition.subtype = direction;
    transition.duration = 0.25f;
    transition.fillMode = kCAFillModeForwards;
    transition.removedOnCompletion = YES;

    [[UIApplication sharedApplication].keyWindow.layer addAnimation:transition
        forKey:@"transition"];
    [[UIApplication sharedApplication] beginIgnoringInteractionEvents];
    [CATransaction setCompletionBlock: ^ {
        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(transition.duration *
            NSEC_PER_SEC)), dispatch_get_main_queue(), ^ {
            [[UIApplication sharedApplication] endIgnoringInteractionEvents];
        });
    }];

    [self presentViewController:modalViewController animated:NO completion:nil];

    [CATransaction commit];

}
```

Vista de Favoritos

Para implementar esta vista, se declara como *UITableViewDelegate* y *UITableViewDataSource* el controlador de Favoritos. Esto nos obliga a especificar los métodos:

- *- (void) tableView: (UITableView *) tableView didSelectRowAtIndexPath: (NSIndexPath *) indexPath*: este método es llamado cuando se pulsa una de las filas de la tabla. Puesto que cada fila es un favorito, cuando se pulsa una tabla hay que obtener de qué favorito se trata y abrir una vista de resultado de dicho favorito.

Bloque de código 7.18: Obtener favorito de la lista y abrir *view* resultado.

```
Favorito *fav = [listaFavoritos objectAtIndex:indexPath.row];
NSError *e;
NSDictionary *json =
[NSJSONSerialization JSONObjectWithData: [fav.json
    dataUsingEncoding:NSUTF8StringEncoding]
                                options: NSJSONReadingMutableContainers
                                  error: &e];
[self abrirViewResultado:json];
```

- *- (NSInteger) tableView: (UITableView *) tableView numberOfRowsInSection:(NSInteger) section*: devuelve el número de filas que va a tener la tabla. En este caso, es necesario acceder a la base de datos para conocer el número de favoritos que tiene el dispositivo.

- *- (CGFloat) tableView: (UITableView *) tableView heightForRowAtIndexPath: (NSIndexPath *) indexPath*: debe devolver la altura de cada fila. En esta tabla, cada fila se construye tomando una subvista como referencia para, luego, rellenar sus campos con los datos de cada favorito concreto. Por lo tanto, en este caso debe devolver la altura en píxeles de dicha *subview*, que es 100.

- *- (UITableViewCell *) tableView: (UITableView *) tableView cellForRowAtIndexPath: (NSIndexPath *) indexPath*: se llama para cada celda que se construye, y en él se le debe indicar que datos corresponden a dicha celda. Obtenemos estos datos de un vector de Favoritos, que se ha cargado previamente. Asignamos, a la celda *n*, el favorito de la posición *n-1* del vector.

Vista de Acerca de

Esta vista es de las más sencillas de la aplicación. Aparte de la barra de pestañas, contiene un botón para visualizar el tutorial y un *scroll* en el que aparecen las licencias de las librerías utilizadas.

7.1.9. Auto Layout

El *Auto Layout* [3], es una herramienta presente en el *Xcode* que ayuda a ajustar la posición de los componentes en la vista para que la posición sea la correcta, independientemente del tamaño de la pantalla del dispositivo en el que se ejecute la aplicación. Esto es bastante útil (cuando se sabe utilizar) ya que la pantalla del *iPhone 4* es más corta que la de *iPhone 5*, y esta aplicación debe verse bien en ambas.

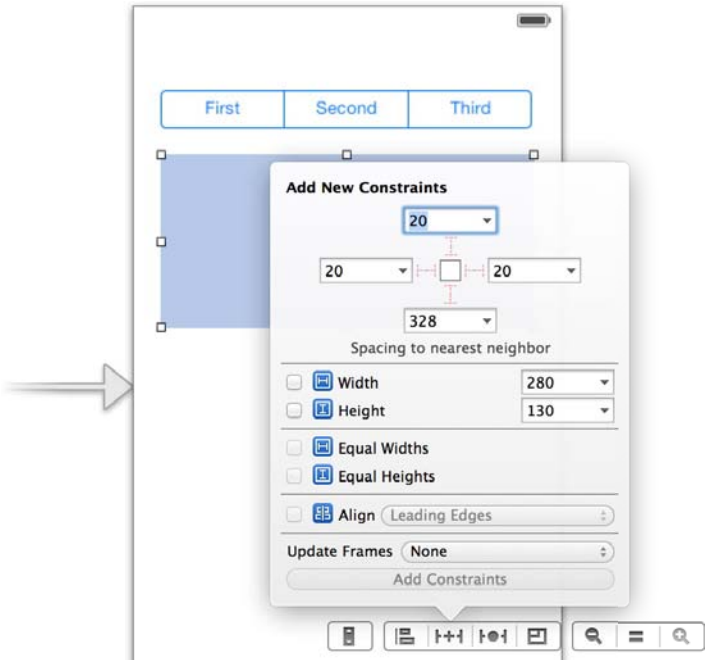


Figura 7.4: Añadiendo restricciones en *Auto Layout*.

Este sistema fija restricciones de posición de los componentes con respecto a los bordes de la pantalla o a otros componentes. Para esta aplicación, se ha conseguido que las vistas se vean correctamente en ambos dispositivos fijando los elementos superiores a la parte superior, los inferiores a la inferior, y dejando libertad de altura a los elementos centrales para que esta varíe en función del espacio restante.

7.2. Implementación del servidor

En este apartado se explican los detalles más importantes o menos comunes sobre la implementación del servidor. El servidor, escrito en *Python*, se divide en 4 ficheros:

- *modelo.py*
- *baseDeDatos.py*
- *servidor.py*
- *reconocimiento.py*

7.2.1. Fichero *modelo.py*

En este fichero de declaran las clases *Imagen* y *Estado*, que se utilizan para añadir elementos a la base de datos y para construir la respuesta que se devuelve al cliente, después de ejecutar

el reconocimiento de una imagen. Además, contiene de un par de métodos que vale la pena explicar.

La clase *Imagen*

Tal y como se ha visto en el diagrama de clases 6.18 la clase *Imagen* es una clase abstracta, que extienden las clases concretas *ImagenInfor*, *ImagenWeb*, *ImagenUrl*, *ImagenGaleria*, *ImagenVideo* y *ImagenAudio*. Para declarar una clase como abstracta en *Python*, es necesario importar una librería externa al propio lenguaje. Luego, antes del constructor de la clase, se escribe la instrucción *__metaclass__ = ABCMeta* para indicar que la clase es abstracta. Solo falta, en cada una de las clases que extienden a la clase abstracta, declararlas como hijas de la clase *Imagen*. Esto se hace escribiendo el nombre de la clase madre entre paréntesis después del nombre de la clase hija. En el Bloque de código 7.19 se muestra un ejemplo de esto.

Bloque de código 7.19: Clase *Imagen* abstracta en *Python*.

```
class Imagen(object):
    ''' Objeto imagen, en el cual se almacenan los datos correspondientes a una imagen. '''
    # Marcamos la clase como abstracta.
    __metaclass__ = ABCMeta
    def __init__(self, _id, tipo, nombre, descripcion):
        self._id = _id
        self.tipo = tipo
        self.nombre = nombre
        self.descripcion = descripcion
    def getId(self):
        return self._id
    def __str__(self):
        return 'Imagen('+'Id: '+str(self._id)+' Nombre: '+self.nombre+')'

class ImagenInfo(Imagen):
    ''' Subobjeto de la clase imagen, correspondiente a las imágenes de tipo Información. '''
    def __init__(self, _id, nombre, descripcion, autor):
        super(ImagenInfo, self).__init__(_id, 'INFO', nombre, descripcion)
        self.autor = autor
```

La clase *Estado*

La clase estado es la que se devuelve, dentro de un *JSON*, al cliente *iPhone*. Esta clase contiene una imagen (de tipo *Imagen*, explicada en el apartado 7.2.1), un código (que se le asigna dependiendo del resultado del reconocimiento) y un diccionario con valores *String* y con la clave del mismo tipo que el código. A continuación se incluye el código que refuerza la explicación:

Bloque de código 7.20: Clase *Estado*.

```
class Estado(object):
    ''' Clase para crear estados de las peticiones al servidor '''
    def __init__(self, codigo, imagen):
        mensajes = {0 : 'No se ha podido reconocer la imagen',
                    1 : 'La imagen ha sido reconocida de manera correcta.',
```

```
                2 : 'La imagen tiene poca calidad.'}
        self.codigo = codigo
        self.mensaje = mensajes[codigo]
        self.imagen = imagen

    def __str__(self):
        return 'Estado (Codigo: ' + str(self.codigo) + ', Mensaje: ' + self.mensaje + ')'
```

La función *crearImagenDesdeJSON*

La base de datos, implementada en el *SGBD MongoDB* devuelve los objetos en forma de *JSON*. Esta clase se encarga de transformar los *JSON* que contienen una imagen, cualquiera que sea su tipo, y la convierten a un objeto de la clase correspondiente. Su funcionamiento es, primero, averiguar el tipo de imagen (leyendo el campo *tipo* del *JSON*), para luego, dependiendo de cuál sea este, generar una instancia de una clase u otra.

Bloque de código 7.21: Función *crearImagenDesdeJSON*.

```
def crearImagenDesdeJSON(imagenJSON):
    ''' Recibe un JSON y devuelve un objeto de tipo Imagen correspondiente a este. '''
    nombre = imagenJSON['nombre']
    descripcion = imagenJSON['descripcion']
    tipo = imagenJSON['tipo']
    _id = str(imagenJSON['_id'])
    if tipo == 'INFO':
        imagen = ImagenInfo(_id, nombre, descripcion, imagenJSON['autor'])
    elif tipo == 'WEB':
        imagen = ImagenWeb(_id, nombre, descripcion, imagenJSON['contenidoHTML'])
    elif tipo == 'URL':
        imagen = ImagenUrl(_id, nombre, descripcion, imagenJSON['url'])
    elif tipo == 'GALERIA':
        imagen = ImagenGaleria(_id, nombre, descripcion, imagenJSON['listaURLs'])
    elif tipo == 'VIDEO':
        imagen = ImagenVideo(_id, nombre, descripcion, imagenJSON['urlVideo'])
    elif tipo == 'AUDIO':
        imagen = ImagenAudio(_id, nombre, descripcion, imagenJSON['urlAudio'])
    else:
        imagen = None

    return imagen
```

La función *JSONificar*

Bloque de código 7.22: Función *crearImagenDesdeJSON*.

```
def JSONificar(objeto):
    ''' Recibe un objeto y devuelve un JSON con su contenido. '''
    return json.dumps(objeto, default = lambda o: o.__dict__)
```

Esta función, que a primera vista puede parecer simple por su longitud, no lo es en absoluto. Su función es convertir a *JSON* un objeto. Para ello, llama a una función de la librería

externa *json* que se encarga de hacer justamente esto: el método *dumps()*. No obstante, si no se especifica que el parámetro *default* que recibe debe ser la función *lambda o: o.__dict__*, el método *json.dumps()* no funciona correctamente, para objetos de la clase *Imagen* o de las clases que la extienden.

7.2.2. Fichero *baseDeDatos.py*

Este fichero interactúa con la base de datos de *MongoDB*. En sus atributos figuran una conexión, una referencia a la base de datos, rutas a las imágenes de la base de datos y a los accesos, y un conjunto de extensiones permitidas:

Bloque de código 7.23: Constructor de la clase *BaseDeDatos*.

```
class BaseDeDatos(object):
    ''' Objeto para gestionar la comunicaciónn con la base de datos MongoDB. '''
    def __init__(self, rutaImagenesBaseDeDatos, rutaImagenesAcesos):
        self.connection = Connection()
        self.db = self.connection.ServidorReconocimientoIMG
        self.rutaImagenesBaseDeDatos = rutaImagenesBaseDeDatos
        self.rutaImagenesAcesos = rutaImagenesAcesos
        self.EXTENSIONES = set(['png', 'jpg', 'jpeg', 'gif'])
```

Una vez definida la referencia de la conexión de la base de datos, las consultas de la base de datos se realizan de la siguiente forma:

- Mediante el método *find()* se realizan consultas. Este método puede recibir parámetros variados para restringir la consulta. Por ejemplo, sin parámetros devuelve todos los elementos de la sección de la base de datos sobre la que se realiza, mientras que si añadimos el parámetro *find(tipo = 'web')*, solamente devuelve aquellos elementos que contienen, en el campo tipo, el valor *'web'*.
- Hay que limitar el cursor que devuelve el método *find()* para poder convertirlo en lista.
- Ya podemos recorrer la lista con los elementos de la consulta.

El fragmento de código siguiente ejemplifica la explicación:

Bloque de código 7.24: Consulta para obtener una lista con las imágenes de la base de datos.

```
    def getImagenes(self):
        cursor = self.db.imagenes.find()
        cursor = cursor.limit(cursor.count())
        lista = list(cursor)
        imagenes = []
        for elem in lista:
            imagenes.append(crearImagenDesdeJSON(elem))
        return imagenes
```

Para insertar un elemento en la base de datos, el procedimiento es aún más sencillo. Basta con convertir a *JSON* el elemento a guardar y insertarlo en la base de datos. No obstante, hasta

que se encontró la instrucción exacta que hacía que el programa funcionara, se probaron muchas variantes, sin éxito.

Bloque de código 7.25: Insertando en la base de datos.

```
    def insertarAcceso(self, acceso):
        ''' Inserta un objeto de tipo Acceso en la base de datos. '''
        accesoJSON = self.JSONificar(acceso)
        self.db.accesos.insert(json.loads(accesoJSON))
```

7.2.3. Fichero *servidor.py*

Este fichero es el que incluye los servicios *REST* y se encarga comunicar los recursos de los demás ficheros. Es decir, crea instancias de las clases de los otros ficheros y las pone en contacto para que puedan interactuar entre ellas.

En el *main* del fichero, se crea una instancia de la clase *BaseDeDatos*, otra de la clase *Reconocedor* (a la que se le inyectan como parámetros un comparador y dos detectores, que en la sección 3.1 se explican), se añade un sistema de *logger* para conservar información sobre lo que ocurre en el servidor, y se ejecuta el servidor.

Bloque de código 7.26: *Main* del servidor.

```
if __name__ == '__main__':
    db = BaseDeDatos(RUTA_IMAGENES_BASEDEDATOS, RUTA_IMAGENES_ACCESOS)
    comparador = ComparadorFLANN(flannIndexKDTree = 4, coeficienteDistancia = 0.75)
    detector = DetectorSURF(hessianThreshold = 1000, nOctaves = 2, nOctaveLayers = 2)
    detectorAux = DetectorSURF(hessianThreshold = 500, nOctaves = 2, nOctaveLayers = 4)

    reconocedor = Reconocedor(db, detector, comparador, detectorAux)

    file_handler = RotatingFileHandler(RUTA_APP_LOG+'/servidor.log', maxBytes=1024 * 1024 *
        100, backupCount=20)
    file_handler.setLevel(logging.DEBUG)
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
    file_handler.setFormatter(formatter)
    app.logger.addHandler(file_handler)

    app.run(host='0.0.0.0', port=15000,threaded=True)
```

Para que los clientes puedan, mediante una *URL*, acceder a un servicio del servidor, hay que marcar cada método con la anotación *@app.route('urldelmétodo')*, tal y como se muestra en el Bloque de código 7.27 para el método que devuelve las imágenes en miniatura de cada resultado (para mostrarlas en el cliente, en la vista de favoritos). Podemos observar en éste ejemplo, además, cómo pasar parámetros a las funciones desde la *URL*: añadiendo entre ¡ y ¿ el nombre de la variable que recibe el método.

Bloque de código 7.27: Servicio *getThumbnail(identificador)*.

```
@app.route('/thumbnail/<identificador>')
def getThumbnail(identificador):
    ''' Servicio que recibe el identificador de una imagen y devuelve la imagen thumbnail
        correspondiente. '''
```

```
    ruta = os.path.join(RUTA_IMAGENES_BASEDEDATOS, identificador, 'thumbnail.jpg')

    return send_file(ruta, mimetype='image/jpg')
```

7.2.4. Fichero *reconocimiento.py*

Este fichero contiene el algoritmo para reconocer imágenes y, además, la paralelización en hilos del mismo, para obtener una respuesta en un tiempo mucho menor. Puesto que es muy extenso y contiene muchos conceptos complicados, este apartado se divide en subapartados.

La clase *ImagenAnalizada*

Se ha creado una clase en que contiene, además de la imagen en sí y su ruta, los *key points* y descriptores de dicha imagen, que son el resultado de aplicar a la imagen un algoritmo detector. De esta forma, no es necesario aplicar el algoritmo a la imagen cada vez que se necesitan los descriptores, si no que, al iniciar el servidor, se detectan los puntos clave y descriptores de todas las imágenes de la base de datos y se guardan en una lista de objetos del tipo *ImagenAnalizada*.

La clase *Detector*

Puesto que, tal y como se explica en la sección de estudio previo 3.1, no se puede conocer, a priori, cuál de todos los algoritmos detectores de características es más beneficioso para este problema, se decide implementar tres de los más populares: *SURF*, *SIFT* y *ORB*.

De la misma forma que se ha explicado antes con la clase *Imagen*, en la sección 7.2.1, la clase *Detector* es una clase abstracta, que extienden las clases concretas *DetectorSURF*, *DetectorSIFT*, y *DetectorORB*. No obstante, encontramos como diferencia que, en este caso, existe un método abstracto, que se declara de la siguiente forma:

Bloque de código 7.28: Método abstracto *detectar(imagen)*.

```
    @abstractmethod
    def detectar(self, imagen):
        '''
        Args:
            imagen(ImagenAnalizada)

        Returns:
            Un objeto del tipo ImagenAnalizada con los descriptores y keyPoints resultantes del
                análisis.
        '''
        pass
```

Cada una de las clases hijas de *Detector* tiene la misma estructura. Como atributos, cuentan con los parámetros que el algoritmo requiere. Además, implementan el método que en el padre

se declara como abstracto. Dentro de este método, cada clase obtiene los puntos clave y los descriptores de la imagen utilizando el algoritmo correspondiente, proporcionado por *OpenCV*.

La clase *Comparador*

Con esta clase ocurre lo mismo que con la clase *Detector*. Se ha implementado la función de *matching*, que compara los descriptores de dos imágenes, mediante dos algoritmos: *FLANN* y *BFMatcher*. Por lo tanto, de la misma forma que en la clase *Detector*, existe una clase *Comparador* abstracta, con el método abstracto *comparar(imagen1, imagen2)*, que las clases hijas implementan mediante sus respectivos algortimos.

No obstante, existe aquí una pequeña diferencia. Puesto que, para este proyecto, el problema es comparar una imagen contra un conjunto, en la clase abstracta *Comparador*, se implementa el método *compararVarios(imagenesEntrnamiento, imagenConsulta)*, que recorre las imágenes de la base de datos y llama al método abstracto *comparar*, que compara cada imagen con la recibida en la petición al servidor utilizando el algoritmo concreto seleccionado.

De esta forma, solamente es necesario implementar un método en todas las clases hijas: *comparar*. Esto se debe a que *compararVarios* realiza la misma función, aunque modificando el algoritmo que lleva a cabo la comparación.

Bloque de código 7.29: Fragmento del método *compararVarios*.

```
        for i in range(len(imagenesEntrenamiento.values())):
            coincidencias, t = self.comparar(imagenesEntrenamiento.values()[i], imagenConsulta)
            listaCoincidencias.append(len(coincidencias))
            porcentaje = listaCoincidencias[i]*100.0 /
                len(imagenesEntrenamiento.values()[i].getDescriptores())
            porcentajes.append(porcentaje)
```

Las clases *ComparadorFLANN* y *ComparadorBFMatcher* también mantienen una estructura idéntica. Tienen como atributos los parámetros de cada algoritmo e implementan el método comparar. En este método, utilizan funciones de *OpenCV* para comparar los descriptores de dos imagen. De esta forma, se obtiene un vector en el que se almacenan los descriptores que se encuentran en las dos imágenes comparadas. De este modo, cuando se obtienen los vectores resultantes de comparar la misma *imagenConsulta* con todas y cada una de las *imagenesEntrenamiento*, analizando este vector de coincidencias, podemos saber a cuál de todas las imágenes de la base de datos se parece más.

No obstante, no es suficiente con contar el número de coincidencias entre las imágenes para saber a qué imagen de la base de datos se corresponde la fotografía recibida. Y esto ocurre porque no todas las imágenes tienen el mismo número de descriptores. Un cuadro, por ejemplo, puede tener 3000 descriptores, mientras que una tarjeta con un logotipo puede tener menos de 100. Entonces, es posible que, al comparar una fotografía de la tarjeta con la imagen de la tarjeta encuentre 50 coincidencias, mientras que al comparar la misma fotografía con el cuadro encuentre 200. No obstante, 50 de 100 es el 50 % y 200 de 3000 es el 6.6667 %. Este porcentaje es el que se ha tomado como determinante para establecer el ratio de similitud entre dos imágenes.

Paralelizazión del reconocimiento

Cuando se envía una imagen a reconocer, se obtienen sus descriptores y se comparan con los descriptores de las imágenes de la base de datos. Cuando el número de imágenes de la base de datos crece, el tiempo que se tarda en reconocer la imagen, también. Para reducir este tiempo, se ha paralelizado la el proceso de *matching*.

Para ello, se ha creado un vector global *imagenesEntrenamiento* en el que se almacenan, con la estructura de un diccionario, las imágenes de la base de datos. Se han dividido las imágenes de la base de datos entre los procesadores del servidor en el que se esté ejecutando el reconocimiento, de forma que, si hay ocho procesadores (como es el caso del servidor de la empresa *Rubycon-IT*), cada procesador comparará la imagen consulta (la fotografía enviada desde el cliente móvil) con una octava parte del conjunto de imágenes de la base de datos. Cuando van terminando los hilos, van añadiendo al vector global de diccionarios, una nueva clave en cada posición donde se guardan las coincidencias entre la imagen de dicha posición del vector y la imagen consulta recibida en el servidor. De este modo, cuando todos los procesadores terminan de ejecutar sus tareas, obtenemos un vector global en el que cada posición del vector es un diccionario que contiene tanto la imagen de entrenamiento, como las coincidencias de dicha imagen con la imagen consulta. Posteriormente, este vector se ordena según el número de coincidencias. Si existe algún elemento del vector que supera el porcentaje de coincidencias establecido, se devuelve dicha imagen. En caso de haber varias, se devuelve la imagen cuyo porcentaje de coincidencias sea mayor.

El código, paralelizado de esta forma, funciona correctamente para una sola petición simultánea. Si se realizan varias peticiones simultáneas, puesto que todos los hilos modifican el mismo vector global, el resultado devuelto es totalmente impredecible. Se ha realizado de esta forma por cuestión de tiempo. No obstante, la opción de realizar la paralelización para que el servidor pueda aceptar diferentes peticiones de forma simultánea, se plantea en el apartado de trabajo futuro.

Capítulo 8

Pruebas sobre los algoritmos de reconocimiento de imágenes

En este proyecto se implementan 3 algoritmos detectores de características de la imagen (*SURF*, *SIFT* y *ORB*) y dos algoritmos comparadores de descriptores (*FLANN* y *BFMatcher*). No obstante, la decisión de utilizar uno de los descriptores y no el otro se determina a la hora de decantarse por uno de los tres detectores de características, puesto que *FLANN* funciona con *SURF* y *SIFT*, mientas que *BFMatcher* lo hace con *ORB*.

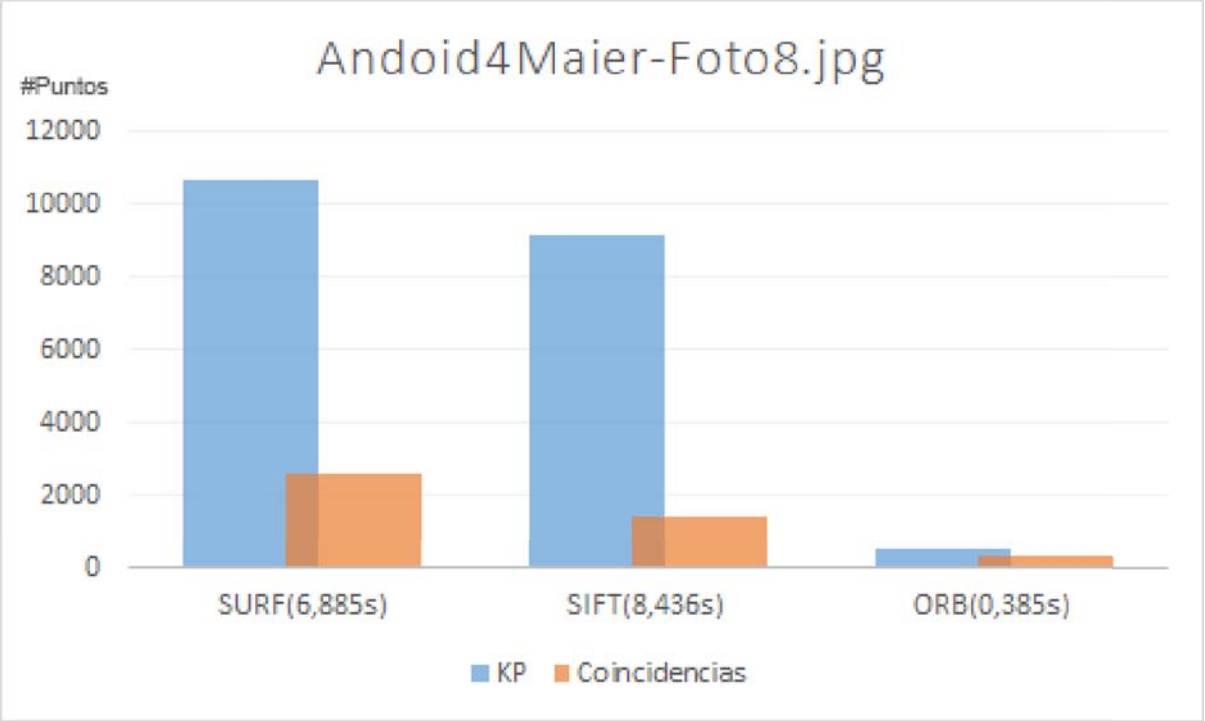
No obstante, es necesario hacer varias pruebas utilizando todos los algoritmos para saber cuál funciona mejor (aplicado al tipo de imágenes que se van a utilizar en la base de datos de esta aplicación). Pero, no sólo eso, si no que, además, los algoritmos tienen diversos parámetros. También es necesario experimentar variando estos parámetros para conocer de forma empírica, como afectan estos valores a la efectividad del reconocimiento. Para ello, se llevaron a cabo una serie de pruebas, que se detallan a continuación.

Intentando abarcar un espectro más amplio de los tipos de imágenes que se van a reconocer en la aplicación, se han realizado las pruebas sobre dos imágenes. Una con un número elevado de descriptores, es decir, con muchos detalles y texturas (*Android4Maier*), y otra que es un logotipo (*Por1cafealdía*), la cual cosa significa que los descriptores que contiene son mucho menores, por no contener prácticamente detalles que generen descriptores. Además, para acentuar el número de descriptores obtenidos, se ha realizado la fotografía de la imagen de *Android4Maier* con una cámara de 8 megapíxeles, mientras que la fotografía de la tarjeta de *Por1cafealdía* se ha realizado con una cámara de 3 megapíxeles. De esta forma, se cubre un espectro amplio de imágenes.

8.1. Análisis preliminar

El objetivo de la primera prueba que se realiza es conocer, a grandes rasgos, la efectividad de cada algoritmo detector. Estos son los *KP (Key Points)* y las coincidencias obtenidas entre fotografías de las dos imágenes (*Android4Maier - 8mp* y *Por1cafealdía - 3mp*) y sus homónimas

en la base de datos, utilizando los parámetros por defecto para cada algoritmo.



KP (Key Points)

Figura 8.1: Prueba preliminar de algoritmos detectores para la imagen *Android4Maier*.

Como se muestra en el gráfico anterior, utilizando los detectores *SURF* y *SIFT* se obtiene un número mucho mayor, tanto de puntos clave como de coincidencias, que utilizando el *ORB*.

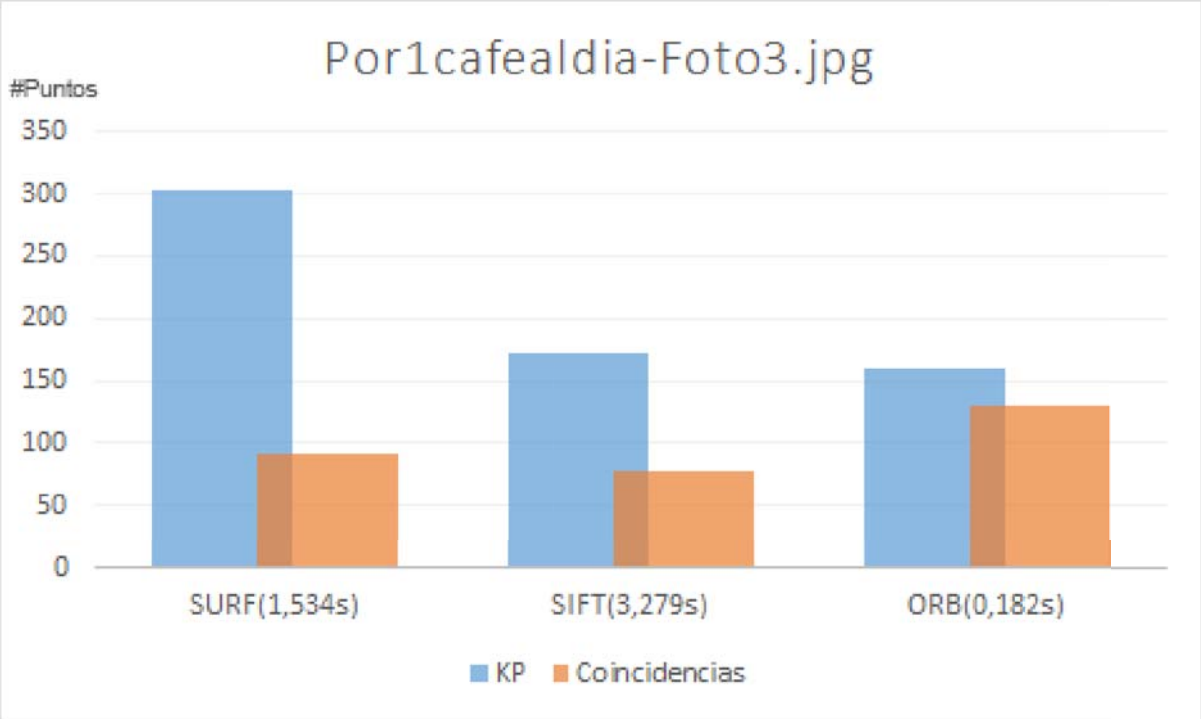


Figura 8.2: Prueba preliminar de algoritmos detectores para la imagen *Por1cafealdia*.

El esta ocasión, se obtiene un número más similar puntos clave con los tres algoritmos, aunque *ORB* ha encontrado un mayor número de coincidencias.

Conclusiones: Aunque el algoritmo más rápido es el *ORB*, la cantidad de coincidencias que obtiene es mucho menor, para la imagen con muchos descriptores, que el obtenido por los otros descriptores. No obstante, para la imagen logotipo, aunque sigue obteniendo menos descriptores, el porcentaje de coincidencias es mayor.

8.2. Análisis *SURF*

En esta prueba se busca conocer el porcentaje de coincidencias obtenido al comparar cada una de las dos imágenes (*Android4Maier - 8mp* y *Por1cafealdía - 3mp*) con todas las imágenes de la base de datos. Estas imágenes, en los resultados, se han nombrado por un título, acompañado del sufijo BaseX.jpg, donde X son los megapíxeles de la cámara con la que se ha capturado la imagen.

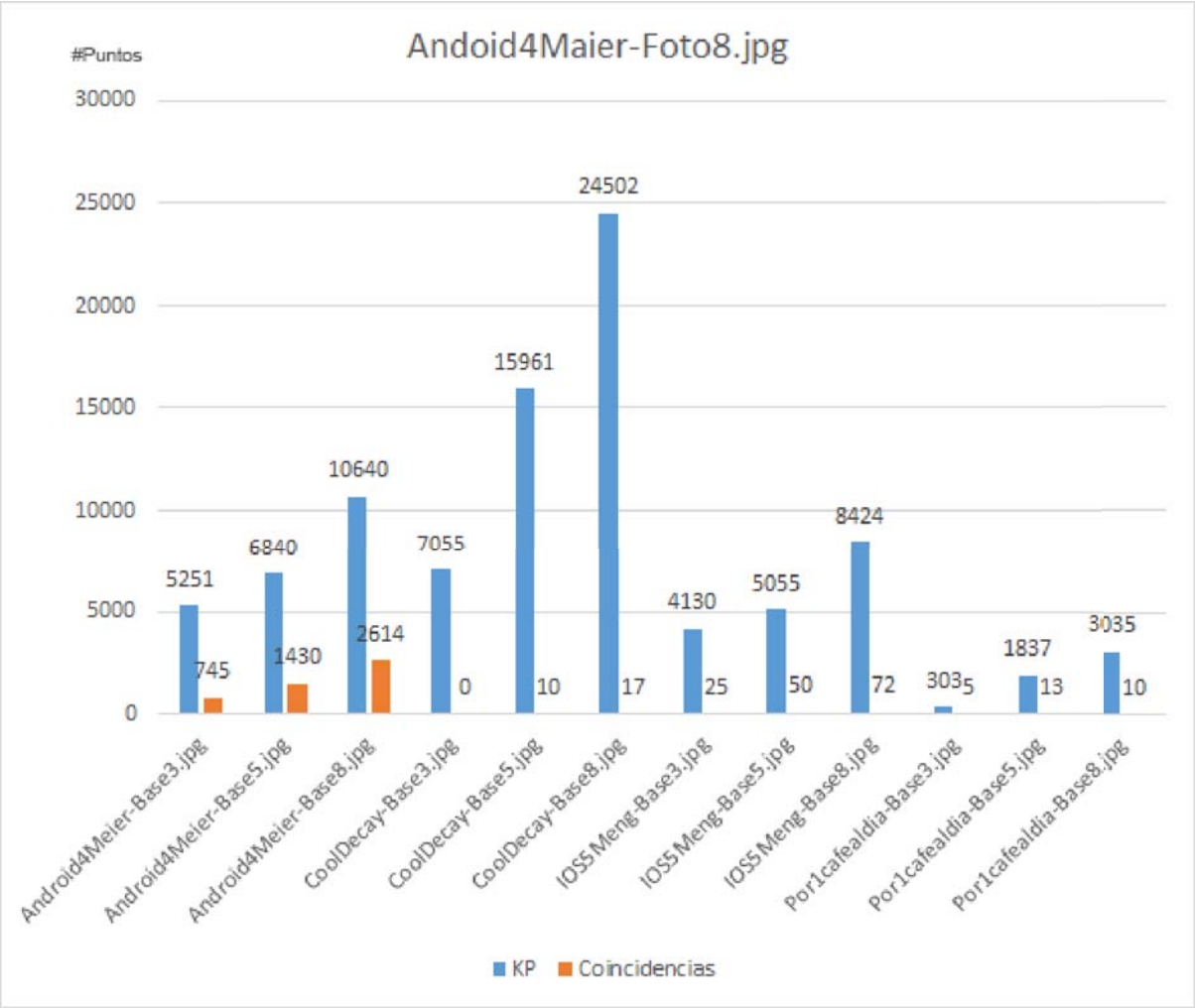


Figura 8.3: Descriptores y coincidencias obtenidas para la imagen *Android4Maier* comparada con las imágenes de la base de datos de pruebas.

En el gráfico anterior, se muestran los puntos clave obtenidos para cada imagen de la base de datos (en azul) y el número de coincidencias entre estos *Key Points* y los de la imagen de *Android4Maier*. Como se aprecia, el número de coincidencias entre esta imagen y sus homónimas en la base de datos (con varias resoluciones) es mucho mayor que el número de coincidencias entre la imagen y otras que no son ella misma en la base de datos.

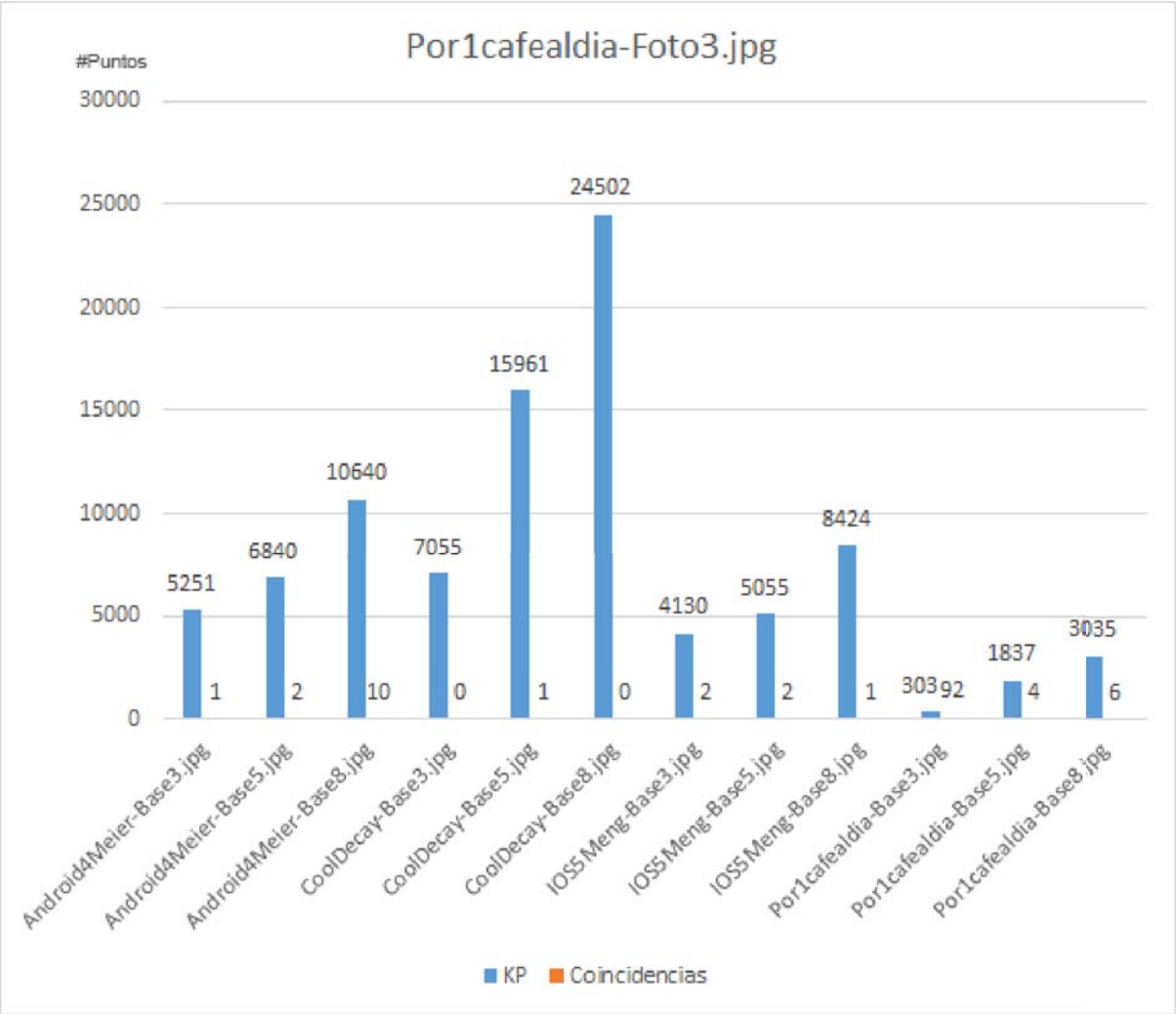


Figura 8.4: Descriptores y coincidencias obtenidas para la imagen *Por1cafealdia* comparada con las imágenes de la base de datos de pruebas.

En este caso, al tener la imagen a reconocer un número menor de descriptores, el número de coincidencias también es menor. Pero, sigue siendo considerablemente mayor que el número de coincidencias entre la imagen *Por1cafealdia* y las demás (92 contra menos de 11).

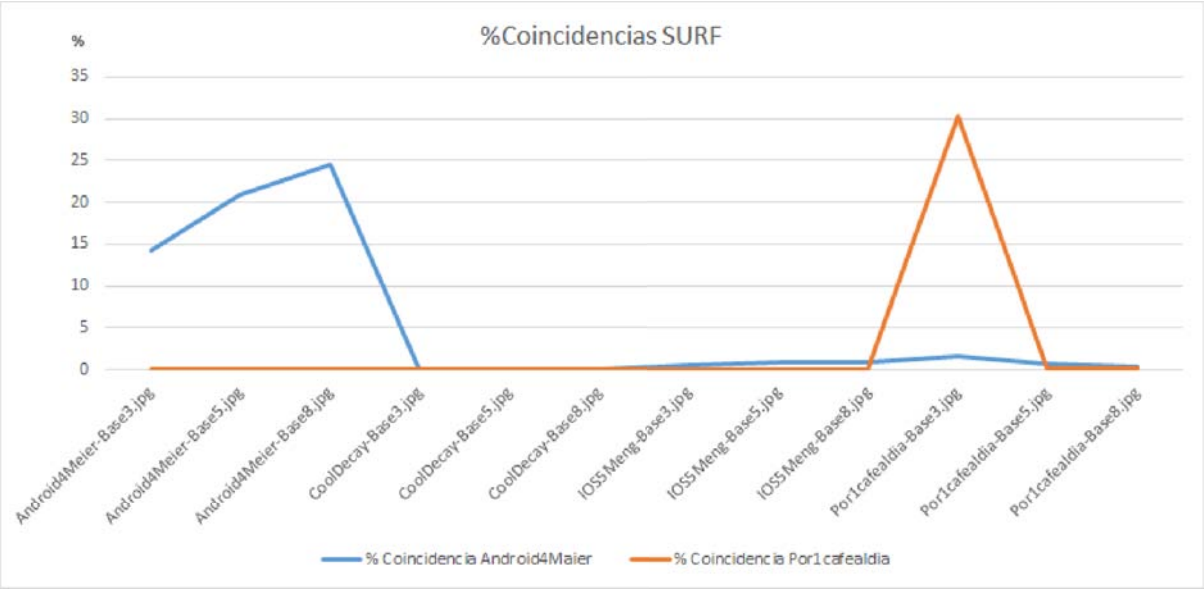


Figura 8.5: Porcentaje de coincidencias entre las imágenes *Android4Maier - 8mp* y *Por1cafealdía - 3mp*.

Por último, en la Figura 8.5 se muestran los porcentajes de coincidencias entre la imagen de *Android4Maier* (en azul) y de *Por1cafealdia* (en naranja), con respecto a todas las de la base de datos. Se aprecia como el porcentaje de coincidencias aumenta al comparar cada imagen con su homónima en la base de datos.

Conclusiones: Observando la Figura 8.5, concluimos que el algoritmo *SURF*, aún con los parámetros por defecto, es un algoritmo fiable para diferenciar imágenes, ya que obtiene un porcentaje de coincidencias mucho menor con las imágenes que no están en la fotografía, que con las que sí.

8.3. Análisis *SIFT*

Esta prueba es la misma que la anterior, pero modificando el algoritmo utilizado para obtener los descriptores de las imágenes, que en este caso es *SIFT*. Los resultados obtenido son los siguientes:

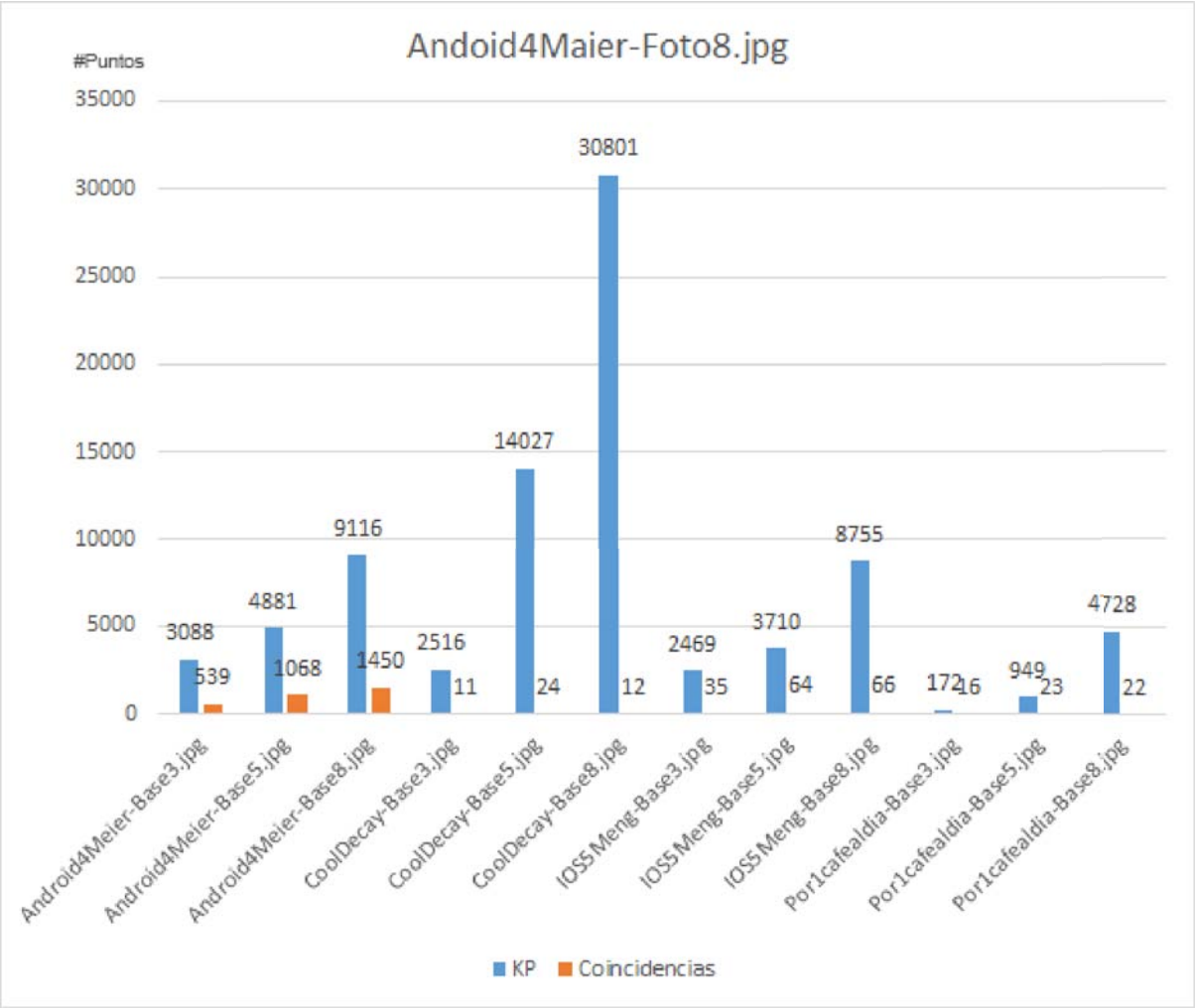


Figura 8.6: Descriptores y coincidencias obtenidas para la imagen *Android4Maier* comparada con las imágenes de la base de datos de pruebas.

De la misma forma que ocurre con el comparador *SURF*, utilizando *SIFT*, también se aprecia una diferencia notable entre las coincidencias obtenidas al comparar la imagen *Android4Maier* con ella misma en la base de datos, que al compararla con las demás.

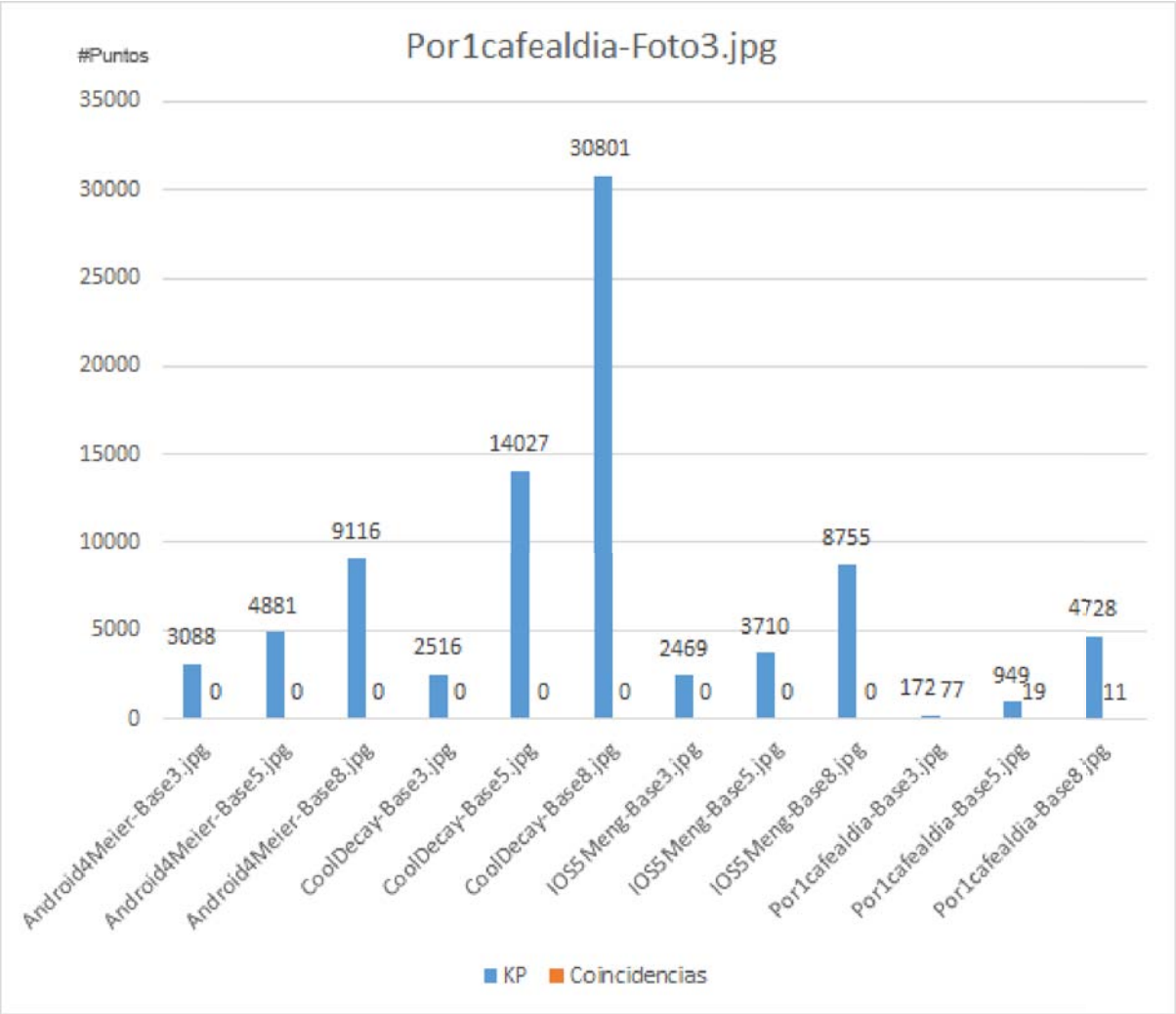


Figura 8.7: Descriptores y coincidencias obtenidas para la imagen *Por1cafealdia* comparada con las imágenes de la base de datos de pruebas.

También, al comparar la imagen de *Por1cafealdia* con ella misma en la base de datos se obtiene un número de descriptores mayor que al compararla con las demás (77 contra 0).

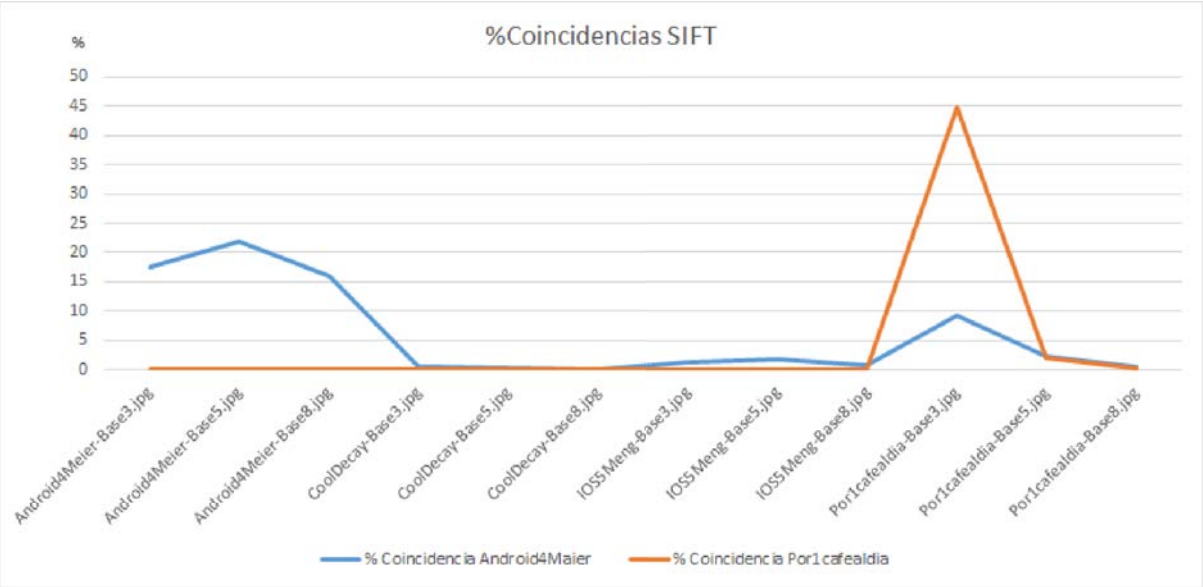


Figura 8.8: Porcentaje de coincidencias entre las imágenes *Android4Maier - 8mp* y *Por1cafealdía - 3mp*.

Se aprecia como el porcentaje de coincidencias aumenta cuando se compara la imagen con su correspondiente en la base de datos, disminuyendo mucho al compararla con otras.

Conclusiones: Los resultados obtenidos y las conclusiones sobre este algoritmo también son similares a los de *SURF*. Este también es, incluso con los parámetros por defecto, un algoritmo válido para diferenciar imágenes.

8.4. Análisis *ORB*

Se vuelve a repetir la prueba, pero en este caso para el algoritmo detector *ORB*, con el que se extraen los siguientes resultados.

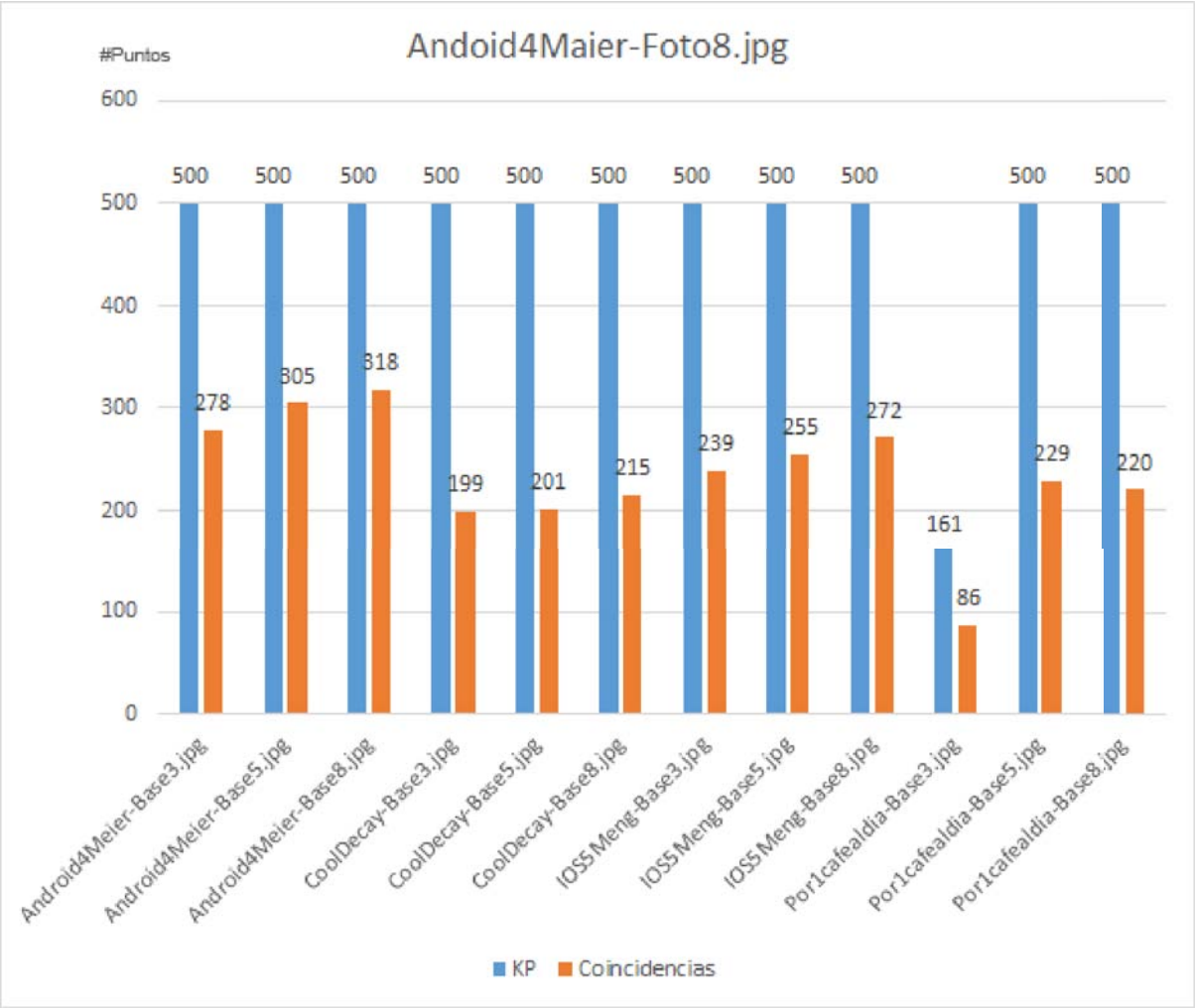


Figura 8.9: Descriptores y coincidencias obtenidas para la imagen *Android4Maier* comparada con las imágenes de la base de datos de pruebas.

Utilizando el algoritmo *ORB*, se obtiene un número muy elevado de coincidencias al comparar la misma imagen con cualquiera de las contenidas en la base de datos. Esto es un resultado muy negativo, puesto que indica que este algoritmo no es útil para diferenciar unas imágenes de otras. Además, se observa como los *Key Points* (puntos clave) no superan en ningún caso los 500, cosa que nos indica que los parámetros por defecto de este algoritmo definen un máximo de puntos clave a extraer.

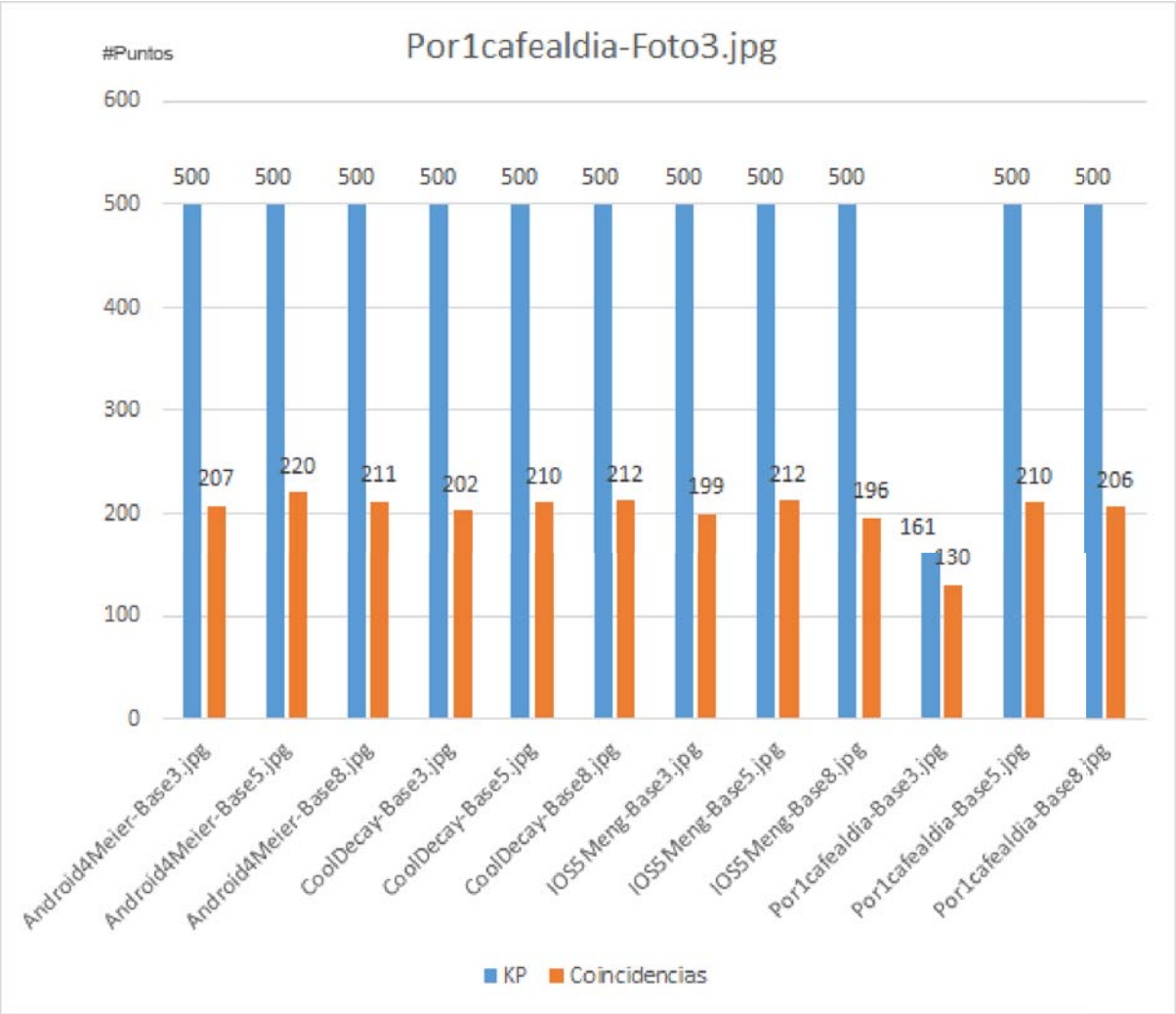


Figura 8.10: Descriptores y coincidencias obtenidas para la imagen *Por1cafealdia* comparada con las imágenes de la base de datos de pruebas.

Otra vez, con este algoritmo, volvemos a obtener un número de coincidencias que varía muy poco o nada al comparar la imagen de *Por1cafealdia* con cualquiera de las imágenes de la base de datos.

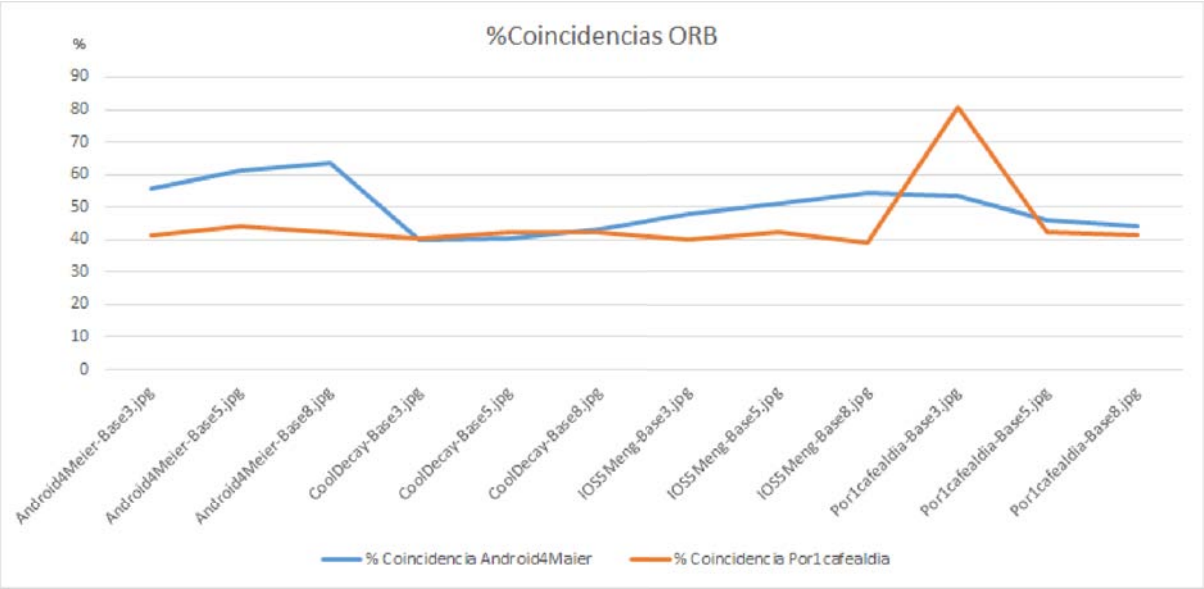


Figura 8.11: Porcentaje de coincidencias entre las imágenes *Android4Maier - 8mp* y *Por1cafealdía - 3mp*.

En consecuencia de las anteriores gráficas, el porcentaje de coincidencias que obtenemos no es suficiente para determinar a qué imagen de la base de datos se corresponde la imagen comparada.

Conclusiones: De estas pruebas, se concluye que los parámetros por defecto de *ORB* no son adecuados para discernir si dos imágenes contienen una imagen común o no imágenes, ya que todos las imágenes de la base de datos han sacado al menos un 40 % de coincidencias, independientemente de con qué imagen se esté comparando. Faltará determinar, más adelante, en el apartado 8.7, si modificando los parámetros del algoritmo, los resultados son más adecuados.

8.5. Parámetros *SURF*

En esta sección se realizan pruebas para determinar los parámetros con los que se obtienen mejores resultados. Para el algoritmo *SURF*, los parámetros que se pueden modificar son:

- **hessianThreshold**: umbral para la detección de puntos clave. Se retienen los puntos clave cuyo valor *hessian* es mayor que el valor establecido. Por lo tanto, cuanto mayor sea el valor, menos puntos clave se obtendrán. Para este proyecto, sería óptimo un algoritmo que extrajese el menor número de puntos posibles, pero que sirvieran para diferenciar en un mayor porcentaje una imagen de otra.
- **nOctaves**: es el número de pirámides *gaussianas* que el algoritmo utiliza. Para obtener rasgos más grandes, se debe aumentar el valor de este parámetro.

- **nOctaveLayers**: número de imágenes dentro de cada octava de una pirámide *gaussiana*.

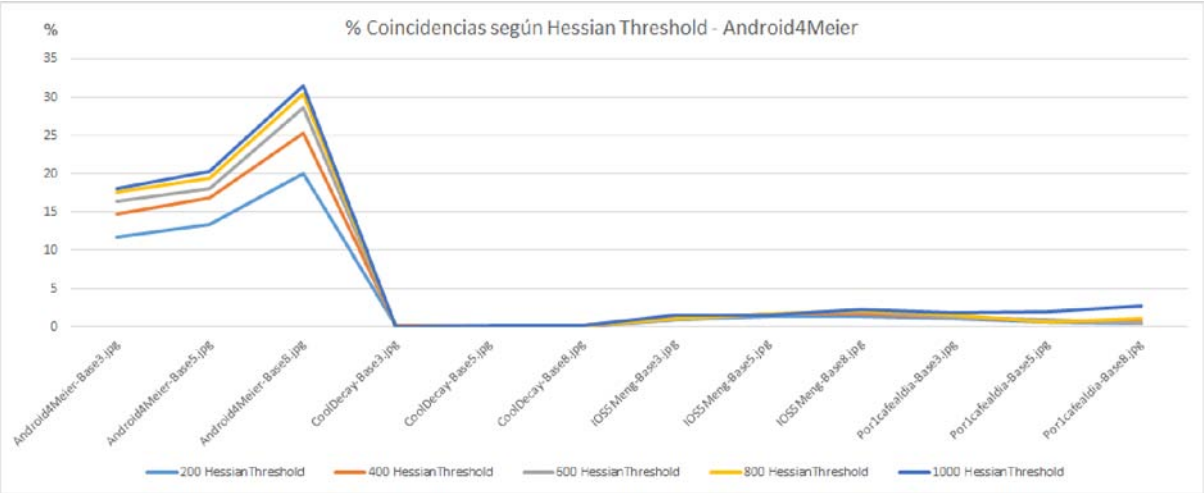


Figura 8.12: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **hessianThreshold** de *SURF*.

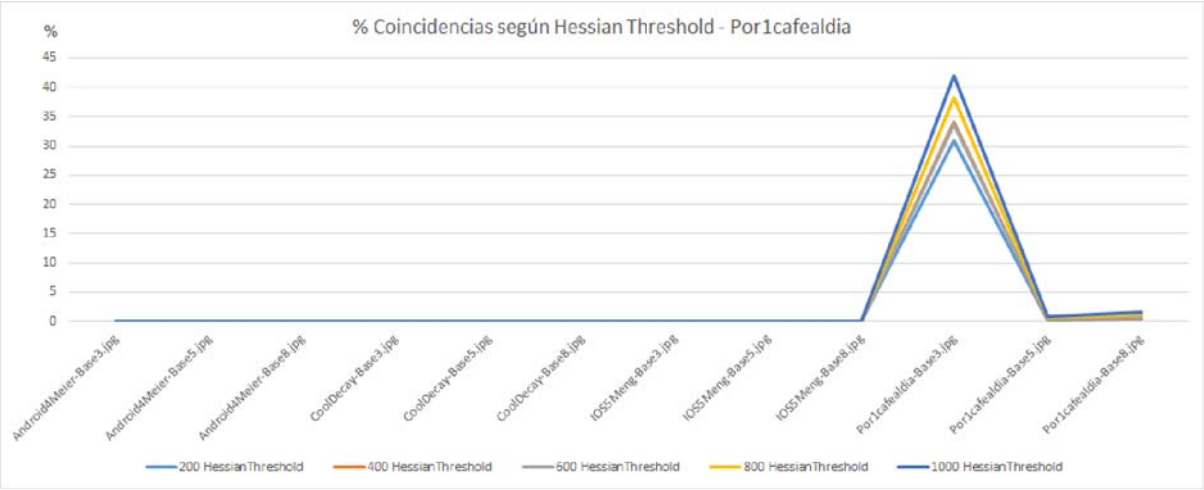


Figura 8.13: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **hessianThreshold** de *SURF*.

hessianThreshold: Por los resultados obtenidos, el valor de este parámetro es más beneficioso para el reconocimiento cuanto más alto es.

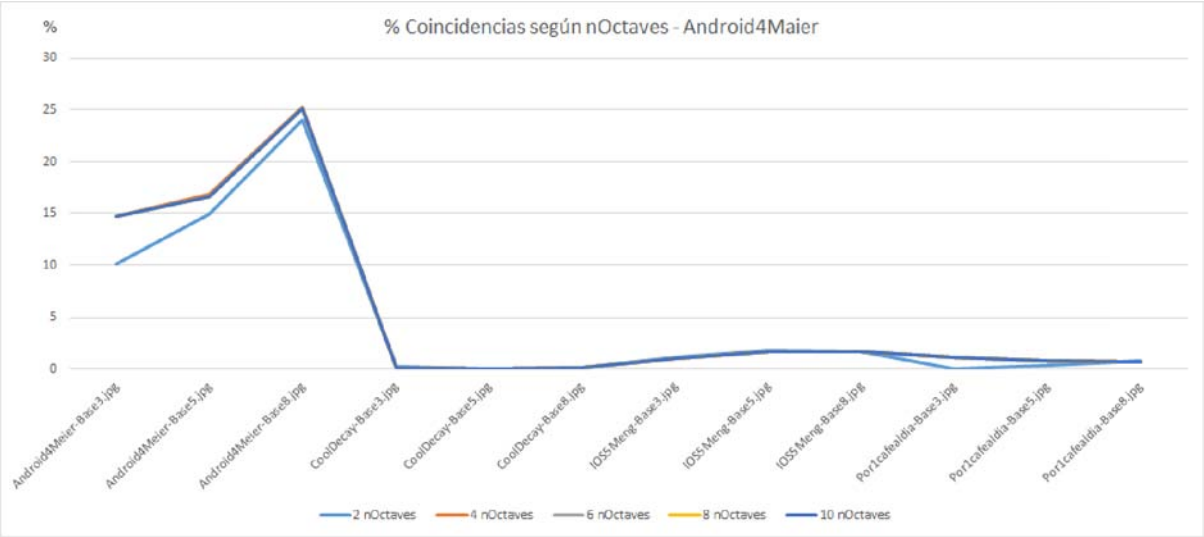


Figura 8.14: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **nOctaves** de *SURF*.

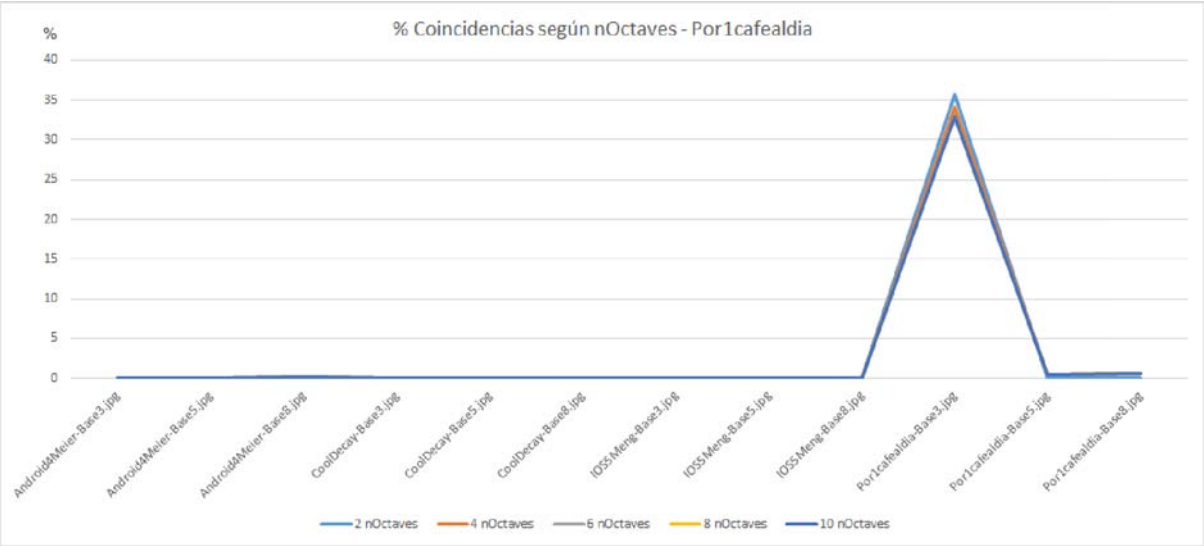


Figura 8.15: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **nOctaves** de *SURF*.

nOctaves: Con el valor 2 para el parámetro *nOctaves*, obtenemos una variación mayor del porcentaje de coincidencias obtenidas.

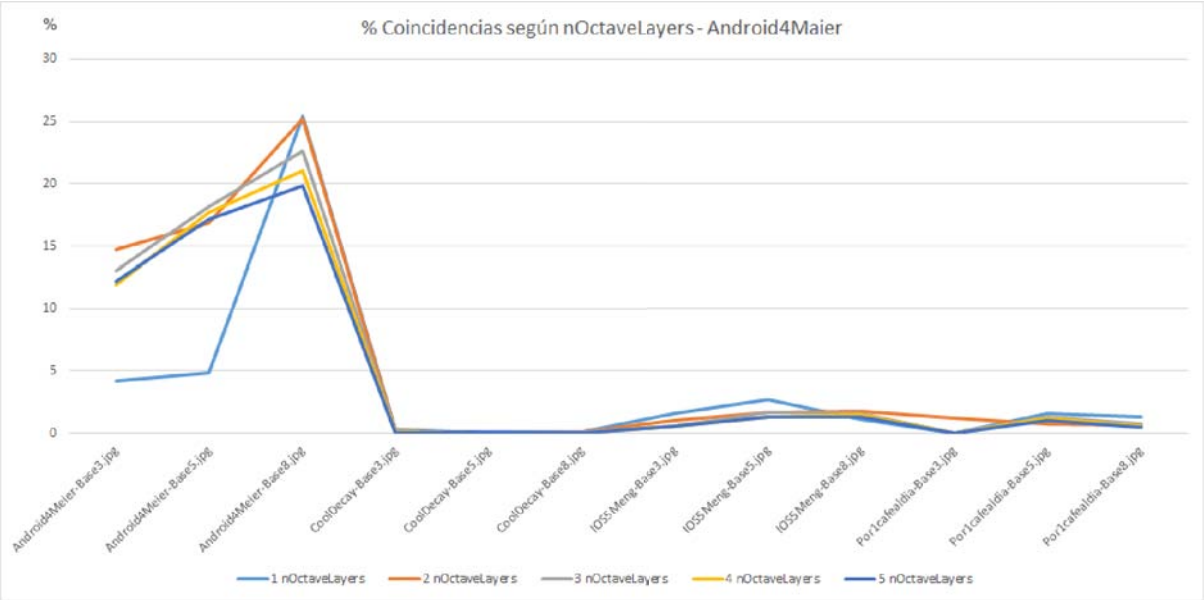


Figura 8.16: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **nOctaveLayers** de *SURF*.

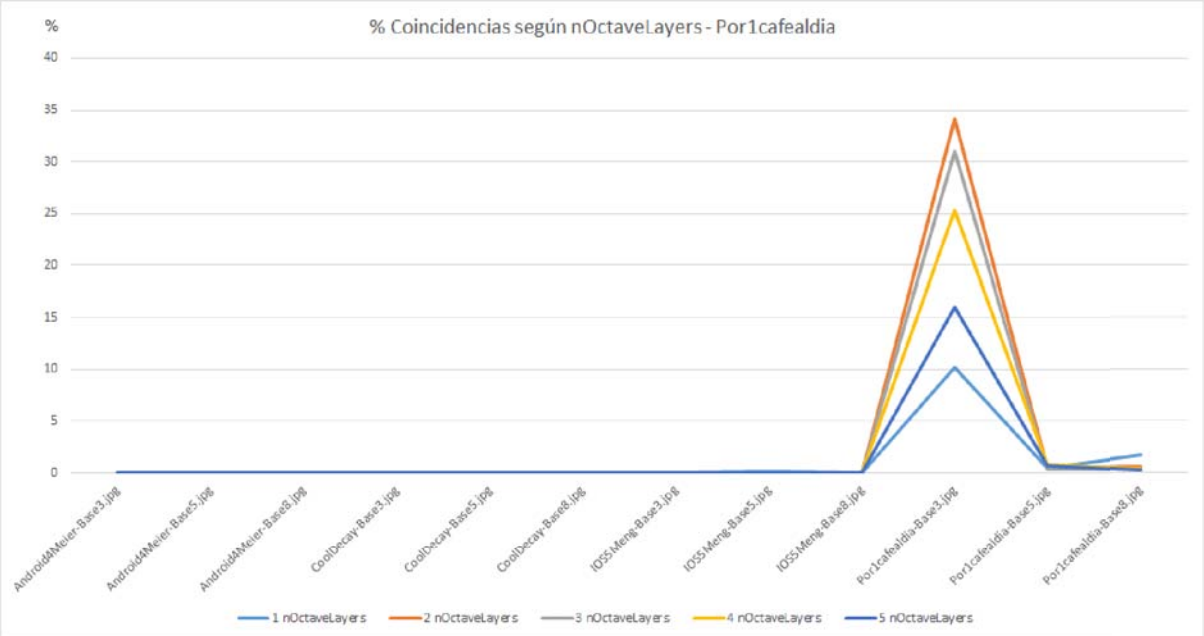


Figura 8.17: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **nOctaveLayers** de *SURF*.

nOctaveLayers: Aunque no es un parámetro muy determinante para la imagen con un mayor número de descriptores, sí lo es para la imagen del logotipo, que obtiene unos resultados mucho mejores con el valor 2 para *nOctaveLayers*.

8.6. Parámetros *SIFT*

En esta sección se realizan pruebas para determinar los parámetros con los que se obtienen mejores resultados utilizando *SIFT* como algoritmo detector de características. Para este algoritmo, los parámetros a modificar son:

- **nOctaveLayers** : número de capas en cada octava.

- **contrastThreshold** : umbral de contraste para filtrar las características en las regiones de bajo contraste. Cuanto mayor sea el umbral, menos características se obtienen.

- **edgeThreshold** : umbral utilizado para filtrar las características de borde. En este caso, a mayor valor de este parámetro, más características son obtenidas por el algoritmo.

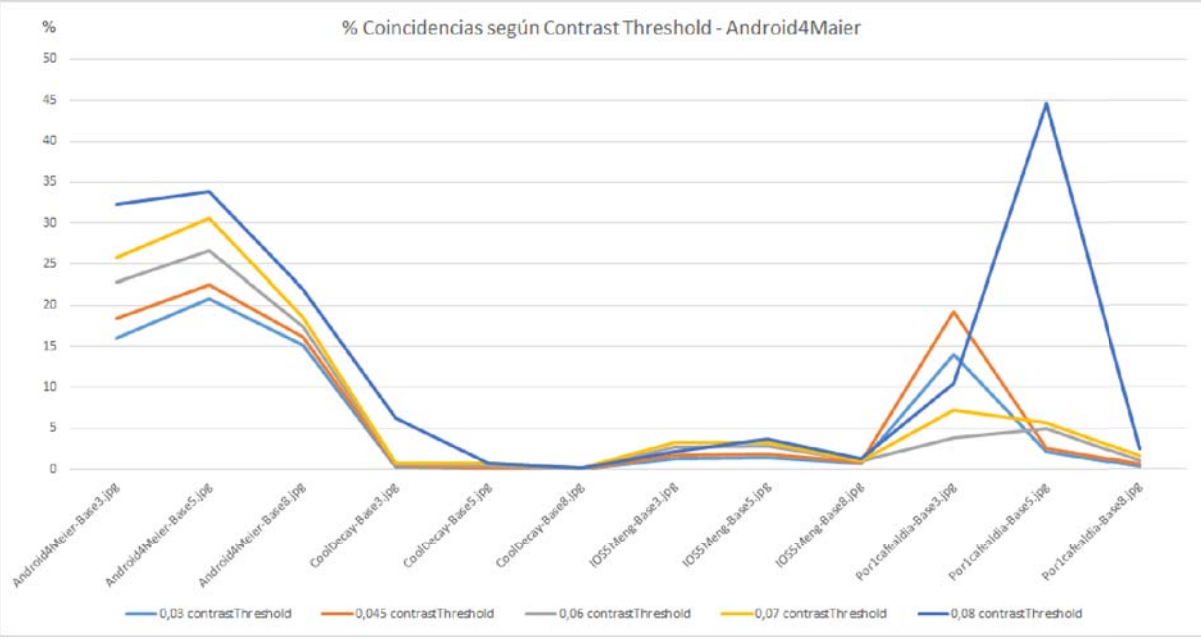


Figura 8.18: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **contrastThreshold** de *SIFT*.

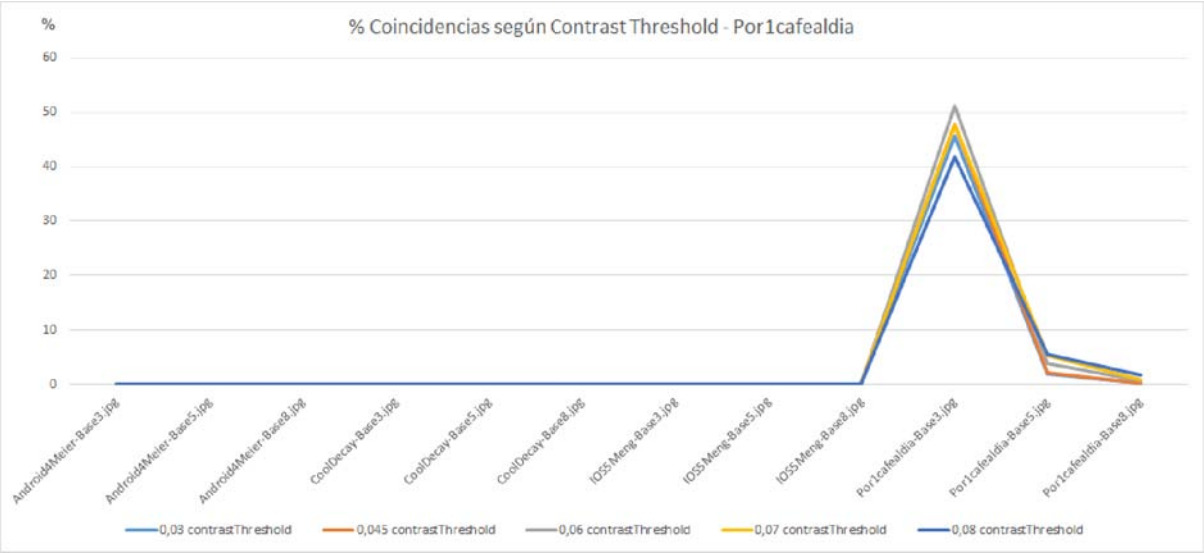


Figura 8.19: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **contrastThreshold** de *SIFT*.

contrastThreshold: Teniendo en cuenta los dos gráficos, el valor con el que mejores resultados se obtienen es con *0.06*, para este parámetro. De hecho, se observa como, para otros valores, los resultados son totalmente contradictorios.

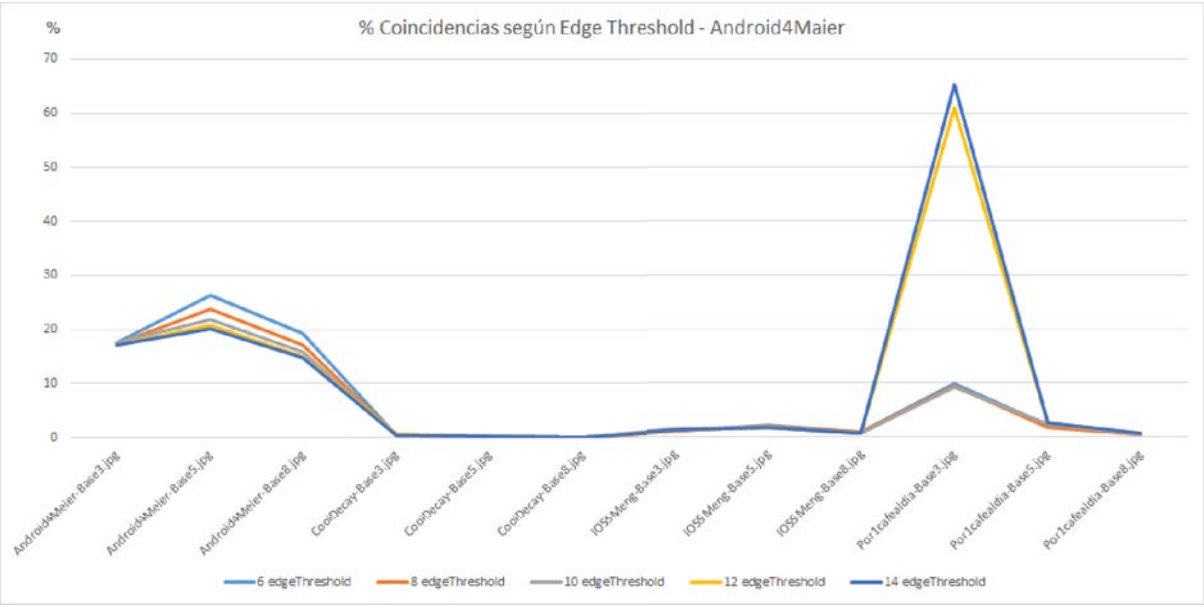


Figura 8.20: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **edgeThreshold** de *SIFT*.

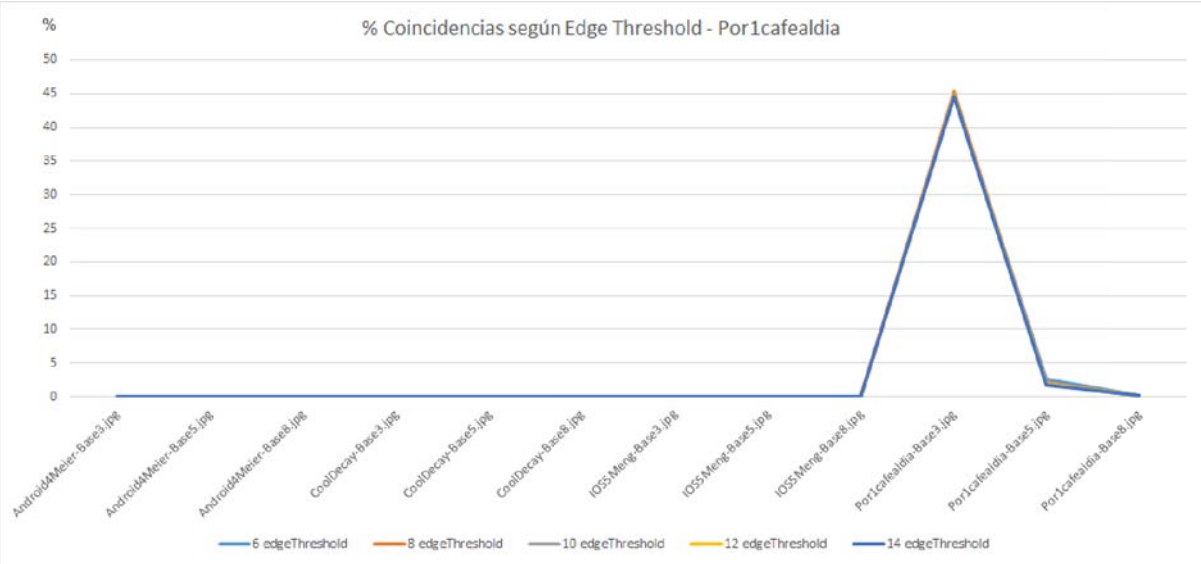


Figura 8.21: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **edgeThreshold** de *SIFT*.

edgeThreshold: Se observa como los resultados empeoran al aumentar el valor de *edgeThreshold*, hasta el punto de ser fatales cuando alcanza valores de más de 10.

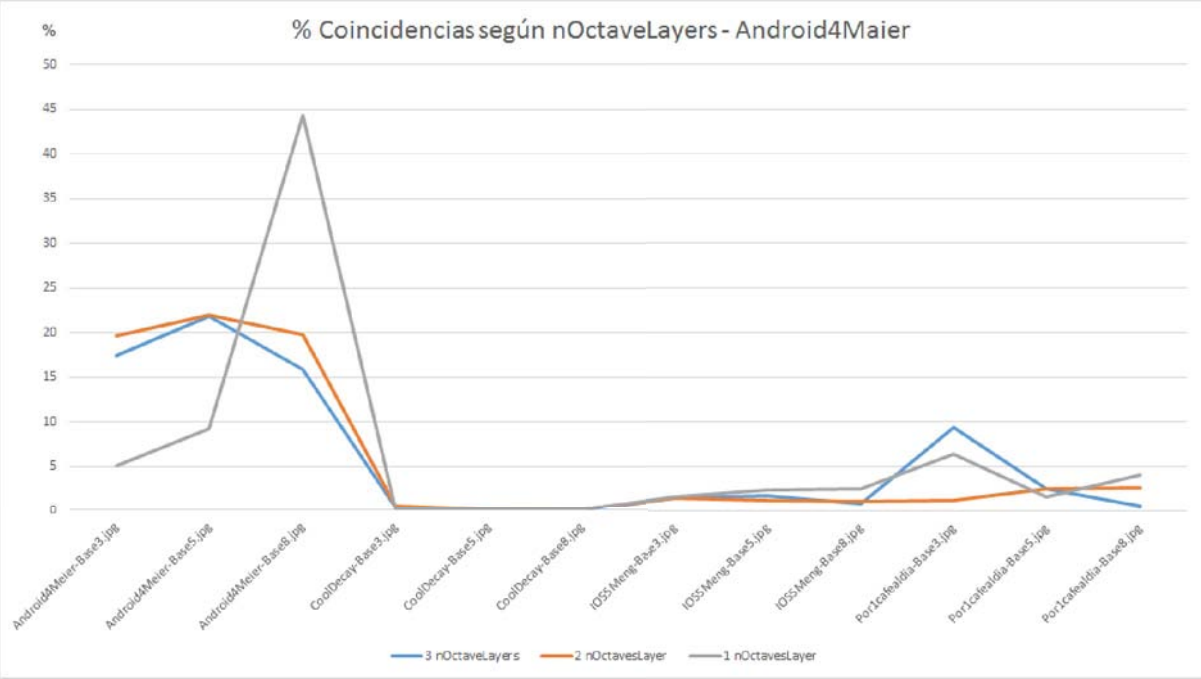


Figura 8.22: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro **nOctaveLayers** de *SIFT*.

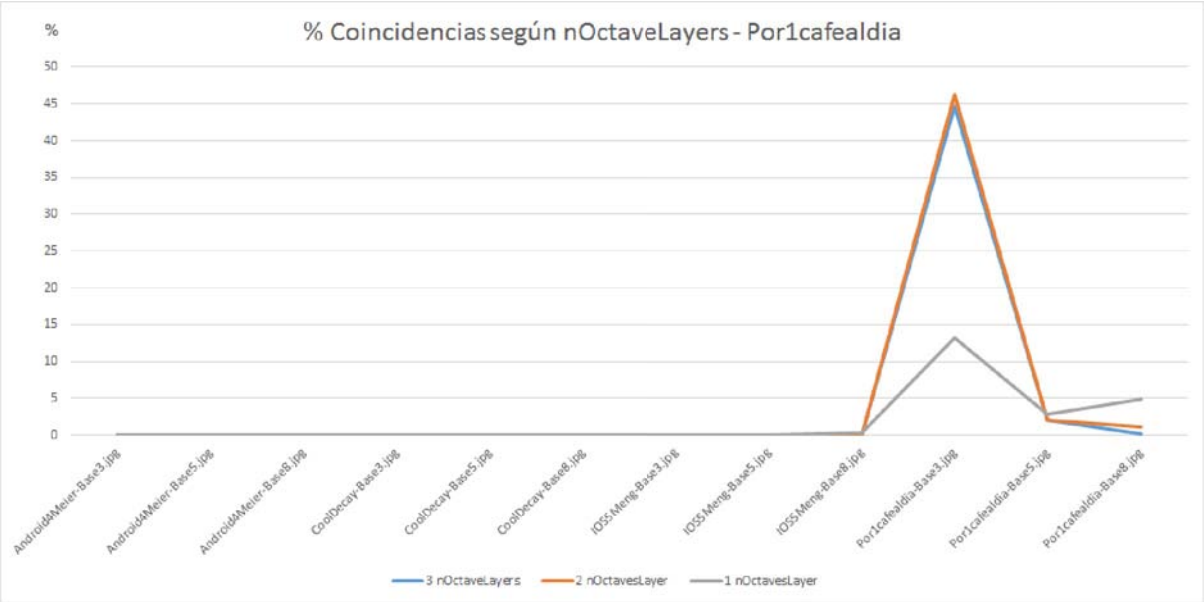


Figura 8.23: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro **nOctaveLayers** de *SIFT*.

nOctaveLayers: Eliminamos el valor *1* para este parámetro por no funcionar bien con la imagen del logotipo, y el de *3* por la misma razón, pero con la imagen de más descriptores.

8.7. Parámetros *ORB*

Para la realización de pruebas con el algoritmo *ORB*, al modificar parámetros, se obtienen, al menos en la base de datos utilizada para hacer pruebas, resultados contradictorios, en los que la variación lineal de los parámetros produce resultados con un comportamiento no explicable. Por esta razón, se ha decidido eliminar de la lista de posibles algoritmos a utilizar el detector *ORB* y el comparador que lo acompaña: *BFMatcher*. No obstante, a continuación se adjuntan los resultados obtenidos para diversos parámetros. Se puede observar como, para la imagen que obtiene un mayor número de descriptores, los resultados, independientemente de los parámetros, están entre el 100 % y el 20 %, valores totalmente insuficientes para determinar el reconocimiento de una imagen.

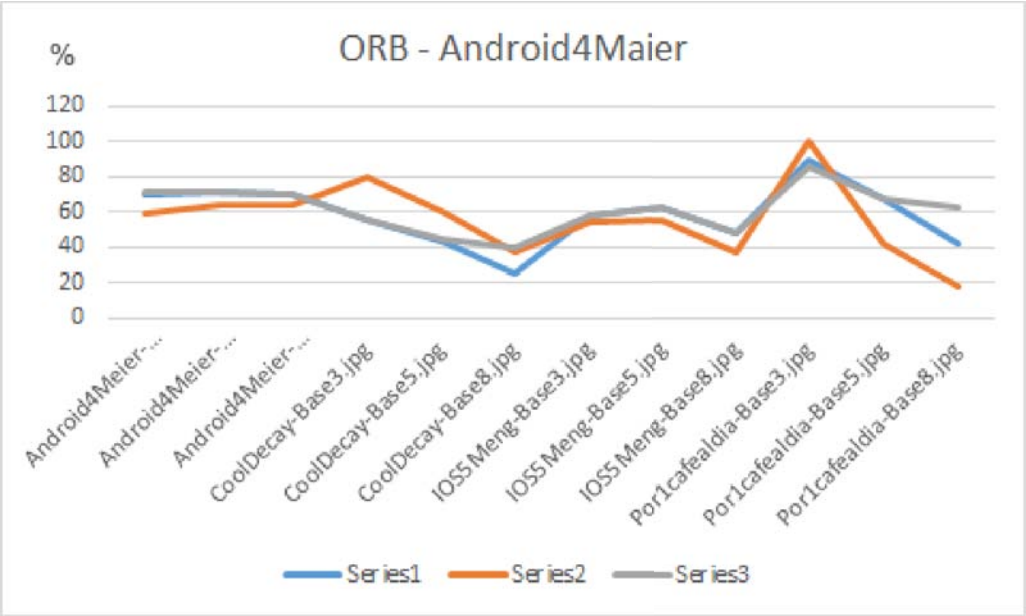


Figura 8.24: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* con el detector *ORB*.

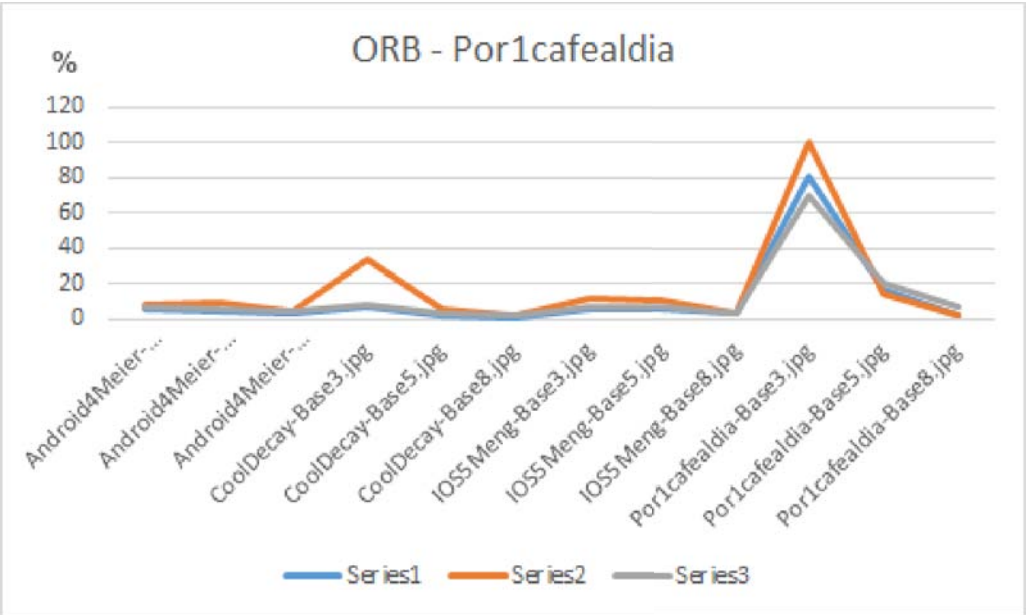


Figura 8.25: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* con el detector *ORB*.

8.8. Parámetros *FLANN*

Descartado el algoritmo *ORB* y, por consiguiente, su *matcheador BFMatcher*, solamente queda analizar qué parámetros conviene utilizar con el algoritmo comparador de descriptores

FLANN. Los parámetros que se estudian son:

- **FLANN_INDEX** : número de *kd-trees* paralelos a utilizar.

- **coeficiente_distancia** : distancia mínima a la que se encuentran dos puntos clave para determinar si se incluye dicha característica en la lista de coincidencias aceptadas.

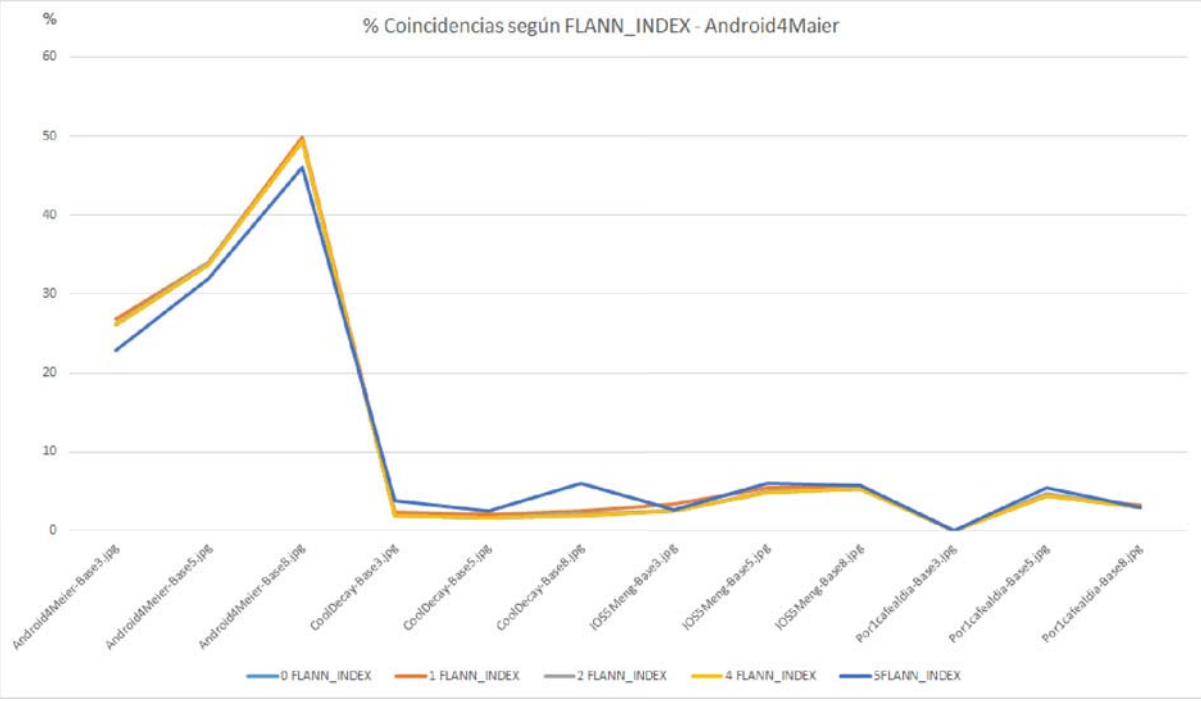


Figura 8.26: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro *FLANN_INDEX*.

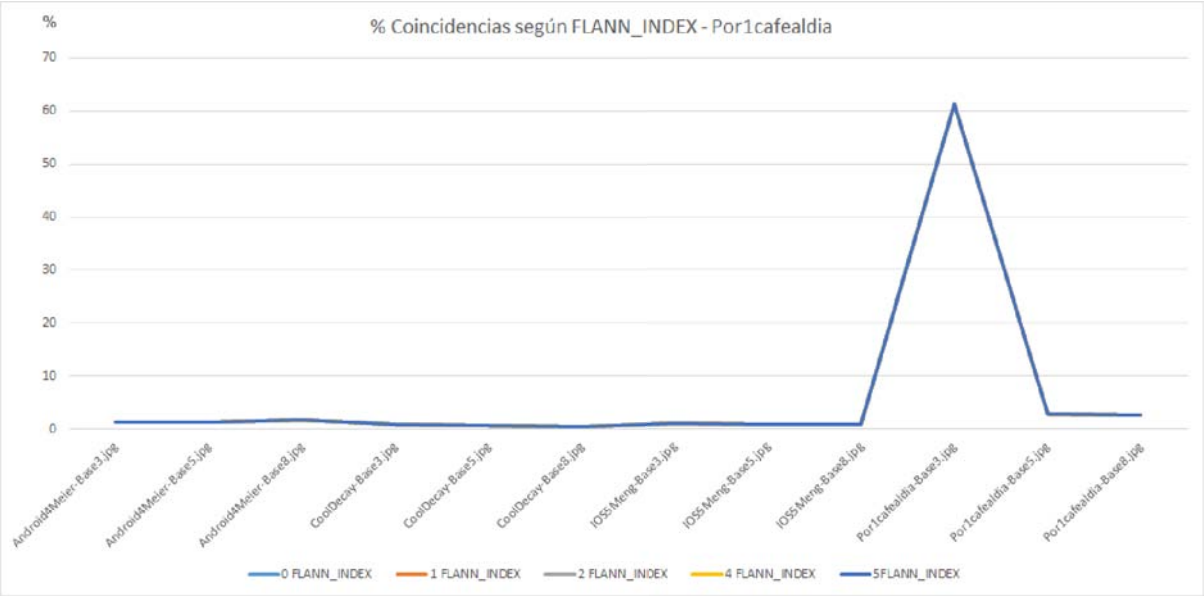


Figura 8.27: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro *FLANN_INDEX*.

FLANN_INDEX: El tiempo que tarda el algoritmo en realizar la comparación se va reduciendo cuando aumenta el valor de este parámetro. No obstante, para el valor 5, los resultados empiezan a empeorar, por lo que se decide establecer el valor del parámetro en 4.

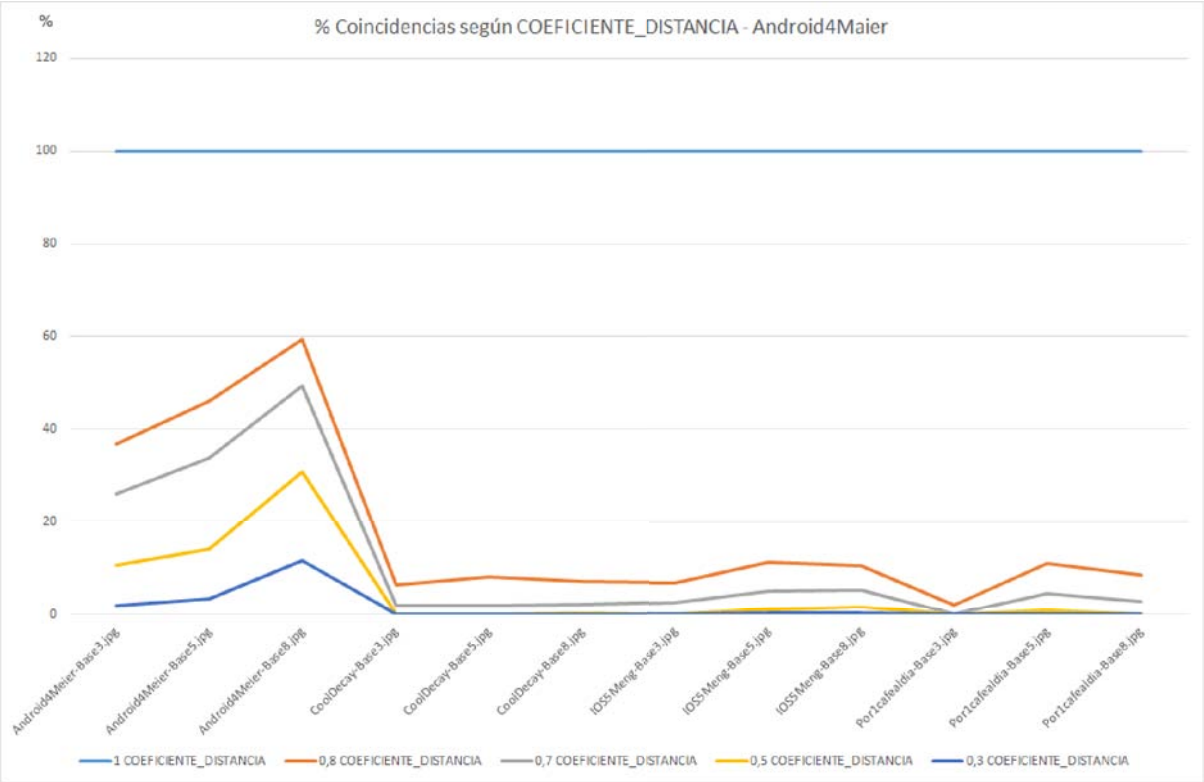


Figura 8.28: Porcentaje de coincidencias con la imagen *Android4Maier - 8mp* según varía el parámetro *coeficiente_distancia*.

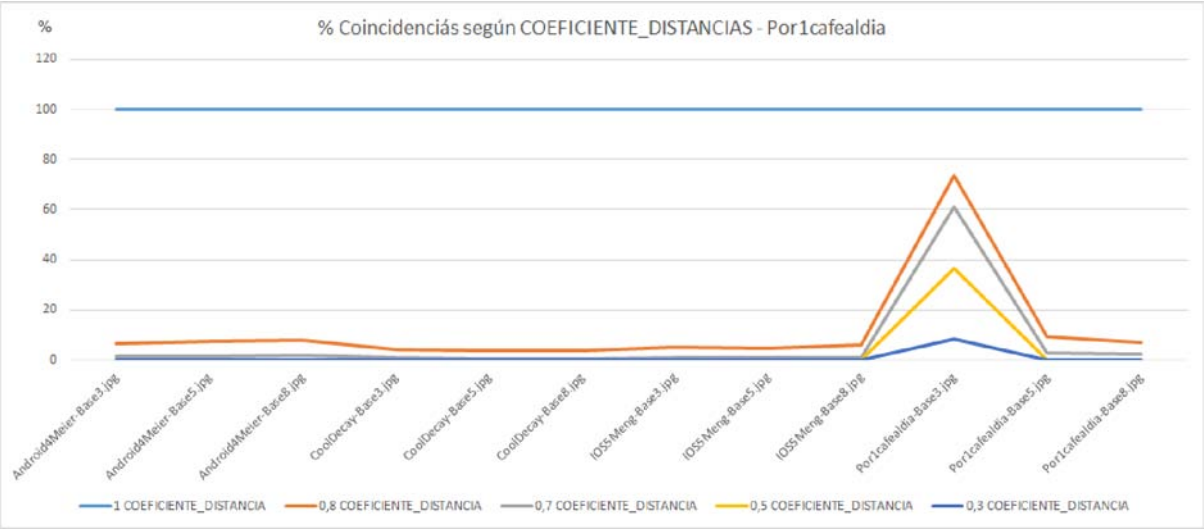


Figura 8.29: Porcentaje de coincidencias con la imagen *Por1cafealdía - 3mp* según varía el parámetro *coeficiente_distancia*.

coeficiente_distancia: Los resultados obtenidos son mejores cuanto mayor es el coeficiente de distancia. De hecho, si éste se reduce en exceso, obtenemos que todos los puntos clave coinciden.

8.9. Conclusiones de las pruebas sobre los algoritmos de reconocimiento

Después de analizar las pruebas, se concluye que los algoritmos a utilizar para el reconocimiento de imágenes son *SURF* y *FLANN*. Finalmente, se ha desestimado la utilización del *SIFT* puesto que los resultados eran ligeramente inferiores y el tiempo, superior, de modo que no resultaba beneficiosa en ningún aspecto su utilización.

A la hora de implementar el reconocimiento de imágenes, se decide utilizar el detector *SURF* con dos parámetros diferentes, tal y como se muestra en el Bloque de código 8.1. El primero, con los parámetros obtenidos mediante las pruebas. Si, tras aplicar este algoritmo, no se obtienen suficientes descriptores, se aplica *SURF* con otros parámetros, con el fin de obtener más. De este modo, se consigue una mayor efectividad para recocer imágenes que contienen menos detalles.

Bloque de código 8.1: Parámetros finales.

```
comparador = ComparadorFLANN(flannIndexKDTree = 4, coeficienteDistancia = 0.75)
detector = DetectorSURF(hessianThreshold = 1000, nOctaves = 2, nOctaveLayers = 2)
detectorAux = DetectorSURF(hessianThreshold = 500, nOctaves = 2, nOctaveLayers = 4)
reconocedor = Reconocedor(db, detector, comparador, detectorAux)
```

Capítulo 9

Resultado final

En este capítulo se muestra el resultado obtenido después de analizar, diseñar e implementar todos los aspectos que se detallan en esta memoria. En la Figura 9.1 se muestra la visión general de la aplicación, cuyas vistas se muestran y comentan en los apartados 9.1, 9.2, 9.3, 9.5, 9.6, 9.7, 9.8, 9.9 y 9.9.1.

Figura 9.1: En la Figura 9.1 se muestra un resumen de las diferentes vistas de la aplicación.

9.1. Pantalla de inicio

El estado final de la vista de la pantalla de inicio (*V1* en la Figura 9.1) es el que aparece en la Figura 9.2. Utilizando un estilo plano, sin sombras, relieves o demás características similares, se ha implementado, para esta vista, una parte superior en la que aparecen las tres pestañas disponibles (resaltando la pestaña actual). La parte central muestra un logotipo de la empresa *Rubycon-IT* y un mensaje de bienvenida, con un fondo azul y letras blancas. Por último, en la parte inferior aparece el botón que nos dirige a la pantalla de cámara.

Figura 9.2: *Screen* de la *view* de inicio.

9.2. Vista de Favoritos

Desde esta vista (*V2* en la Figura 9.1), que se muestra en la Figura 9.3, se pueden visualizar los favoritos previamente marcados como tal desde el dispositivo. Además, al seleccionar uno de ellos, se accede a la vista de resultado que lo muestra.

Figura 9.3: *Screen* de la *view* de Favoritos.

9.3. Vista de *Acerca de*

Esta vista (*V3* en la figura 9.1), consta de un botón para acceder al tutorial y de un *scroll* en el que se incluye información sobre los organismos que han hecho posible la aplicación y las licencias utilizadas.

Figura 9.4: *Screen* de la *view* de Favoritos.

9.4. Pantalla de cámara

La pantalla de cámara (*V4* en la Figura 9.1), que se muestra en la Figura 9.5, se compone, finalmente, de tres botones. De izquierda a derecha, el primero corresponde a la acción de *ir a galería*, el segundo, a *capturar fotografía* y el tercero, a *cerrar*. Además de estos botones, la vista incluye una región en la que se muestra lo que la cámara del dispositivo captura.

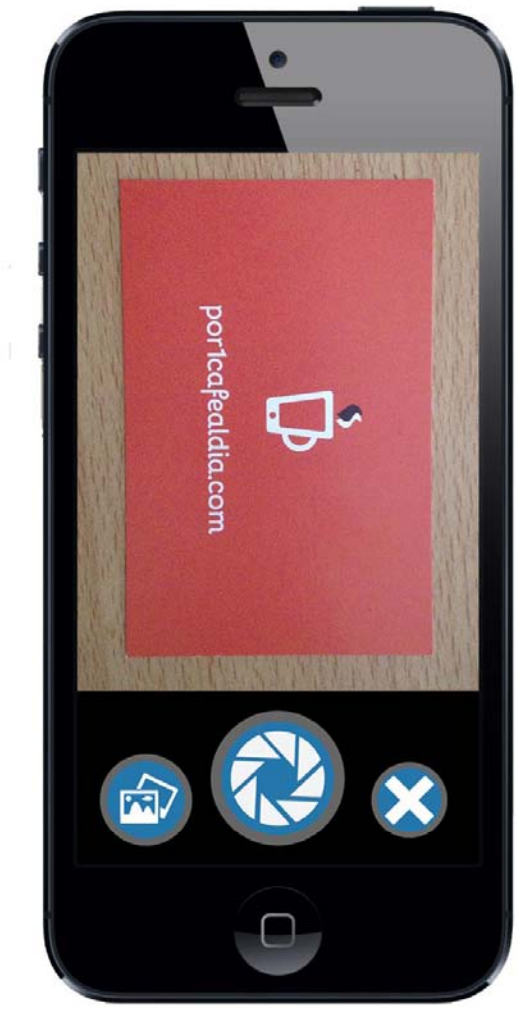


Figura 9.5: *Screen* de la *view* de cámara.

9.5. Barra superior de las pantallas de resultado

Todas las pantallas de resultado (*V9*, *V10*, *V11* y *V12* en la Figura 9.1) cuentan con una barra superior. En ella se incluye, aparte del título del resultado que se está mostrando, sea del tipo que sea, un botón para volver a la pantalla de inicio, otro para marcar el resultado como favorito y otro para compartir el resultado: Figura 9.6.

Figura 9.6: *Screen* de la barra superior.

Para los iconos de los botones, se ha utilizado una fuente llamada *Font Awesome* [13]. Esta fuente funciona muy bien para dar un acabado atractivo a los botones de las aplicaciones cuando no se dispone de un equipo de diseño que proyecte cada uno de los iconos de la aplicación. En la Figura 9.7 se muestran algunos de los iconos incluidos en esta fuente.

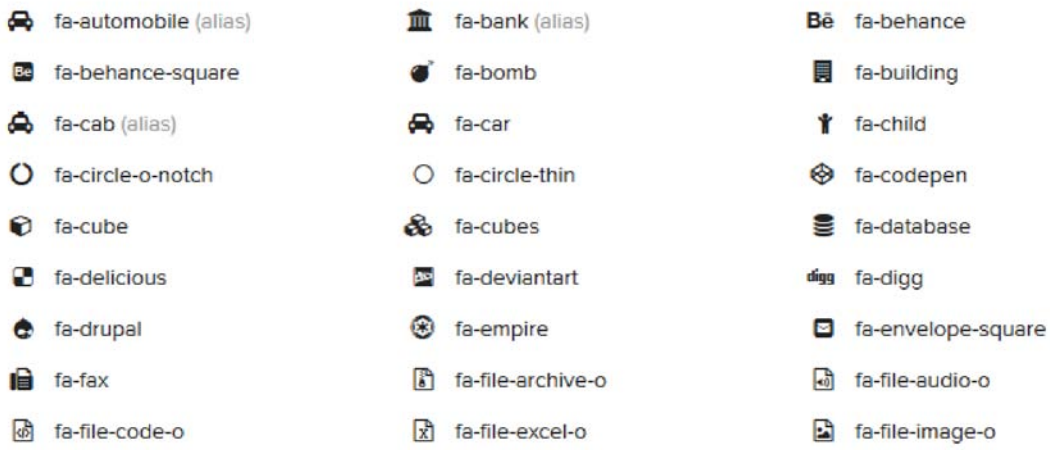


Figura 9.7: En la Figura 9.7 se muestran ejemplos de iconos incluidos en la fuente *Font Awesome*.

9.6. Resultado Web

Tal y como aparece en la Figura 9.8, esta vista (*V12* en la Figura 9.1) cuenta, únicamente, con la barra superior para poder volver atrás, compartir y marcar como favorito el resultado, y con un *visor web* en el que aparece la *web* devuelta como resultado.

Figura 9.8: *Screen* de la *view* de resultado *Web*.

9.7. Resultado Galería

A la hora de implementar la vista de galería (*V10* en la Figura 9.1), surge un problema. Se han definido una serie de gestos que afectan a la imagen que se está visualizando:

- Deslizar el dedo a izquierda o derecha: cambia la imagen que aparece en pantalla.
- Tocar y arrastras con el dedo: mover la imagen por la pantalla.
- Pinzar con los dedos en la imagen: *zoom*.

Figura 9.9: *Screen* de la *view* de resultado Galería.

El problema está en que el gesto de mover la imagen hacia la derecha o a la izquierda y el gesto de deslizar hacia la izquierda o hacia la derecha son, para el dispositivo que captura los gestos, el mismo. Por lo tanto, al captarlos simultáneamente, el dispositivo no sabe si cambiar de imagen o mover la que se está visualizando.

Para solucionar este problema, solo se permite mover la imagen cuando se ha hecho *zoom* de algún tipo sobre la imagen. No obstante, una vez se ha hecho *zoom*, si se quiere cambiar de imagen hay que volver al *zoom* original. Para ello, se ha añadido un botón en la interfaz (que no aparece en el *mockup* de la figura 6.9) de forma que esta queda tal y como aparece en la figura 9.9.

9.8. Resultado Audio

La Figura 9.10 son dos capturas de pantalla de la vista de *Resultado Audio* (*V9* en la Figura 9.1). Como se aprecia, además del texto que acompaña al audio, aparece un *slider* con el que se puede manejar el momento de la pista de audio que se quiere escuchar, un indicador con el segundo exacto de la pista en el que nos encontramos, y un botón. Este mismo botón sirve para reproducir y pausar la pista de audio. Además, su aspecto varía, de modo que aparece el botón de *play* cuando el audio está detenido, y el símbolo de *pause* cuando el audio se está reproduciendo.

Figura 9.10: *Screen* de la *view* de resultado Audio.

9.9. Resultado Vídeo

Para esta vista en concreto (*V11* en la Figura 9.1), el paso de *mockup* (sección 6.1.6) a vista real fue más complicado, en lo que a diseño de la interfaz de usuario se refiere, que las demás. Esto se debe a que, al rotar el dispositivo y pasar a pantalla completa hay que cambiar el tamaño y la disposición de todos los elementos. También se ha añadido a ciertos elementos transparencia para que la visualización de todos los elementos sea el más adecuado, así como se ha programado un retardo tras el cual el *slider* y el botón de *play/pause* desaparece, para que sólo se visualice el vídeo. Al final, la interfaz ha resultado de la siguiente forma:

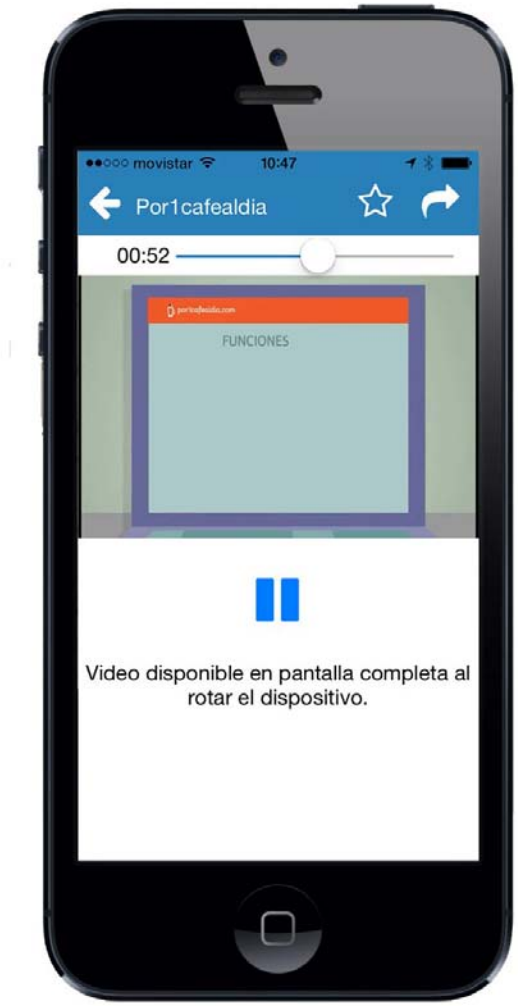


Figura 9.11: *Screen* de la *view* de resultado Audio en orientación del *iPhone* vertical.

Figura 9.12: *Screen* de la *view* de resultado Audio en orientación del *iPhone* horizontal.

9.9.1. Tutorial

Es petición del cliente el contar con un tutorial poco común, diferente de los que aparecen en la mayoría de aplicaciones. Para intentar conseguirlo, se ha aportado un toque de diseño más moderno al tutorial, y la reacción del cliente ha sido positiva.

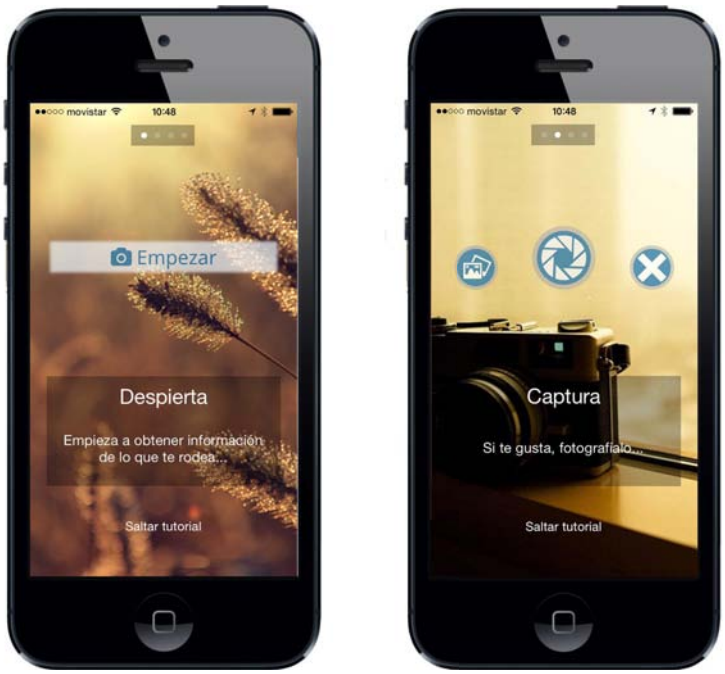


Figura 9.13: *Screens* de la pantalla 1 y 2 del tutorial.

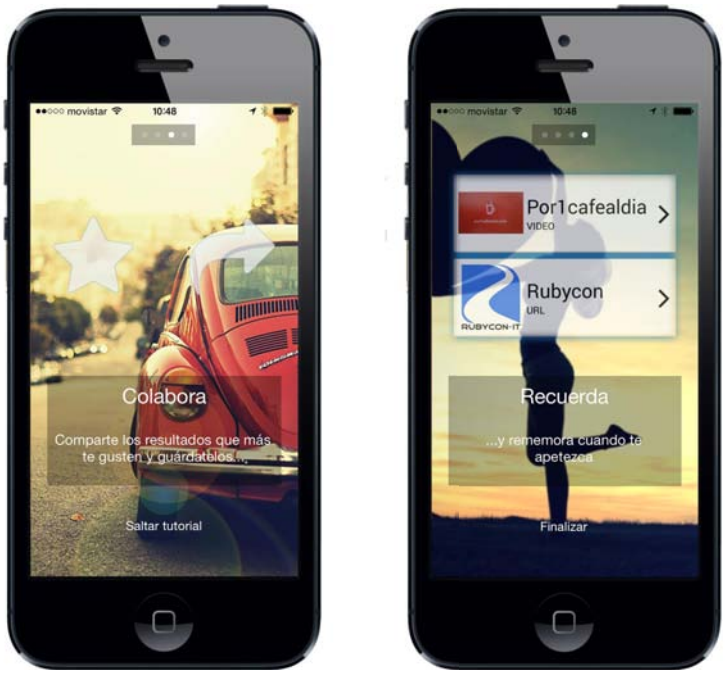


Figura 9.14: *Screens* de la pantalla 3 y 4 del tutorial.

Capítulo 10

Conclusiones

En este apartado final se describen las conclusiones del trabajo realizado, tanto a nivel técnico, como a nivel personal. Además, se añade un apartado en el que figuran posibles mejoras o trabajo futuro relacionado con este proyecto.

10.1. Conclusiones técnicas

Después de cuatro meses de trabajo integrado en el equipo de la empresa *Rubycon-IT* desarrollando este proyecto, se puede concluir que se han conseguido los objetivos:

- Implementación de una aplicación que satisface las necesidades del cliente.
- Realizar un sistema capaz de reconocer imágenes y descartar las imágenes que no pertenecen al conjunto de imágenes a reconocer en un tiempo inferior a 20 segundos.
- Diseñar el sistema modularmente, preparándolo para posibles extensiones en el futuro.
- Implementar reproductores de multimedia en nativo para el sistema operativo *iOS*.
- Creación de una base de datos con *MongoDB* en el servidor y otra con *Core Data* en local.
- Adaptar la aplicación para su correcto funcionamiento tanto en *iPhone 4* como en *iPhone 5*.

Por lo tanto, podemos concluir que los objetivos planteados al principio del proyecto, en el apartado 1.4 se han cumplido, por lo que el trabajo realizado es satisfactorio.

10.2. Conclusiones personales

A nivel personal, la experiencia de trabajo vivida en la empresa *Rubycon-IT* ha sido muy enriquecedora. En ella, el estudiante se ha dado cuenta de su capacidad para:

- Afrontar un proyecto que no sabe si es capaz de realizar y llevarlo a cabo.
- Aprender nuevos lenguajes de programación y aplicarlos de forma efectiva.
- Trabajar en un entorno laboral real.
- Integrarse en un equipo de trabajo real, dentro de la estructura de la empresa.
- Tomar decisiones determinantes para el proyecto, teniendo en cuenta las recomendaciones de los superiores, llegando a buen término.
- Participar en una metodología ágil a todos los niveles.

Y, además, fruto de esta experiencia, el estudiante considera que:

- Es fundamental saber trabajar en equipo para desarrollar *software* de forma rápida y adaptándose a las cambiantes necesidades del cliente y del mercado.
- Las metodologías ágiles son, como se nos ha explicado durante el grado, el nuevo paso que hay que seguir para progresar en la forma de realizar *software*.
- El código generado en equipo (con metodologías como por ejemplo el *Pair Programming* es un código de mayor calidad y que el equipo de trabajo reconoce como código de autoría compartida. Ningún miembro del equipo cree que el código es más suyo que de otra persona. Además, esta forma de trabajar cuenta con la ventaja de que, si un trabajador ha de ausentarse durante un tiempo, hay otros que tienen el mismo nivel de conocimiento de su trabajo, por lo cual el proceso no se estanca.
- En las nuevas metodologías de trabajo, la buena relación entre los integrantes del equipo es fundamental.

Todos estos objetivos se han podido llevar a cabo gracias a la formación recibida durante el Grado en Ingeniería Informática por la Universitat Jaume I, en la que se han transmitidos conocimientos que el estudiante ha puesto en práctica en la realización de este proyecto:

- **EI1020 Bases de Datos** y **EI1041 Diseño e Implementación de Bases de Datos**: realización de la base de datos.
- **EI1021 Sistemas Distribuidos**: diseño basado en el paradigma *cliente-servidor*.
- **EI1024 Programación Concurrente y Paralela**: paralelización en hilos del algoritmo comparador de descriptores.

- **EI1039 Diseño de Software**: patrones de diseño utilizados en la implementación del sistema.
- **EI1032 Análisis de Software** y **EI1040 Gestión de Proyectos de Ingeniería del Software**: planificación y gestión del proyecto.
- **EI1048 Paradigmas de Software**, **EI1049 Taller de Ingeniería del Software** y **EI1050 Métodos Ágiles**: conocimientos sobre planificación, gestión y realización de proyectos con metodologías ágiles.

10.3. Trabajo futuro

Este proyecto tiene varias líneas de desarrollo y, por ello, se dividen las posibles mejoras o ampliaciones que se podrían realizar para este proyecto.

- Reconocimiento de imágenes:
  1. Mejora de algoritmo de reconocimiento. Para cada fotografía reconocida, se podría recortar la imagen dentro de la fotografía (delimitar la imagen ya se hace dentro del reconocimiento) y asignar dicha imagen como una más a tener en cuenta en la base de datos. De esta forma, el sistema iría creciendo y aumentando las probabilidades de reconocer la imagen.
  2. Reconocimiento en *3D*. Investigar la posibilidad de, haciendo diversas fotos a objetos tales como, por ejemplo, un bote cilíndrico de galletas, reconocer objetos tridimensionales.
- Servidor:
  1. Realización de un servidor que pueda atender diferentes peticiones simultáneamente, utilizando un vector para cada petición.
- Aplicación móvil:
  1. Implementación de todas las vistas en modo horizontal del dispositivo.
  2. Implementación del sistema para *iPad*.
  3. Establecer un sistema de puntuación para los diferentes resultados devueltos, con el fin de que los usuarios pudieran valorarlos. De esta forma, si la aplicación se utilizara para fines publicitarios o comerciales, esta información se podría enviar (con el consentimiento del usuario) al servidor para que el cliente pudiera conocer la opinión que tienen los usuarios sobre cada uno de sus productos.
- *Backend*:
  1. Implementación de un sistema de *Backend* mediante un terminal *web* para añadir, editar y eliminar campañas e imágenes al sistema.
  2. Implementación de una aplicación móvil *iOS* para gestionar el sistema y poder añadir, editar y eliminar campañas e imágenes al sistema.

Bibliografía

[1] Alassian. Atlassian. piensa a lo grande. trabaja inteligentemente. lanza al mercado rápido. https://www.atlassian.com/es/?_mid=a015f3a6524e4ffc6332c19f3cc31b1c&gclid=CjkKEQjwlcSdBRD3wva3-KOAo80BEiQAjNIhiW5nXlm0-N5-xwmU_tjbuAR84ZD-LNIO5zZ2QPx1MQfw_wcB. [Consulta: 30 de Junio de 2014].

[2] Alassian. Jira. tu negocio. tus reglas. https://www.atlassian.com/es/software/jira. [Consulta: 30 de Junio de 2014].

[3] Apple Inc. Auto layout guide. https://developer.apple.com/library/ios/documentation/userexperience/conceptual/AutolayoutPG/Introduction/Introduction.html. [Consulta: 2 de Julio de 2014].

[4] Apple, Inc. Blocks programming topics. https://developer.apple.com/library/ios/documentation/cocoa/Conceptual/Blocks/Articles/00_Introduction.html. [Consulta: 30 de Junio de 2014].

[5] Apple Inc. Delegates and data sources. https://developer.apple.com/library/ios/documentation/general/conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html. [Consulta: 30 de Junio de 2014].

[6] Apple Inc. Designing for ios 7. https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/. [Consulta: 25 de Junio de 2014].

[7] Apple Inc. Grand central dispatch (gcd) reference. https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html. [Consulta: 29 de Junio de 2014].

[8] Apple Inc. Introduction to core data programming guide. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html. [Consulta: 3 de Julio de 2014].

[9] Apple Inc. ios developer program. https://developer.apple.com/programs/ios/. [Consulta: 27 de Junio de 2014].

[10] Apple Inc. Programming with objective-c. https://developer.apple.com/library/mac/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. [Consulta: 28 de Junio de 2014].

[11] Django Software Foundation. Django, the web framework for perfectionists with deadlines. https://www.djangoproject.com/. [Consulta: 28 de Junio de 2014].

[12] e-conomic. Definición de know how. http://www.e-conomic.es/programa/glosario/definicion-know-how. [Consulta: 23 de Junio de 2014].

[13] Gandy, Dave. Font awesome. the iconic font and css toolkit. http://fortawesome.github.io/Font-Awesome/. [Consulta: 1 de Julio de 2014].

[14] Iarocci, Nicola. Python rest api framework. http://python-eve.org/. [Consulta: 29 de Junio de 2014].

[15] Itseez. Opencv, open source computer vision. http://opencv.org/. [Consulta: 26 de Junio de 2014].

[16] Joyent, Inc. nodejs. http://nodejs.org/. [Consulta: 28 de Junio de 2014].

[17] Labbé, Mathieu. Find-object website. https://code.google.com/p/find-object/. [Consulta: 26 de Junio de 2014].

[18] Lindenberg, Tony. Feature detection with automatic scale selection. *International Journal of Computer Vision*, 30, 1998.

[19] Microsoft. Lambda expressions. http://msdn.microsoft.com/en-us/library/bb397687.aspx. [Consulta: 30 de Junio de 2014].

[20] MongoDB, Inc. Mongodb official website. http://www.mongodb.org/. [Consulta: 25 de Junio de 2014].

[21] Mountain Goat Software, LLC. Play. estimate. plan. http://www.planningpoker.com/. [Consulta: 30 de Junio de 2014].

[22] Open Source Initiative. Flask web development, one drop at a time. http://opensource.org/licenses/BSD-2-Clause. [Consulta: 26 de Junio de 2014].

[23] Oracle. Enterprise javabeans technology. http://www.oracle.com/technetwork/java/index-jsp-140203.html. [Consulta: 27 de Junio de 2014].

[24] Pivotal Software, Inc. Spring. let's build a better enterprise. http://spring.io/. [Consulta: 29 de Junio de 2014].

[25] Ries, Eric. The lean startup. http://theleanstartup.com/. [Consulta: 25 de Junio de 2014].

[26] Ronacher, Armin. The bsd 2-clause license. http://flask.pocoo.org/. [Consulta: 26 de Junio de 2014].

[27] Rubycon-IT. Rubycon information technologies. http://www.rubycon.es/. [Consulta: 24 de Junio de 2014].

[28] Thompson, Mattt. Afnetworking, a delightful networking framework for ios and osx. http://afnetworking.com/. [Consulta: 28 de Junio de 2014].

[29] Wells, Don. Pair programming. http://www.extremeprogramming.org/rules/pair.html. [Consulta: 24 de Junio de 2014].

[30] Wikipedia contributors. Client-server model. http://en.wikipedia.org/w/index.php?title=Client%E2%80%93server_model&oldid=613647194. [Consulta: 22 de Junio de 2014].

[31] Wikipedia contributors. Qr code. http://en.wikipedia.org/wiki/QR_code. [Consulta: 28 de Junio de 2014].

[32] Wikipedia contributors. Representational state transfer. http://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=614159381. [Consulta: 22 de Junio de 2014].

[33] Wikipedia contributors. Surf. http://en.wikipedia.org/wiki/SURF. [Consulta: 27 de Junio de 2014].