Clasificación y búsqueda de imágenes usando características visuales

Antonio Nicolás Pina

Universidad de Murcia

Facultad de Informática

Proyecto Fin de Carrera

Dirigido por

Ginés García Mateos

Junio de 2010

Resumen

En este proyecto se aborda el análisis, diseño e implementación de una aplicación para la clasificación y búsqueda automática de imágenes no segmentadas, permitiendo una rápida búsqueda por contenido sobre una base de datos que puede ser arbitrariamente grande. Para ello, se han aplicado dos tipos de técnicas de comparación: basadas en color y basadas en puntos característicos (*features*).

El primero de los métodos basados en color que se ha probado es el ***matching***, en el que la imagen de entrada se compara píxel a píxel con cada imagen de la base de datos dentro de un espacio de color multicanal. Para acelerar el proceso y aumentar la robustez de la comparación, se reducen todas las imágenes a un tamaño prefijado y se comparan con ese menor tamaño, aumentando así la coincidencia entre imágenes muy similares que, a una mayor escala, tendrían una gran diferencia entre ambas.

La segunda de las técnicas basadas en color que se ha implementado es el uso de **histogramas de color**, en los que se obtiene un resumen de las cantidades de color que cada imagen contiene, y la comparación se realiza sobre estos resúmenes, en lugar de hacerlo sobre los píxeles de la imagen. Al hacer las comparaciones entre imágenes sobre la cantidad de color que contienen y no sobre sus píxeles, se obtiene una cierta invarianza a los objetos que aparecen en la escena independientemente de su posición, centrándose sólo en el color que poseen, en cualquiera de los espacios de color que existen, lo que puede resultar interesante para algunas aplicaciones.

De entre los métodos basados en *features* se ha optado por utilizar SIFT (*Scale Invariant Feature Transform*), que permite describir objetos de una escena no segmentada con invarianza a la posición, escala y rotación. Una característica SIFT se define como el punto de máxima respuesta del operador diferencia de Gaussianas sobre un espacio de escala. Un espacio de escala es una pirámide de escalas de la imagen de entrada, calculada aplicando el suavizado de Gauss un número determinado de veces. Sobre la pirámide de escalas, se calculan unas imágenes de salida como la diferencia de los niveles de suavizado adyacentes. Finalmente, sobre estas imágenes de salida, una vez refinadas, se tiene como resultado unos puntos significativos, que serán los máximos locales de las diferencias de Gaussianas en ciertos valores de escala.

Además, se ha utilizado otro método de extracción de características, inspirado en SIFT, llamado SURF (*Speeded Up Robust Features*). **SURF** promete una obtención de características tan adecuada como SIFT pero con un menor coste computacional.

Debido a la gran necesidad de proceso y de memoria que los métodos basados en características requieren, se ha tenido que definir un método de clasificación que haga más eficiente la búsqueda, permitiendo así una mejor escalabilidad del sistema. De entre los métodos de clasificación que existen, los más adecuados son aquellos con estructura arbórea, ya que permiten obtener tiempos de búsqueda *sublineales*.

De entre éstos, se ha implementado un **árbol de *clusters*** de características, en el que cada nodo es dividido en nodos hijos mediante el algoritmo de **k-medias**.

Para medir la eficiencia de cada uno de los métodos anteriores, se han creado **dos bases de datos de imágenes de prueba**, que permitan realizar comparaciones fiables sobre la bondad de cada técnica en un entorno reducido y controlado. La primera de las bases de datos de prueba contiene 100 imágenes, distribuidas en 20 categorías, con un alto grado de similitud entre las imágenes de cada categoría, de forma que sea más sencillo para los métodos obtener un buen porcentaje de acierto, y así simular una base de datos real, en la que la inmensa cantidad de imágenes almacenada hace que muchas de ellas sean muy similares. La segunda base de datos contiene 50 imágenes en 10 categorías, con una mayor dificultad; esto es, con un menor parecido entre las imágenes de cada categoría, por lo que el método se verá obligado a hacer una mejor descripción de los elementos relevantes si quiere superar correctamente la prueba. Así, con los dos tipos de pruebas, se podrá ver qué método es más adecuado en cada situación.

Los **resultados** que se han obtenido para los métodos **basados en color han sido muy buenos**, tanto sobre la base de datos de imágenes simples como sobre la realista. Esto es debido a la similitud de colores que el mismo objeto tiene independientemente de cual sea la escena en la que se encuentra, siempre que su tamaño relativo a la escena lo haga suficientemente relevante.

En cambio, con los métodos basados en ***features*, la fiabilidad ha sido ligeramente inferior** que en los métodos de color, debido principalmente a la variación de pose que los objetos tienen entre imágenes, así como a la dificultad de obtener buenos comparadores de *features* en un entorno complejo y realista. Además, estos métodos han demostrado ser **mucho más lentos** que los basados en color, por lo que su uso queda muy limitado.

Además, el árbol de clasificación utilizado para acelerar las búsquedas ha complicado más el proceso de comparación de *features*, **empeorando** en algunos casos el resultado final, ya que cuando se consulta en el árbol se obtiene una lista de *features* sin ningún orden o relación aparente. Por otro lado, en el método de clasificación básico, en el que se tiene una lista global de *features*, es más sencillo obtener las características relacionadas de una foto y **establecer una biyección** entre ambas fotografías, facilitando así la obtención de buenos resultados. Por ello, podemos concluir que aunque el árbol acelera en gran medida las consultas respecto a tener una lista global de *features*, su utilización no ha aportado mejoras a la búsqueda, ya que el **porcentaje de aciertos es menor**, debido a esa dificultad de combinar las *features*.

Agradecimientos

Al principio, no creí que me fuera a dedicar a esto. Y es que la informática y los ordenadores llegaron a mi vida algo más tarde que a la gente de mi generación. El ajustado presupuesto familiar y los hermanos fuera de casa hicieron que un ordenador se viera como un lujo más que como la herramienta que hoy en día es, ¡y ya estábamos en el siglo XXI!

Sin embargo, un día llegó. No olvidaré aquel AMD K6-II que, con tanto esfuerzo y pago aplazado, mis padres me regalaron por navidad. Aquel fue el dia en que mi vida empezó a cambiar, y me condujo por unos caminos que ni me había planteado. Tal fue mi pasión que, cuando un día, casi por sorpresa, dije que iba a estudiar informática, a nadie de mi familia le extrañó mucho. Ahora me doy cuenta que, sin ellos, nada de esto hubiera sido posible.

Porque sin el cariño y la comprensión de mis padres, ni los consejos de mi hermana Jose, no podría haber aguantado los malos días, por eso les quiero dedicar este proyecto. Tampoco me puedo olvidar de mi prima Isabel, esa persona especial que me hace volver a tener fé en la humanidad, ni del resto de mi familia, que me han escuchado cuando lo he necesitado.

También quiero agradecer el esfuerzo a mis amigos, a algunos por su apoyo moral, y a otros por su colaboración, porque no sería justo que acabara sin mencionar a Daniel, Miriam, Fran, Juan y al resto de mis *followers*, porque leer mi *timeline* me hace un poquito más feliz, gracias a todos.

Finalmente, quiero agradecer su esfuerzo y dedicación a Ginés, a Alberto y Diego, algunos de esos pocos profesores buenos de la facultad, de los que disfrutan con su trabajo y que contagian su pasión a los alumnos. Sólo unas pocas personas tienen todo mi respeto y admiración, y estos profesores, sin duda, son unos buenos ejemplos de ello.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

"Una búsqueda comienza siempre con la suerte del principiante y termina siempre con la prueba del conquistador." - Paulo Coelho, El Alquimista.

En los últimos años, el reconocimiento de objetos en escenarios complejos, como los de una aplicación realista, ha cobrado gran importancia en el campo de la visión por computador. Hasta ese momento, el reconocimiento se limitaba a patrones simples, con un elevado número de restricciones para simplificar el problema. Por ejemplo, últimamente la aparición de robots de tercera generación ha hecho surgir la necesidad de poder percibir y aprender el entorno donde se opera. Esto implica la aplicación del reconocimiento visual de objetos en escenas complejas sin una segmentación previa.

Las nuevas técnicas de reconocimiento de objetos en escenas complejas y no segmentadas tienen un enorme potencial, no sólo industrialmente, sino en otras muchas áreas de la visión artificial. De hecho, múltiples empresas de *software* se han adentrado en este campo, dando a luz algunos sistemas interesantes que explotan las posibilidades que las citadas técnicas de visión artificial nos ofrecen. Quizá el ejemplo más famoso y que más expectación ha creado en los últimos tiempos sea el "Project natal"[1] de Microsoft.

Paralelamente, ha surgido un nuevo área de interés: la creación de buscadores de contenido eficientes y potentes, capaces de realizar sus búsquedas sobre inmensas bases de datos de imágenes. Los usuarios demandan cada vez más un servicio que les ofrezca la posibilidad de buscar contenido en una imagen, pero no únicamente por las etiquetas o la información textual asociadas a la imagen, sino por el propio contenido de la misma. Esto no es más que otro paso hacia la web semántica, en la que el buscador de alguna forma entiende los datos y las relaciones que entre ellos hay.

Sin embargo, la cantidad de datos que aparecen en la web crece muy rápidamente, aumentando la complejidad del problema exponencialmente. Como ejemplo, el popular servicio Flickr, almacena actualmente más de cuatro mil millones de fotos[2]; mientras que Facebook ya recibe tres mil millones cada mes[3], convirtiéndose así en el primer sitio del mundo en alojamiento de fotos.

[1] `http://www.xbox.com/en-US/live/projectnatal/`
[2] `http://blog.flickr.net/en/2009/10/12/4000000000/`
[3] `http://www.facebook.com/press/info.php?statistics`

1.1 Antecedentes históricos

La búsqueda de imágenes ha sido un problema ampliamente estudiado y, en contra de lo que pueda parecer, no es un campo de investigación exclusivo de los últimos años. Ya en los años 80 Banireddy Prasaad, Amar Gupta, Hoo-min Toong y Stuart Madnick del MIT desarrollaron IDBM, el primer motor de búsqueda de imágenes [1]. IDBM manejaba una estructura de fichero invertido de 5 niveles, en la que los dos primeros contenían una lista de etiquetas de texto y modificadores de esas etiquetas, respectivamente. Los otros tres niveles de la estructura se encargaban de localizar las imágenes en disco. Además, para mejorar la respuesta de la aplicación, incluyeron una base de datos de sinónimos, que era accedida antes de cada consulta, de modo que un usuario podía introducir una palabra no asociada a la imagen, pero que era sinónimo de una que si estaba asociada.

No obstante, sólo las mejoras en las capacidades de cálculo y de almacenamiento de los últimos años han permitido dar una solución viable para la búsqueda de imágenes en grandes bases de datos. Actualmente, hay una gran cantidad de buscadores de imágenes distintos, la mayoría privados, por lo que su funcionamiento interno es desconocido. Basándonos en sus características externas, podemos clasificar los buscadores en dos grandes tipos: basados en texto y en contenido.

En la tabla 1.1, extraída de [2], se muestra una lista de los principales buscadores comerciales, con una descripción de sus características más destacadas.

Nombre	Tipo			Tamaño del índice (estimado, en millones)
	Metadatos	Contenido	Contenido desde imagen externa	
Google Images[4]	Sí	Sí (sólo algunas)	No	
Bing image search[5]	Sí	Sí	No	
TinEye[6]	No	Sí	Sí	1100
MiPai[7]	Sí	Sí	Sí	100
Incogna[8]	Sí	Sí	Sí	100

Tabla 1.1: Buscadores comerciales de imágenes más conocidos.

1.1.1 Buscadores de imágenes basados en metadatos

Este tipo de buscadores se basan en asociar una serie de etiquetas de texto a cada imagen y buscar en base a estas etiquetas. Las etiquetas son extraídas del contexto de la imagen, usualmente del documento HTML en el que se encuentra.

Como la clasificación siempre se realiza entre cadenas de caracteres, pueden recoger una gran base de datos sin grandes requerimientos en tiempo y memoria. En cambio, la fiabilidad de los resultados es menor, ya que puede no existir una relación directa entre las etiquetas que el motor de búsqueda enlaza a la imagen y el contenido real de la misma.

Por ejemplo, una fotografía que se encuentre en medio de un documento en el que se habla sobre videoconsolas, es muy probable que tenga algún tipo de juego

[4] `http://images.google.com/`
[5] `http://images.bing.com/`
[6] `http://www.tineye.com/`
[7] `http://mipai.esuli.it/`
[8] `http://www.incogna.com/`

o accesorios. Además, es razonable pensar que tanto el nombre del fichero como la etiqueta "alt" de HTML asociada, tendrán una información valiosa que sirva para clasificar esa fotografía en uno u otro contexto.

Como parámetros de entrada necesitan de una o varias palabras. Esto los hace inicialmente más flexibles que los buscadores basados en imágenes, pues pueden hacerse búsquedas más complejas o con expresiones regulares. La verdadera potencia de un buscador basado en esta tecnología, reside en combinar de forma adecuada los resultados para cada palabra.

Algunos buscadores con métodos más avanzados incluso intentan *entender* qué pide el usuario y devolverle el resultado más adecuado para la consulta que pidió. Éste método requiere de técnicas de inteligencia artificial aplicadas a reconocimiento de cadenas de caracteres y no será tratado en este proyecto, aunque sí nos interesa ver cómo este tipo de técnicas pueden tener algunas partes en común con las basadas en contenido.

Como ejemplo, si buscamos en Google "películas", nos geolocaliza automáticamente y devuelve en primer lugar un enlace a la cartelera en la ciudad en que vivimos, incluyendo críticas, duración, horarios y la posibilidad de comprar las entradas directamente, como se puede ver en la figura 1.1.

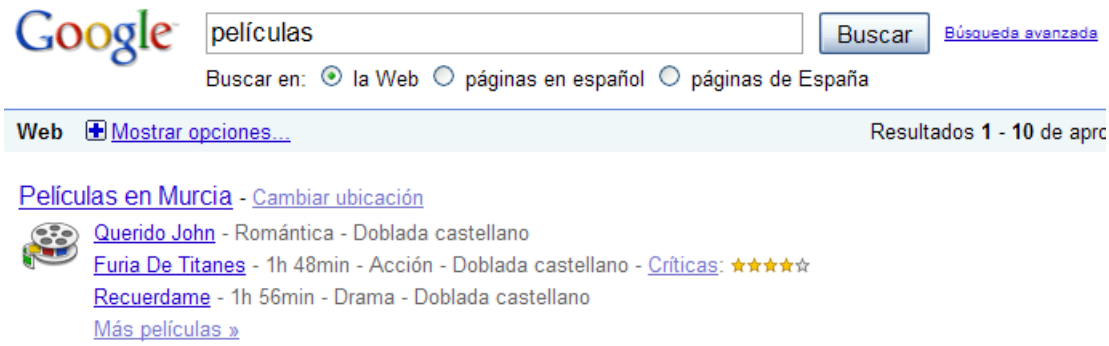

Figura 1.1: Resultados obtenidos al buscar "películas" en Google. El buscador comprende la semántica y devuelve la cartelera de la ciudad en la que vivimos.

En estos buscadores, la clasificación puede hacerse utilizando técnicas muy eficientes, en base a cadenas de caracteres que permiten órdenes de búsqueda *sublineales*, esto es, se puede hacer una búsqueda en la base de datos sin tener que realizar una comparación con cada uno de los elementos que hay almacenados en la misma. Además, cada comparación puede hacerse rápidamente al tratarse de simples cadenas de caracteres. Así, los requerimientos *hardware* no son tan amplios en cuanto a memoria y tiempo de ejecución, permitiendo tiempos de respuesta aceptablemente buenos en espacios de búsqueda relativamente grandes.

1.1.2 Buscadores de imágenes basados en contenido

Pertenecen a una segunda generación de buscadores, y en ellos se realiza la clasificación utilizando características extraídas del contenido de las imágenes de la base de datos, o alguna representación de las mismas. Su funcionamiento básico es muy sencillo, se toma la imagen dada como parámetro de la búsqueda, se extraen características asociadas a la misma y se comparan con las características extraídas de las imágenes que tenemos en la base de datos.

Son muy exigentes computacionalmente hablando ya que cada comparación entre dos elementos de la base de datos implica, generalmente, una gran carga de operaciones en coma flotante sobre matrices o vectores. Además, los métodos de clasificación no son tan eficientes ni fiables al tratar sobre espacios de dimensión mayor que con las cadenas de texto, y en los que los valores de cada posición tienen una gran variabilidad.

El método de extracción de características de cada imagen es un aspecto crucial en estos buscadores, ya que suele suponer un porcentaje elevado del éxito o fracaso de la búsqueda. Algunos de los métodos utilizados para extraer características de una imagen son:

- **Directo.** El objeto asociado a cada imagen es la propia imagen reescalada, convertida al espacio de color más adecuado según la aplicación. Como se verá en la sección 5.1, tanto el tamaño del reescalado como el espacio de color deben escogerse muy cuidadosamente porque de ellos dependerá en gran medida el éxito o fracaso del método.

  Así, la comparación entre dos elementos consiste en comparar dos imágenes píxel a píxel, utilizando algún método de los existentes, como la suma de diferencias al cuadrado o la correlación.

- **Textura.** Es un método un poco más avanzado que el anterior y consiste en extraer los *texels* de la imagen. Entendemos como *texel* (del inglés *texture element*) una matriz de píxeles de la imagen, lo más pequeña posible, de modo que una gran superficie de la imagen puede ser reproducida simplemente repitiendo ese patrón.

  Como ventaja, la comparación y clasificación serán mucho más rápidas al tratarse de una matriz de menor tamaño; en cambio, una de las desventajas que tiene es que es complicado encontrar la descomposición perfecta, si es que existe.

  Para comparar los texels suele utilizarse un análisis de la transformada de Fourier, como en [3] o, más recientemente, métodos basados en *wavelets*, como se explica en [4].

- **Color.** En este método se calcula un histograma de color de las imágenes, que se puede interpretar como un "resumen" de los colores predominantes que contiene la imagen. De este modo, puede tenerse rápida y fácilmente una vista general del aspecto que tiene la imagen dada y, por tanto, puede calcularse la similitud que tendrá con otra usando medidas de similitud entre histogramas. Se verá más en detalle en la sección 5.2.

Por supuesto, es crucial la elección del espacio de color a utilizar, ya que influirá en gran medida en los resultados del método. Otros aspectos relevantes son los canales a utilizar, el número de cubetas por cada canal, y la medida de comparación entre histogramas. Entre los espacios de color más frecuentemente utilizados podemos destacar HLS, HSV, YCrCb, YUV, etc.

- **Formas.** Es el descriptor más sofisticado técnicamente hablando y en el que este proyecto se adentrará en mayor medida en el capítulo 5. Aunque hay una gran cantidad de métodos, ya que es un área muy activa de investigación, todos se basan en la misma idea: encontrar y describir de alguna forma los puntos importantes de la imagen.

  El proceso de extraer los puntos importantes (*features*), es el que se lleva la mayor parte del tiempo empleado en una búsqueda y clasificación; mientras tanto, la clasificación y búsqueda se entienden como una tarea más liviana, aunque en ellas reside el mayor espacio de optimización y de mejora de cada método. Los métodos más recientes de extracción de características incluyen SIFT, SURF, FAST, FERNS.

En la figura 1.2 se muestra una comparativa de los métodos anteriores para una misma imagen, utilizando una imagen de ejemplo sacada de [5].

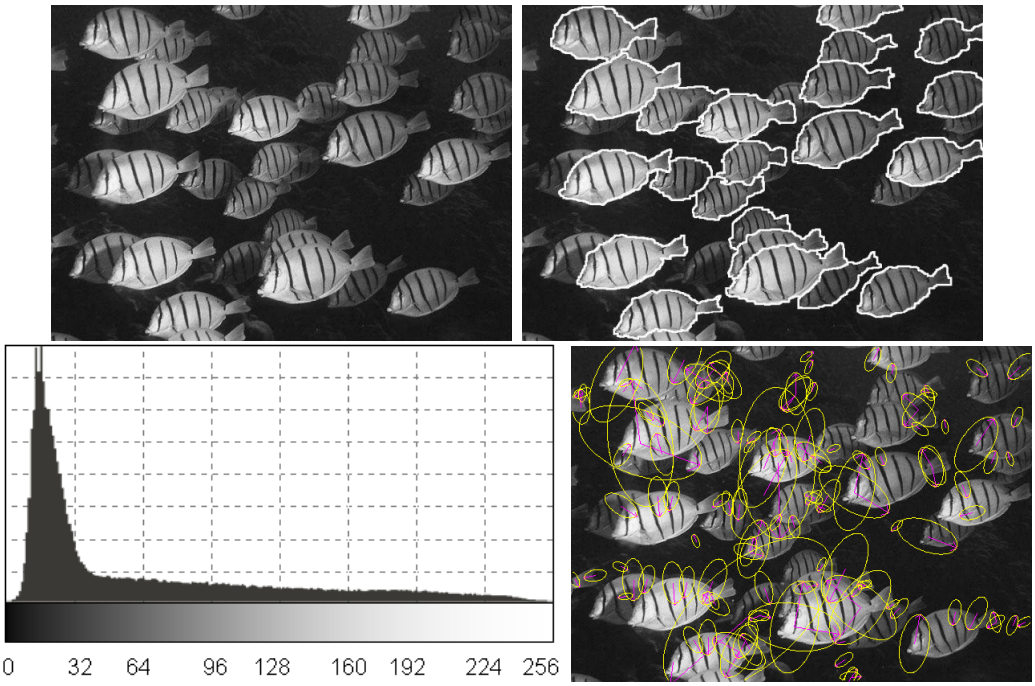

Figura 1.2: Resultados de aplicar diversos métodos a una misma imagen. De arriba abajo, de izquierda a derecha: imagen original, *texels* marcados en blanco sobre la imagen, histograma de color en gris y *features* SIFT dibujadas como una elipse con su orientación.

1.2 Aplicaciones del problema estudiado

En este proyecto, se estudia la construcción de un sistema que puede tener múltiples aplicaciones desde un punto de vista comercial. Para ello se pueden necesitar varios tipos de consultas diferentes, en función de lo que se quiera. Algunos tipos de consultas que se pueden implementar en un sistema de este tipo son:

- **Reconocimiento de categorías**. Este tipo de consulta requiere del sistema que clasifique la imagen de entrada y devuelva la categoría a la que pertenece, posiblemente en forma de etiquetas de texto.
- **Búsqueda de una imagen similar**. Se busca una imagen y se desea obtener otra lo más similar posible de entre las contenidas en la galería.
- **Búsqueda de similares**. Con una imagen de entrada, se buscan las más similares de la galería y se devuelven todas las que superen un cierto umbral de similitud, o un número fijo de ellas.
- **Búsqueda de patrones**. En este otro tipo, lo que se compara es la imagen de entrada con porciones de imágenes de la base de datos, de forma que se obtiene como resultado una lista de imágenes que contienen elementos similares. Por ejemplo, se pueden buscar patrones de caras sobre las fotos de una cámara de seguridad en las que aparecen personas, con el objetivo de identificar una persona en concreto.

Las consultas anteriores pueden ser utilizadas, por sí mismas o como una combinación de ellas, en muchas aplicaciones diferentes. Algunas de estas aplicaciones se detallan a continuación:

- **Búsqueda de imágenes**. El objetivo primario y, por tanto, uno de los campos de aplicación más interesantes es la búsqueda de imágenes relacionadas con una dada en grandes bases de datos. Puede ser muy útil para profesionales del diseño gráfico, que busquen fotos similares a un boceto o impresión que tienen sobre un cierto tema. Del mismo modo, puede ser utilizado por usuarios para encontrar mejores ampliaciones o diferentes ángulos de visión de un monumento o lugar que hayan visitado.
- **Catalogación de contenido multimedia**. Un sistema como el de este proyecto, puede ser utilizado para hacer una rápida clasificación no supervisada de elementos multimedia como, por ejemplo, fotos o vídeos que se suben a una red social y deben ser colocados en una categoría y con unas etiquetas relacionadas, como podrían ser “playa” o “bosque”.
- **Geolocalización**. Del mismo modo que hace Google con su aplicación Goggles[10] para Android, una foto o vídeo tomado en un cierto lugar podría ser consultada en una base de datos, quizá utilizando las imágenes a nivel de calle de Street View[11], y devolver una localización aproximada de dónde nos encontramos.

[10] `http://www.google.com/mobile/goggles/`
[11] `http://maps.google.com/help/maps/streetview/`

- **Reconocimiento de obras de arte**. Una foto de un cuadro dentro de un museo, o de la portada de un libro, bien podría reconocer la obra de la que se trata y darnos en nuestro móvil o tablet la información relacionada de la obra y autor.
- **Publicidad interactiva**. Con un sistema de este tipo podría reconocerse un logo o marca y mostrar la información relacionada como: tiendas más cercanas que tienen ese producto, comparativa de precios, etcétera.
- **Realidad aumentada**. Mediante el reconocimiento de patrones en la escena, pueden desarrollarse aplicaciones de realidad aumentada que permitan jugar en un mundo virtual dibujado sobre el real.
- **Interfaces gestuales**. Podrían segmentarse los gestos que una persona está haciendo y manejar una interfaz gráfica con ellos, sin necesidad de controladores adicionales.
- **Seguridad en aeropuertos**. Mediante cámaras de vigilancia pueden segmentarse las caras de los viajeros y buscarlas en una base de datos de delincuentes buscados, de modo que se pueda alertar si se reconoce la cara de uno de ellos.

1.3 Estructura del documento

El resto de la memoria del proyecto está organizada de la siguiente manera:

- En el capítulo 2 se concretan los objetivos de este proyecto y se describen y justifican tanto las herramientas utilizadas en el desarrollo, como la obtención de los casos de prueba.
- En el capítulo 3 se analiza el estado del arte de los buscadores de imágenes existentes, así como los campos de investigación más activos que intentan cubrir las necesidades presentes y futuras en este área.
- En el capítulo 4 se estudia la arquitectura del sistema y de una aplicación de este tipo, justificando algunas decisiones de diseño desde diversos puntos de vista: balanceo de carga, fiabilidad, seguridad, etc.
- En el capítulo 5 se detallan en profundidad cada uno de los métodos utilizados para la clasificación y búsqueda,
- En el capítulo 6 se incluye la experimentación hecha para cada método de los implementados, así como un análisis de parámetros para cada uno. Para terminar, se hace un estudio comparativo de los resultados obtenidos.
- Finalmente, en el capítulo 7 se analizan las conclusiones y se muestran las futuras vías de trabajo que se podrían seguir a partir del trabajo desarrollado en este proyecto.

2. Objetivos y metodología de trabajo

En este capítulo se presentan los objetivos del proyecto, destacando algunas de las principales propiedades que se desean para el sistema. A continuación, se describen y justifican las herramientas utilizadas en el desarrollo, así como las decisiones de diseño de las bases de datos de prueba que se utilizarán en el capítulo 6 para la experimentación con el sistema. Finalmente, se detallará el equipo utilizado en las pruebas y *benchmarks* realizados.

2.1 Objetivos

Con este proyecto se pretende analizar, diseñar y desarrollar un motor de búsqueda de imágenes que sea capaz de clasificar y consultar eficientemente una base de datos relativamente grande. Como parámetros de entrada, el usuario introducirá una imagen y la aplicación se encargará de devolver una lista de las más parecidas, con un grado de similitud, consumiendo los mínimos recursos posibles, sobre todo en tiempo de ejecución.

En el apartado 2.1.1 se presenta un estudio de lo que consideramos como las características más importantes que el buscador ha de tener. Así mismo, en el apartado 2.1.2 se comentan otros puntos interesantes para el diseño de la aplicación. Este estudio se tomará como base en el capítulo 4 para realizar un correcto diseño de la aplicación, de modo que se cubran de la mejor manera posible los requisitos establecidos.

2.1.1 Características más importantes

En este apartado se estudian las características, tanto funcionales como no funcionales, que consideramos imprescindibles para nuestra aplicación, y que habrán de guiar el diseño de la misma.

- **Ponderación de resultados.** Todos los buscadores estudiados incluyen, directa o indirectamente, una ponderación de cuán bueno es cada resultado que nos devuelve. Para ello, bien sea explícita o implícitamente, realiza un cierto cálculo y ordena los resultados en base a ese cálculo.

  Esto puede parecer trivial pero, como veremos en el capítulo 5, puede que algunos métodos no sean capaces de darnos con exactitud un valor para cada uno de los resultados, en cuyo caso habrá que decidir de alguna forma qué elementos deben colocarse en los primeros lugares y cuáles no.

- **Consulta libre.** Es importante que el usuario pueda realizar la búsqueda sobre cualquier elemento que desee, y que no se encuentre atado a algunos que nosotros le imponemos. Esto no significa que una consulta particular deba devolver por lo menos un resultado, eso dependerá en gran medida del tamaño de nuestra base de datos; sino que implica que el parámetro que el usuario ha introducido puede ser uno cualquiera, incluso uno que no hayamos visto antes.

  Por ejemplo, Google Images no permite esto y restringe las búsquedas similares a un subconjunto de su base de datos que, aunque muy amplia, no recoge todas las posibles imágenes que el buscador ofrece.

- **Consulta por contenido.** Es una característica básica en nuestro proyecto. Aunque que complica en gran medida el algoritmo del buscador, permite realizar consultas en base a dibujos, fotografías o meras aproximaciones de lo que queremos, sin especificar etiquetas de texto o cualquier otro tipo de información externa.

  Como veremos en la sección 3.1, no todos los buscadores de imágenes estudiados incorporan esta opción, y los que la incluyen lo hacen con suerte dispar.

- **Escalabilidad.** Relacionado con el punto anterior, queremos no sólo que vaya rápido, sino que escale bien. Es imperativo que se pueda repartir el trabajo entre un número alto de servidores y que el aumento del rendimiento global sea cercano a un crecimiento lineal, de modo que si se pone el doble de *hardware*, por ejemplo, en término medio se puedan realizar cerca del doble de consultas simultáneamente.

  De igual manera, sería muy interesante que nuestros algoritmos sean capaces de manejar cantidades cada vez más grandes de datos sin que los requisitos del sistema aumenten exponencialmente. Sólo así nuestra aplicación podrá manejar razonablemente un volumen creciente de imágenes.

  Para lograr esto, debería asegurarse que un balanceo de carga puede ser realizado de manera sencilla y eficiente.

2.1.2 Otras características deseables

En este apartado se comentan otras características que pueden resultar interesantes para un buscador de imágenes por contenido, pero que no consideramos imprescindibles para una primera versión de la aplicación.

- **Consulta combinada.** Una opción muy interesante que muy pocos sistemas incorporan es la posibilidad de realizar una consulta combinada por contenido y metadatos. De los estudiados, sólo MiPai incorpora la opción de especificar una cadena de texto para reordenar los resultados de una anterior consulta por contenido.

  Como se ha dicho anteriormente, lo realmente interesante sería poder utilizar esa información de manera combinada, de modo que se incorpore en el algoritmo de búsqueda tanto el contenido como los metadatos asociados a ese

contenido. Así podríamos, en principio, combinar lo mejor de los dos mundos y obtener de una forma sencilla unos resultados con una mayor precisión.

- **Velocidad.** Por supuesto, deseamos que la resolución de una consulta se realice en un tiempo razonable, digamos del orden de un segundo. Por ello, descartaremos aquellos métodos que nos supongan un tiempo de búsqueda muy alto, pese a que puedan obtener buenos resultados, pero en un entorno real no serían aplicables.

- **Velocidad de recuperación.** Por último, y no menos importante, el sistema debe poder recuperarse rápidamente de cualquier problema que surja. Si se consigue que el buscador pueda ponerse en ejecución en un tiempo aproximadamente constante, independientemente del tamaño de la base de datos, una caída en un nodo de ejecución podrá ser solucionada con cierta celeridad para entorpecer en la menor medida posible el funcionamiento del sistema completo.

2.2 Entorno de programación

En esta sección se describirán y se justificarán las herramientas de programación que se han utilizado para el desarrollo de la aplicación, tanto para el núcleo como para su *interface* web.

2.2.1 Programación del núcleo de la aplicación

Para la aplicación encargada de clasificar y hacer las búsquedas se ha decidido hacer una implementación en C++. En primer lugar, C++ provee una plataforma eficiente y muy probada de desarrollar aplicaciones. Además, gracias a su compatibilidad con C, existen librerías ya implementadas para casi todo.

Por otro lado, existe el conjunto de librerías Boost[1], que implementan en C++ partes de lo que será el futuro estándar del lenguaje, la mayor parte del *Technical Report 2* (TR2), que formará la nueva evolución del lenguaje cuando, previsiblemente, a finales de 2011 sea aprobado[2].

Las librerías Boost permiten conceptos tan ajenos, en principio, para C++ como la programación funcional o la metaprogramación, con la que se deja al compilador que resuelva estáticamente algunos de los cálculos más pesados del programa que, en tiempo de ejecución, sólo tendrán que ser recuperados con poco o ningún procesamiento.

En concreto, las partes de las librerías Boost que se han utilizado son:

- **Punteros compartidos**. Permiten objetos compartidos entre varias partes del programa, que son liberados sin ningún peligro cuando dejan de ser necesarios.

- **Sistema de ficheros**. Mediante unas cuantas clases, se abstraen los detalles del sistema de ficheros que se esté utilizando, por lo que puede portarse la aplicación entre sistemas operativos sin apenas modificaciones.

[1] `http://www.boost.org/`
[2] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2697.html`

- **Opciones de programa**. Boost permite especificar una lista de parámetros que el programa necesita o acepta, tanto por línea de comandos como por fichero de configuración, y es la librería la encargada de reconocerlos y devolverlos utilizando un mapa asociativo.

- **Serialización**. Sin duda, es una de las opciones más interesantes de las que ofrece Boost.

  Implementando unos sencillos métodos en cada clase que queramos, puede almacenarse el estado de un objeto y recuperarse de forma casi transparente a la aplicación.

Para la parte de procesamiento gráfico se han utilizado las librerías Intel IPP[3] y OpenCV[4], ya que ofrecen un muy buen rendimiento sobre arquitecturas x86 y AMD64, además de disponer de una buena documentación y soporte.

IPP es una librería de primitivas visuales para C desarrollada por Intel especialmente optimizada para procesadores de su marca, por lo que resulta muy adecuado su uso en el equipo que se ha utilizado para las pruebas, descrito en el apartado 2.4.

OpenCV es una librería *open-source* de procesamiento visual para C, que puede hacer uso de IPP, y que cuenta con el apoyo de una gran comunidad de usuarios, por lo que dispone de funciones implementadas para muchas de las tareas habituales, incluyendo el algoritmo de k-medias y SURF, utilizados en este proyecto.

La implementación de las características SIFT no se incluye dentro de OpenCV, debido a que es una técnica patentada y sujeta a derechos, por lo que se utilizó inicialmente la librería VLFeat[5], pero al no conseguir hacer que funcionara, se optó por utilizar la implementación de SIFT de Rob Hess[6]. Es una librería para C que se apoya en OpenCV 2.0, que se ha adaptado *ex profeso* para compilar con C++ resolviendo algunas incompatibilidades.

Para la comunicación entre la *interface* web del buscador y el núcleo de procesamiento, se ha utilizado el protocolo XMLRPC, mediante la librería XMLRPC-C, una librería para el lenguaje C, con algunos *wrappers* para C++, que permite la actuación como cliente o servidor de una aplicación.

2.2.2 Programación del interface web

En la programación de la web se ha utilizado el lenguaje **PHP**[7], por disponer de un mayor soporte y madurez de herramientas, con el *framework* CodeIgniter[8].

CodeIgniter es un *framework* para PHP que facilita el rápido desarrollo de una aplicación web mediante librerías de funciones ya implementadas, proveyendo un espacio de trabajo orientado a objetos basado en la arquitectura Modelo-Vista-Controlador[9]. En este tipo de arquitectura, existe un controlador que separa la lógica

[3] `http://software.intel.com/en-us/intel-ipp/`
[4] `http://opencv.willowgarage.com/wiki/`
[5] `http://www.vlfeat.org/`
[6] `http://web.engr.oregonstate.edu/~hess/#[[SIFT%20Feature%20Detector]]`
[7] `http://www.php.net/`
[8] `http://codeigniter.com/`
[9] `http://es.wikipedia.org/wiki/Modelo_Vista_Controlador`

de programa de los datos, que se obtienen del modelo, y de su representación en la pantalla, que se escribe en la vista. Además, CodeIgniter dispone de una clase cliente XMLRPC, por lo que resulta especialmente adecuado para la implementación que se necesita.

Para la implementación de las vistas, se ha seguido el estándar **XHTML 1.0 Transitional**[10], consiguiendo que la web valide sin errores en el test del W3C. También se han validado las hojas de estilos **CSS** escritas para la web.

Para conseguir que la aplicación interactúe con el servidor, se han utilizado llamadas **AJAX** (Asynchronous JavaScript And XML) codificadas en **JSON** (JavaScript Object Notation), utilizando la librería **Prototype** para JavaScript.

Finalmente, para transferir la imagen de entrada desde el cliente web hasta el servidor en segundo plano, se ha utilizado **Uber Uploader**[11], un grupo de programas GPLv3 escritos en PHP, Perl y JavaScript.

2.3 Base de datos de pruebas

En esta sección se describe el proceso que se ha realizado para la obtención de una buena base de datos para pruebas, ya que una buena elección de la misma permite una correcta comparación entre métodos. Sólo con una comparación adecuada pueden extraerse conclusiones sobre el funcionamiento de cada uno de los métodos implementados, tarea difícil cuando se trata de comparar la eficacia de algoritmos sobre conjuntos de datos reducidos y, por tanto, diferentes de los datos reales.

En primer lugar, se ha diseñado una base de datos de pruebas para histogramas de color, consistente en 20 categorías con 5 imágenes cada una (100 imágenes en total), eligiendo las imágenes de cada categoría para que sean sencillas de reconocer, con un alto grado de similitud aparente entre las mismas. Para ello, se ha utilizado una herramienta de Google Labs, Image Swirl[12], que mediante una atractiva interfaz Flash permite buscar por similitud sobre un reducido conjunto de fotos, organizando las fotos en forma arbórea según su parecido. En la figura 2.1 se muestra un ejemplo de una categoría de las contenidas en dicha base de datos.

También se creó una segunda base de datos de prueba, que incluye imágenes con una mayor dificultad para todos los métodos, especialmente para los basados en *features*, con 10 categorías de 5 imágenes cada una (50 imágenes en total), en la que, como puede verse en la figura 2.2, las similitudes entre las imágenes de la misma categoría son menores.

Para obtener un caso de prueba más realista, se ha utilizado un *benchmark* especial para reconocimiento de imágenes[13], publicado en [6], y que consiste en 10200 imágenes, divididas en 2550 categorías con 4 imágenes cada una, como se muestra en la figura 2.4.

Finalmente, como un caso de prueba realista y más amplio, se escribió un pequeño script en Bash y PHP que descarga las 500 fotos destacadas del día en Flickr, para todos los días de un año dado, lo que da un total de unas 180000 imágenes, cada una

[10] `http://www.w3.org/TR/xhtml1/`
[11] `http://uber-uploader.sourceforge.net/`
[12] `http://image-swirl.googlelabs.com/`
[13] `http://vis.uky.edu/~stewe/ukbench/`

Figura 2.1: Una categoría de ejemplo de las contenidas en la galería de imágenes sencilla, que contiene 20 categorías con 100 imágenes en total.

Figura 2.2: Una categoría de ejemplo de las contenidas en la galería de imágenes compleja, que contiene 10 categorías con 50 imágenes en total.

con una serie de etiquetas de texto asociadas, extraídas de las etiquetas de Flickr. Este *benchmark* ha sobrepasado la capacidad de memoria del equipo utilizado en todas las pruebas intentadas, por lo que se necesitará mejorar la aplicación en el futuro para cubrir esa cantidad de fotografías.

2.4 Equipo de pruebas

Todas las pruebas documentadas aquí se han realizado utilizando la siguiente configuración:

- **Sistema operativo**. El sistema operativo sobre el que se ha implementado y realizado las pruebas es una distribución de GNU/Linux, en concreto Ubuntu, en su versión 9.10, compilada para 64 bits.
- **Procesador**. El procesador es un doble núcleo Intel Core2 Duo T6500, con 2MiB de caché L3 y 2.1GHz de frecuencia de reloj.

Figura 2.3: Ejemplo de la base de datos de imágenes realista.

Figura 2.4: Una categoría de ejemplo de las contenidas en la galería de imágenes realista, que contiene 2550 categorías con 10200 imágenes en total.

- **Memoria RAM**. El equipo dispone de 4GiB de memoria RAM, configurada en Dual Channel, con dos bancos DDR2 a 667MHz.
- **Disco duro**. Se dispone de 500GB de disco, formateado con el sistema de ficheros ext4, a una velocidad de 5400 rpm.

La ejecución de las pruebas se ha hecho desde el entorno gráfico GNOME, sin ninguna otra aplicación abierta, para evitar variación de resultados entre pruebas debido a posibles interferencias entre aplicaciones. Además, para asegurar la fiabilidad, cada prueba ha sido ejecutada una segunda vez para comprobar que los resultados obtenidos en ambas ejecuciones son similares y, por tanto, correctos.

3. Estado del arte

En este capítulo se realizará un estudio de algunos de los principales buscadores comerciales disponibles en el momento de realizar este proyecto, así como de otros sistemas que el autor cree interesantes para la correcta comprensión de este documento. En la sección 3.1 veremos cómo se intentan cubrir algunos de los requisitos que se definieron en el apartado 2.1.2 y, sobre todo, en el apartado 2.1.1.

Además, se presentan algunas de las técnicas más usadas hoy en día en el reconocimiento y segmentación de objetos en el campo de la visión por computador.

3.1 Principales buscadores comerciales

En esta sección analizaremos las características que algunos buscadores, de los incluidos en la tabla 1.1, ofrecen al usuario, no desde el punto de vista de usabilidad o sencillez, sino fijándonos en la funcionalidad que ofrecen. Desafortunadamente, al ser sistemas privados, no hay documentación sobre las técnicas en las que se basan.

3.1.1 Google Images

El buscador de imágenes de Google es posiblemente el más conocido y también uno de los más usados, pero no por ello el mejor. Si bien es cierto que para un gran porcentaje de búsquedas devuelve unos resultados aceptables, esto no se debe a su algoritmo interno, como veremos. Esta afirmación también es aplicable a "Bing image search", ya que es muy similar a éste y, por ello, se obvia su estudio.

En primer lugar, lo más destacable que vemos al entrar es que se trata de un buscador basado en metadatos, como los descritos en la sección 1.1.1, aunque posteriormente se han incorporado algunas características orientadas a contenido. Esta búsqueda por contenido se limita a ofrecer un enlace "buscar imágenes similares" bajo algunos resultados de la búsqueda, como puede verse en la figura 3.1; pero en ningún caso nos permite utilizar otra imagen, ya sea desde otra *URL* o una foto que el usuario envíe directamente.

Para comenzar - y dado que no se pretende hacer un estudio en profundidad - realizaremos una simple búsqueda con una palabra que puede resultar más o menos complicada: "roborovski".

Como vemos en la figura 3.1, los primeros resultados son correctos: en todos devuelve fotos de hámsters *roborovski* y además el tiempo de búsqueda ha sido muy bajo. Como hemos dicho, esta búsqueda se ha realizado utilizando metadatos y, por tanto, pueden utilizarse algoritmos muy eficientes pero no se está buscando realmente en el interior de la foto sino simplemente en las etiquetas que se han asociado a esa imagen.

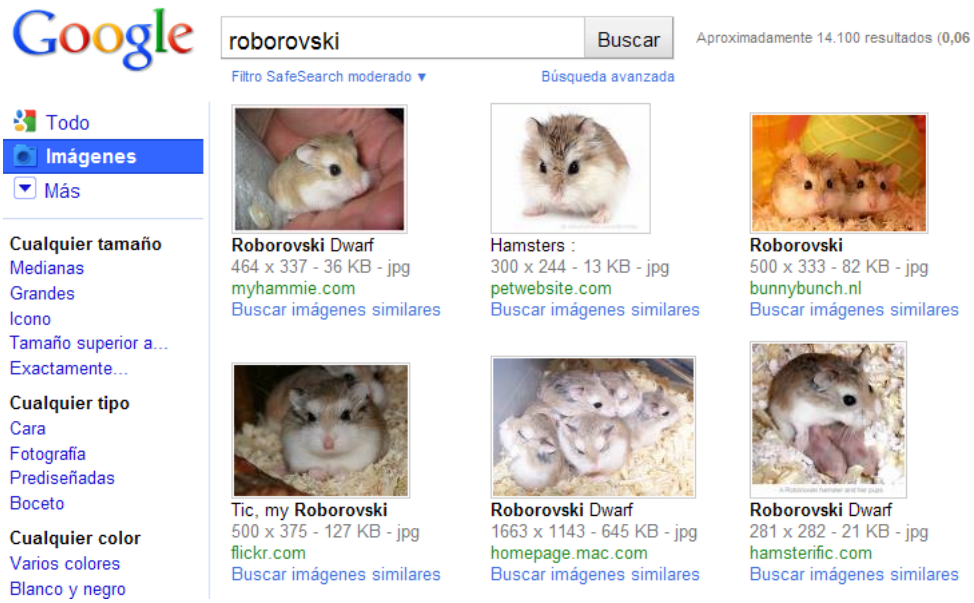

Figura 3.1: Resultados devueltos por Google Images al buscar "roborovski".

Aun suponiendo que esta búsqueda por metadatos nos devuelva el objeto que queremos, como en este caso sucede, puede que lo que estamos buscando no sea "cualquier foto", sino una en concreto, o un subconjunto muy reducido. En ese caso, nuestra búsqueda se complica y puede que no sea realizable mediante metadatos. Para esto es para lo que Google ha introducido el enlace "imágenes similares".

Utilizando esta posibilidad sobre, por ejemplo, la primera imagen de la anterior consulta, nos devuelve los resultados que aparece en la figura 3.2. Como vemos, en este caso las imágenes devueltas tienen un mayor parecido entre sí que las de la figura 3.1.

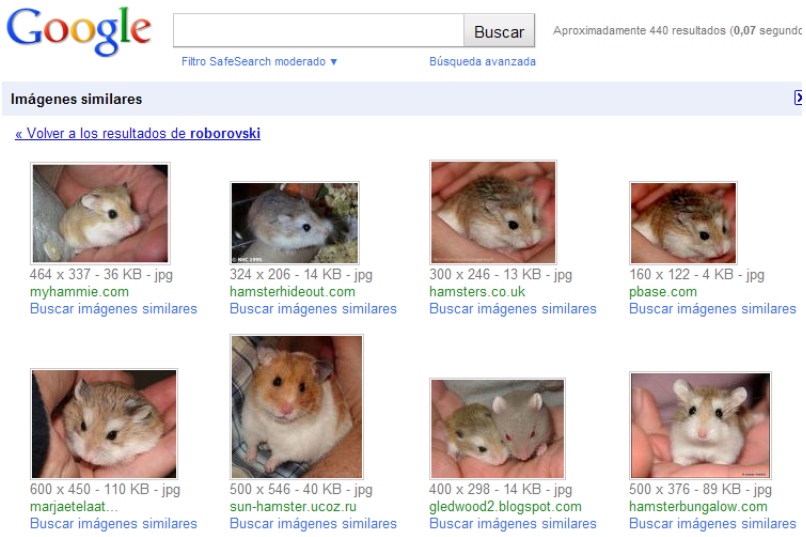

Figura 3.2: Resultados devueltos por Google Images al buscar "roborovski" y después buscar imágenes similares al primer resultado.

Es por esto que consideramos muy interesante en un buscador que nos permita suministrar un patrón de búsqueda por contenido. Así, sin necesidad de disponer de una etiqueta, podríamos obtener unos resultados bastante cercanos a lo que queremos. Por ejemplo, si tenemos una foto de nuestro hámster pero no sabemos su raza, podríamos realizar una búsqueda de la imagen en la web; al encontrar imágenes parecidas de hámsters roborovski, ya sabríamos la raza del nuestro.

Finalmente, citar las restricciones que Google Images nos permite realizar sobre una consulta por metadatos, aunque nunca dentro de las "similares". Algunas de estas son:

- **Tamaño.** Sobre los resultados de la consulta de metadatos, permite filtrar aquellos que sean mayores, menores o igual a un cierto tamaño. Puede resultar interesante, pero no ayuda en el proceso de clasificación ni de recuperación de las soluciones, sólo es un filtro independiente añadido sobre ellas.

- **Caras.** Es una opción muy interesante, aunque lo sería aún más si ofreciera un mayor rango de tipos de objetos, no únicamente caras humanas. Un resultado de esta búsqueda puede verse en la figura 3.3.

Figura 3.3: Resultados devueltos por Google Images al buscar caras de la famosa soprano Angela Gheorghiu.

Como se verá en el capítulo 5, sería muy interesante inferir una base de datos de objetos reconocidos en las imágenes y colocarles una etiqueta como, por ejemplo, "cara". Así, podríamos combinar la búsqueda por metadatos y por contenido, aunque éste es un proceso muy costoso.

- **Colores.** Del mismo modo, Google Images nos permite obtener sólo aquellos resultados en los que predomine un cierto color. A primera vista, no parece muy complicado realizarlo, se entrará en detalle en el capítulo 5.

3.1.2 TinEye

TinEye nos ofrece un buscador puramente orientado a contenido, sin posibilidad de realizar o filtrar los resultados por metadatos. Aunque dicen tener más de mil millones de imágenes indexadas, nuestra búsqueda del hámster no ha devuelto ningún resultado, como se ve en la figura 3.4. Haciendo una consulta más sencilla, sólo obtenemos la misma imagen desde distintas fuentes, como en la figura 3.5.

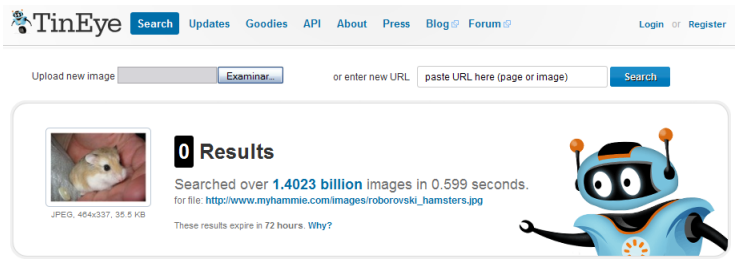

Figura 3.4: Resultados devueltos por TinEye al introducir la foto de un hámster roborovski.

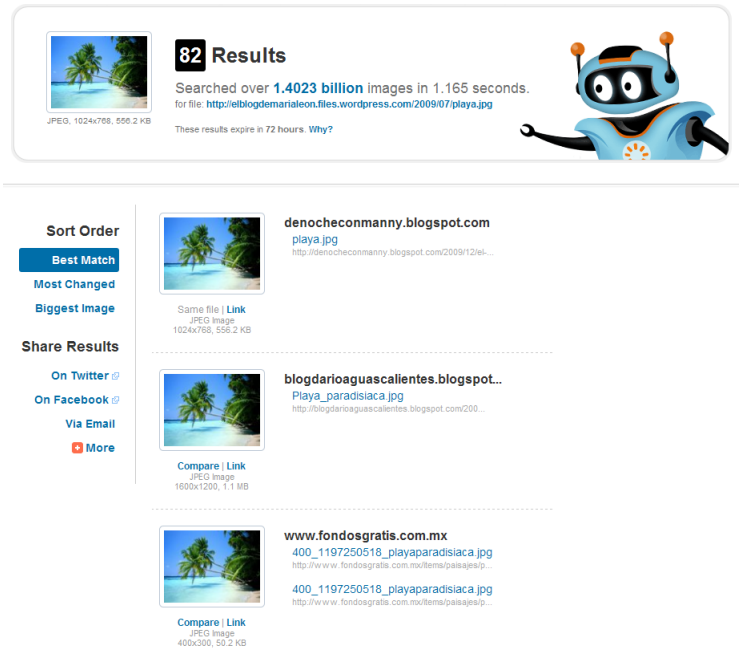

Figura 3.5: Resultados devueltos por TinEye al introducir una consulta más sencilla, la foto de una playa.

Por ello, nos parece que TinEye aún tiene mucho que mejorar y que debe extender sus métodos de búsqueda más allá de una simple comparación directa entre imágenes como la de la sección 5.1, que es la que parece realizar.

3.1.3 MiPai

A diferencia de TinEye, MiPai sí nos ofrece la posibilidad de buscar por metadatos y, además, buscar por contenido desde una *URL* o desde un fichero local subido a su servidor.

Figura 3.6: Resultados devueltos por MiPai al introducir la foto de un hámster roborovski.

En la figura 3.6 se muestran las respuestas obtenidas al realizar una consulta por contenido. Si observamos los resultados detenidamente, aunque en ninguno parece acertar, en todos hay un patrón reconocible: predominan una serie de colores pertenecientes a los de la imagen de entrada. Por ello, podemos intuir que MiPai utiliza un algoritmo por color como los que veremos en la sección 5.2.

Lo interesante de este buscador es que nos permite, una vez realizada la consulta por contenido, reordenar los resultados en base a una etiqueta de texto suministrada por el usuario. El resultado de esto puede verse en la figura 3.7.

3.2 Técnicas más usadas actualmente

Antes de llegar a las técnicas que se utilizan en la actualidad, y sobre las que se investiga más activamente, vamos a hacer un repaso de una de las técnicas que se han aplicado tradicionalmente y que, actualmente, continúa siendo una de las más usadas en la mayoría de proyectos por su sencillez.

3.2.1 Detección de bordes

Se define un borde como la zona en la que una imagen cambia con mayor rapidez su nivel de gris. Como la detección de bordes se realiza utilizando la primera derivada, si la imagen de entrada es multicanal, habrá que decidir qué hacer con cada canal.

La detección de bordes de una imagen se basa en el hecho de que en un borde la primera derivada se comportará de una forma peculiar y predecible. Si una zona de la imagen tiene valores del nivel de gris muy cercanos entre sí, la derivada tendrá un

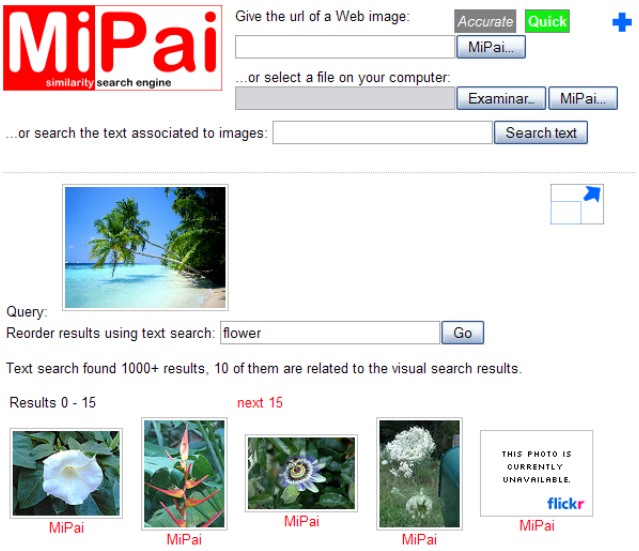

Figura 3.7: Resultados devueltos por MiPai a una consulta combinada, en la que se introduce la foto de una playa y se reordenan los resultados mediante la palabra *flower*.

valor pequeño y cercano a cero, mientras que si el valor varía rápidamente, la derivada tomará un valor alto con signo positivo o negativo, según si el borde era creciente o decreciente en su nivel de gris. Es por esto que el resultado suele representarse como una imagen gris en la que se marcan los bordes en blanco o negro, según el signo.

De entre los algoritmos para detección de bordes, probablemente el más famoso y utilizado sea el **algoritmo de Canny**, disponible en [7] y explicado en español en [8]; fue publicado en 1986 y consta de cuatro pasos principales:

1. **Suavizado**. Como puede verse en la figura 3.8, aplicar el algoritmo de Canny directamente sobre una imagen produce gran cantidad de falsos bordes, debido al ruido de la imagen. Por ello, se hace un paso previo en el que se aplica a la imagen un suavizado Gaussiano de radio pequeño, para reducir ese ruido sin alterar la localización de los bordes.

2. **Cálculo del gradiente**. Dado que la imagen pertenece a un espacio bidimensional, no se puede calcular la derivada en términos absolutos, sino que se utilizan las derivadas parciales en cada variable. Así, sobre la imagen suavizada, se calcula el gradiente como las derivadas parciales en x e y, resultando las dos imágenes como las que pueden verse en la figura 3.9.

3. **Supresión de no máximos**. En este paso se asegura que los bordes no tienen más de un píxel de ancho, calculando en cada píxel una dirección aproximada del gradiente, y eliminando los píxeles que no se encuentre en dirección perpendicular al gradiente. De este modo, utilizando las dos imágenes generadas en el paso anterior y con la aproximación de la dirección que sigue el gradiente, cada borde se reduce a una línea de un píxel de ancho, como puede verse en la figura 3.10.

Figura 3.8: Resultado del algoritmo de detección de bordes de Canny con diferentes suavizados. De izquierda a derecha: imagen original, resultado con un suavizado medio y resultado con un nivel de suavizado mayor.

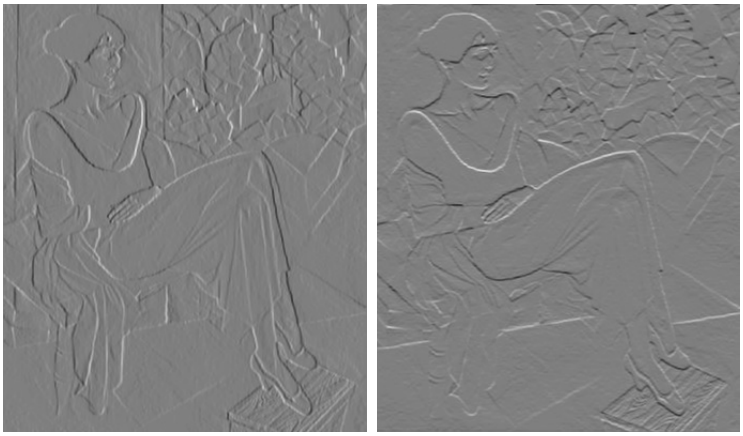

Figura 3.9: Derivadas parciales calculadas en el algoritmo de detección de bordes de Canny. De izquierda a derecha: derivada parcial vertical y derivada parcial horizontal. En blanco o negro se muestra el signo positivo de la derivada.

4. **Umbralización con histéresis**. Finalmente, se binariza la imagen resultante, para obtener una imagen binaria, en la que cada píxel indica si es un borde o no. Este proceso depende de dos variables, un valor máximo y un mínimo para el algoritmo: en primer lugar, se marcan como borde todos los píxeles cuyo valor sea superior al valor máximo dado, y luego se continúa el borde en los píxeles adyacentes (perpendiculares a la dirección del gradiente) que tengan un valor superior al valor mínimo que se proporciona como entrada. En la figura 3.11 se muestra el resultado de aplicar este proceso con diferentes valores para dichos umbrales.

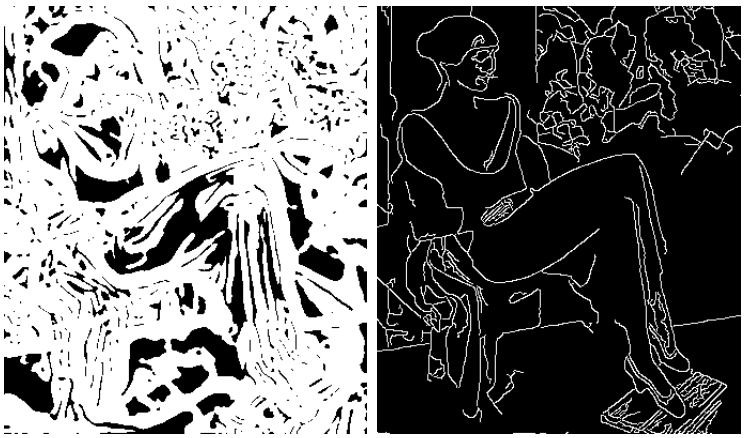

Figura 3.10: Resultado del algoritmo de detección de bordes de Canny. De izquierda a derecha: Antes de la supresión de no maximos y después.

Figura 3.11: Resultado del algoritmo de detección de bordes de Canny con diferentes umbrales. En la imagen de la derecha se ha aplicado un umbral más restrictivo que en la izquierda.

3.2.2 Técnicas basadas en color

Hay otro tipo de técnicas Dentro de las técnicas basadas en color, destacan los histogramas de color y el *matching* de imágenes.

El ***matching*** consiste en comparar dos imágenes píxel a píxel. De este modo, el descriptor asociado a cada imagen es un reescalado de la propia imagen, convertida a cualquier espacio de color. Esta técnica, además de ser muy rápida y sencilla, permite conseguir buenos resultados cuando se dispone de bases de datos lo suficientemente grandes, ya que es muy probable que se tengan imágenes muy similares a la pedida ya clasificadas. Por otro lado, si lo que queremos es reconocer objetos en escenas no segmentadas, este método no es adecuado porque no reconoce ningún tipo de contenido dentro de la imagen, sino que se limita a comparar, dando el mismo peso a todos los píxeles de la imagen, sin importar si son interesantes o no. En la sección 5.1 se hace un estudio exhaustivo de esta técnica.

La técnica basada en **histogramas de color** calcula el resumen de la cantidad de color que contiene una imagen, y utiliza esta información para representar a la

imagen. Esta técnica permite abstraer la posición y rotación de los objetos de la imagen, dado que sólo interesa la cantidad de color que poseen. Éste es uno de las puntos a favor de los histogramas de color respecto al *matching* de imágenes. En cambio, como desventaja, tenemos que dos imágenes que tengan un color parecido, serán reconocidas como muy similares, cuando la escena puede ser muy diferente y que simplemente la cantidad de color sea la misma. En la sección 5.2 se hace un análisis de esta técnica en profundidad.

3.2.3 Características SIFT y SURF

Hoy en día, una buena parte de la investigación para el reconocimiento de objetos en entornos no segmentados se centra en detectores de características, principalmente con métodos basados en SIFT y similares. Como se verá en la sección 5.3, SIFT es un método de extraer los puntos relevantes de una imagen de forma que sean independientes a la escala, rotación, traslación y a una pequeña variación en la pose tridimensional de los objetos.

Debido a su éxito en el ámbito científico, su uso está muy extendido y han surgido gran cantidad de métodos basados en SIFT, como SURF, que utiliza *wavelets* para mejorar la velocidad del método con unos resultados similares.

El punto débil de estas técnicas suele ser la complejidad de cálculo de los descriptores que representan cada uno de los puntos detectados para hacerlos independientes a la rotación, posición, tamaño e iluminación. Como se ve en la figura 3.12, en la que se ha aplicado el detector SURF, la mayor parte de puntos interesantes han sido detectados, dejando pocos o ninguno en áreas menos relevantes, como el cielo.

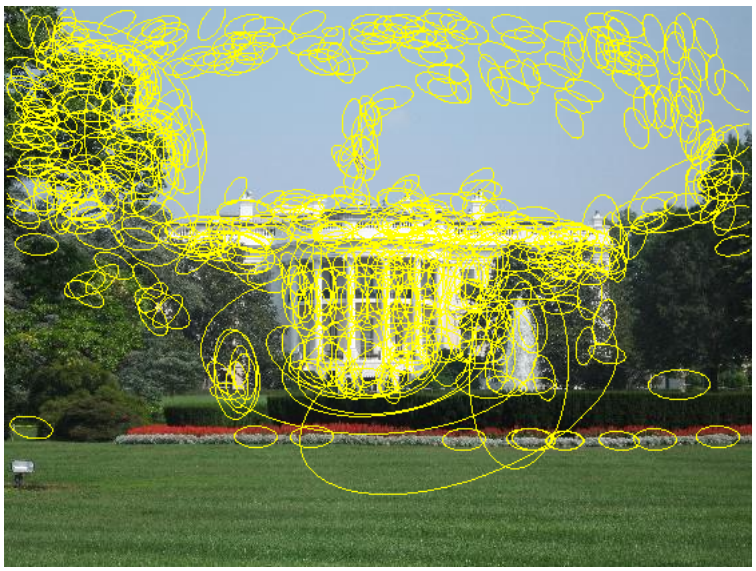

Figura 3.12: Puntos característicos de una imagen encontrados por el algoritmo SURF. Cada elipse está centrada sobre un punto relevante y tiene la orientación del mismo.

4. Arquitectura del sistema

Este capítulo pretende analizar la arquitectura de un buscador, para ello, se utilizarán las características definidas en el apartado 2.1.1 como base para el diseño de la aplicación, y se justificará el modelo elegido para el sistema.

4.1 Escalabilidad del sistema

Entendemos como escalabilidad la capacidad de un sistema para poder ejecutar un volumen mayor de trabajo añadiendo nuevos recursos, y que el coste de añadir esos nuevos recursos sea el menor posible. Esto es, si necesitamos manejar un volumen de peticiones del doble de tamaño, los nuevos recursos (memoria, tiempo de CPU, espacio en disco, etcétera) deberán ser, aproximadamente, el doble. Alcanzando un diseño que sea altamente escalable podremos ampliar el sistema sin que se vea comprometido o limitado.

Por lo anterior, la escalabilidad de la aplicación y el sistema completo debe ser una prioridad de diseño; cada día aparece una cantidad inmensa y creciente de nuevos contenidos que los usuarios comparten en Internet, por ello es imperativo que una aplicación que pretenda dar una idea de lo más representativo de la red, como es un buscador, deba tener una muy alta capacidad de adaptación y crecimiento, que permitan al sistema crecer en la misma medida en la que los usuarios lo demanden.

Para lograr esto, se ha realizado un diseño basado en dos puntos principales: balanceo de carga y encapsulación de los algoritmos de búsqueda.

4.1.1 Balanceo de carga

Para alcanzar un buen balanceo de carga, se han encapsulado los algoritmo de búsqueda en un objeto, proceso que se ampliará en el apartado 4.1.2. De este modo, puede replicarse el mismo objeto en varios servidores, permitiendo así que el cálculo pesado de la búsqueda se haga en el servidor que menos carga está soportando. Por supuesto, estos parámetros pueden ser modificados según las necesidades de la aplicación, cambiando la implementación del balanceador de carga, contenido en los servidores web que así sólo se encargarán de servir contenido estático (aunque quizá escrito en algún lenguaje dinámico).

En la figura 4.1 se muestra la figura que refleja una arquitectura completa para el sistema, en la que se tiene una gran cantidad de servidores y se consigue una alta disponibilidad, haciendo redundantes los datos que los algoritmos de búsqueda utilizan, así como replicando dichos algoritmos en granjas de servidores distribuidas geográficamente.

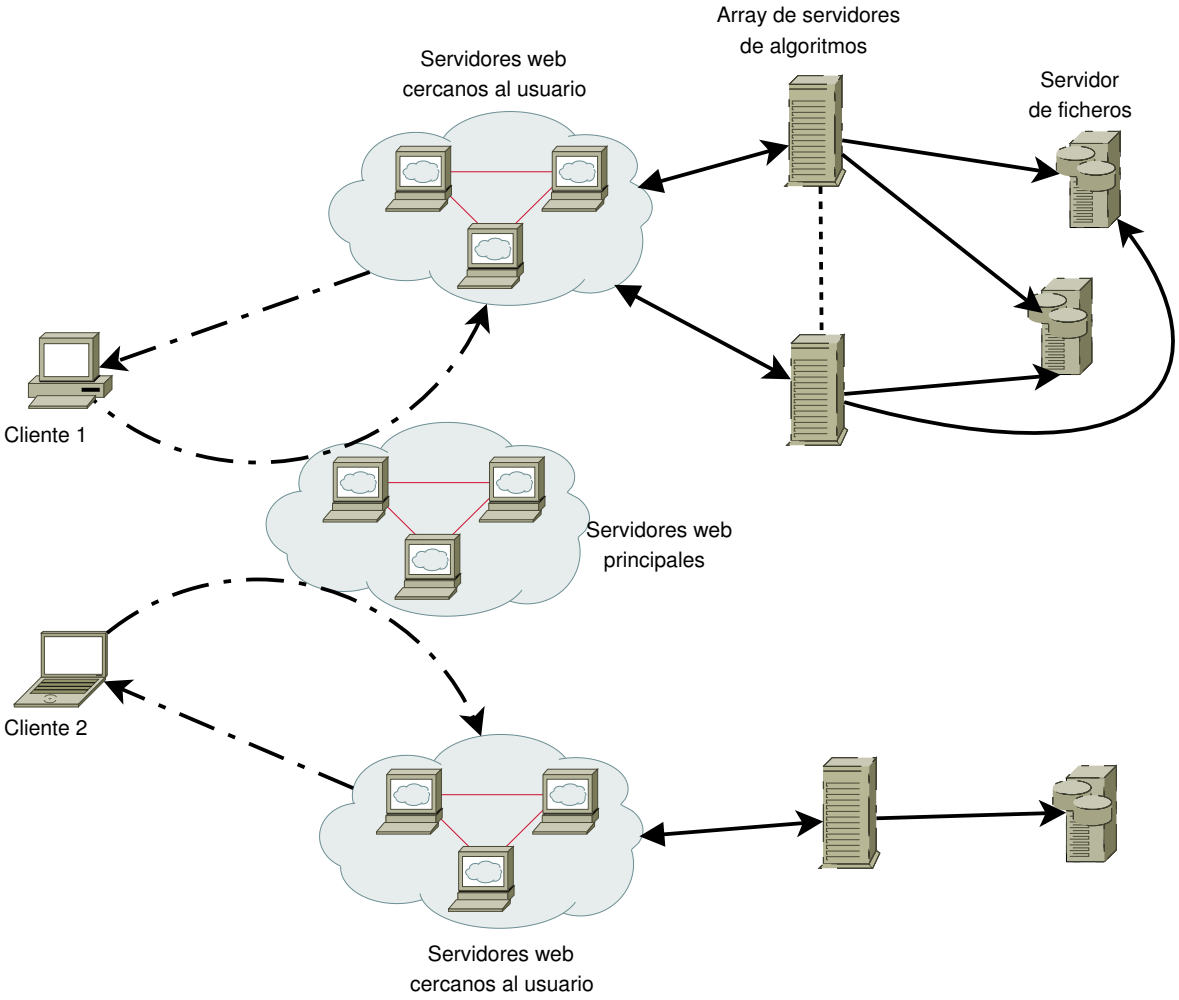

Figura 4.1: Arquitectura completa del sistema. Dos clientes envían sus consultas al buscador, son redirigidos a un *cluster* cercano, y allí se procesa su petición en un array de servidores.

4.1.2 Encapsulación de los algoritmos

Para alcanzar una mayor escalabilidad sin complicar la arquitectura, es interesante encapsular cada algoritmo de búsqueda dentro de una o varias clases (un *namespace* de C++). De este modo, añadir o modificar un nuevo algoritmo de búsqueda puede ser tan sencillo como enganchar las nuevas clases a la jerarquía ya definida y registrar sus métodos con las funciones creadas para tal fin.

Encapsulando y abstrayendo el conocimiento de este modo, podemos simplemente replicar cada *namespace* en varios servidores, obteniendo de forma sencilla un nuevo servidor de ejecución de algoritmos, lo que nos permitirá incrementar sustancialmente el rendimiento global de la aplicación, soportando una mayor cantidad de peticiones simultáneas.

Además, teniendo un modelo abstracto de algoritmo de búsqueda, es posible implementar varios algoritmos que satisfagan las diferentes características que se han definido en el apartado 2.1.1, tenido una familia de algoritmos que hagan consulta por contenido, otra por metadatos y otra que hagan una consulta combinada por

contenido y metadatos.

4.1.3 Modelo de comunicaciones

Como se puede ver en la figura 4.1, la aplicación va a hacer un uso importante de la red para comunicar los diferentes componentes, para poder alcanzar la escabilidad definida en esta sección. Aunque la cantidad de datos transmitida no va a ser importante, ni la latencia va a suponer un gran problema, pues la mayor parte del tiempo de ejecución se empleará en la búsqueda propiamente dicha, es definir un modelo de comunicaciones que permita realizar correctamente el balanceo y recuperación de la aplicación frente a caídas.

Para la comunicación entre la aplicación web y cada uno de los puntos de ejecución se ha elegido un modelo basado en llamadas remotas en XML: **XML-RPC**. Veamos las ventajas y desventajas que este método nos aporta:

- **Fiabilidad**. Como es un método antiguo, fue desarrollado en 1998 por Microsoft entre otros, las librerías que lo implementan están muy probadas y rara vez dan algún problema.

- **Portabilidad**. Al estar basado en HTTP, puede portarse el servidor a un gran número de arquitecturas.

- **Disponibilidad**. Hay librerías para prácticamente todos los lenguajes que implementan servidores y clientes XML-RPC, muchas de ellas basadas en soluciones libres como *Apache*.

- **Extensibilidad**. El servidor puede decidir en cada momento qué métodos implementa y cuales no, por lo que puede añadirse muy fácilmente un nuevo método a unos servidores de ejecución mientras que los demás siguen ejecutando la versión anterior. En este modelo, es el cliente, a través de la interfaz web, el que decide a qué método llamar de los que el servidor le ofrece.

- **Tipos de datos**. Otra de las carencias de XML-RPC es que sólo ofrece un conjunto cerrado de tipos de datos que el cliente envía al servidor, por lo que si se necesita un tipo de los que no están incluidos no se puede añadir y ha de emularse utilizando los ya dados.

- **Seguridad**. Este es el principal punto flaco del método. Al tratarse de simples llamadas codificadas en XML sobre HTTP, si un atacante sabe en qué direcciones y puertos se ejecutan nuestros servidores, podría poner en apuros a la aplicación. Sencillamente enviando peticiones XML mal codificadas, los *parsers* que incluyen las librerías pueden volverse locos intentando descifrar el mensaje y provocar un fallo del sistema.

  Además, el servidor de ejecución no dispone, a priori, de mecanismos de autenticación que le permitan identificar si el cliente realmente es el que debe ser, por lo que si no se implementa ninguna política adicional podrían darse ataques del tipo *Man-in-the-middle*, entre otros.

A pesar de que, en general, la seguridad de XML-RPC no es muy buena, para nuestra aplicación es razonable pensar que los servidores de ejecución se encontrarán en entornos muy acotados y detrás de *firewalls* que impidan ataques de denegación de servicio o cualquier otro tipo.

Por todo lo anterior, se ha elegido a XML-RPC como el protocolo de comunicación entre la interfaz web y los servidores de ejecución, pensando que las ventajas compensan con creces las carencias que pueda tener, aunque es importante no olvidarlas de cara a un futuro en el que se puedan volver problemáticas y haya que tomar la decisión de sustituir el protocolo por otro más adecuado. Aunque gracias a que el protocolo está implementado como un objeto, como parte de una jerarquía, puede sustituirse la implementación del protocolo por el que se quiera sin modificar sustancialmente la aplicación.

4.2 Velocidad de ejecución

En esta sección vamos a estudiar brevemente cómo se puede acelerar la velocidad de ejecución de las consultas, entendiendo como "velocidad de ejecución" a la rapidez con la que se ejecuta una consulta simple.

La velocidad de ejecución está muy relacionada con el concepto de balanceo de carga, pero va un paso más allá. No sólo queremos que se resuelvan muchas peticiones en poco tiempo, lo que sería una aplicación con un gran *throughput*, sino que pretendemos que cada una de las consultas que se están ejecutando tarde lo mínimo posible. Estos conceptos entran en conflicto, ya que una forma de hacer que una consulta individual se ejecute rápido sería limitar que sea la única del sistema y, por tanto, utilice todos los recursos, pero eso no es nada escalable, como vimos en la sección 4.1.

Es por esto que, apoyándonos en la encapsulación de los algoritmos, se definen a continuación unos patrones e ideas de implementación que se han utilizado o pueden utilizarse en el desarrollo de una aplicación de este estilo.

4.2.1 Lenguaje de programación

Uno de los elementos principales que se tienen en cuenta cuando se comienza a desarrollar una aplicación es el lenguaje con el que se va a programar. La elección del lenguaje a utilizar, aunque puede llegar a ser un problema, casi nunca es una solución. Por tanto, es interesante que se permita la inclusión de varios lenguajes diferentes, para que se pueda utilizar el más adecuado para cada algoritmo o método.

No es necesario utilizar lenguajes que sean compatibles a nivel de binario, ya que la aplicación puede estar distribuida en varias máquinas cada una puede estar ejecutando un lenguaje diferente y comunicarse con un protocolo de nivel superior.

Aunque nuestra aplicación se ha desarrollado íntegramente utilizando C++, gracias a lo anterior podría incluirse una máquina que ejecute una copia de los algoritmos o uno nuevo escrito en cualquier otro lenguaje, siempre que nos sea interesante por razones de eficiencia o sencillez.

4.2.2 Optimización del código

Aunque nuestro código esté escrito en un lenguaje muy eficiente para el tipo de tarea, es necesario que esté optimizado para el *hardware* real sobre el que se está ejecutando.

No sólo es importante el estilo de programación que se utilice, sino que la micro-optimización que las librerías o el compilador puedan hacer también lo son. El uso de instrucciones avanzadas del procesador (MMX, SSE, 3DNow, ...) pueden suponer una mayor ventaja en rendimiento que el uso de una mejor estructura de datos o un método de ordenación más eficiente.

Por lo anterior, se ha utilizado una combinación de la librería OpenCV que, además de ser *open-source* está muy optimizada para plataformas X86, con la librería Intel IPP que está muy ajustada para la ejecución en procesadores de la marca Intel.

4.2.3 Uso de aceleración GPU

Una técnica que cobra cada vez más importancia en las aplicaciones científicas y de cálculo intensivo es el uso de tarjetas aceleradoras de vídeo como núcleos de ejecución de operaciones en punto flotante. Aunque en nuestra aplicación no se ha utilizado esto, puede ser muy interesante para un futuro en el que se desee exprimir más las posibilidades de cada servidor de ejecución.

El uso de una tarjeta gráfica (GPU) en la aceleración de cálculos científicos no es nuevo, pero está comenzado a tomarse muy en serio debido a la introducción de soporte nativo en las soluciones comerciales, sobre todo de la tecnología CUDA de nVidia. Al fin y al cabo, una tarjeta gráfica no es más que un gran procesador diseñado para ejecutar un inmenso número de operaciones en coma flotante por segundo, por lo que la idea de pasarle algún trabajo parece muy sensata.

Como se ha dicho, gracias al soporte nativo esto se facilita enormemente esta tarea, cuando antes se hacía transformando los datos a una matriz que se pareciera a una imagen de vídeo y aplicando ciertos efectos de vídeo sobre la misma, de modo que se realizaran los cálculos que deseábamos hacer. Como dato, una nVidia GeForce GTX 295 supera los 1700 GFLOPs (miles de millones de operaciones en coma flotante por segundo)[1]; mientras que un procesador por el mismo precio apenas alcanza los 50 GFLOPs[2].

En la figura 4.2, extraída de [9], puede verse una comparativa del tiempo necesario para ejecutar varios algoritmos en una sola CPU, utilizando ocho núcleos, y usando una GPU para acelerar los cálculos. El incremento de rendimiento cuando se utilizan ocho núcleos de procesamiento es de, aproximadamente un 200 %, reduciendo así el tiempo necesario para ejecutarse a un tercio; mientras que si se hacen los cálculos en el procesador gráfico, consigue reducirse el tiempo a un sexto, aproximadamente el doble que con la CPU multiprogramada.

Es interesante ver cómo la implementación sobre una GPU ha superado a un procesador de ocho núcleos y es que, además de conseguir una gran mejora en

[1] http://en.wikipedia.org/wiki/GeForce_200_Series
[2] http://www.intel.com/support/processors/sb/cs-023143.htm

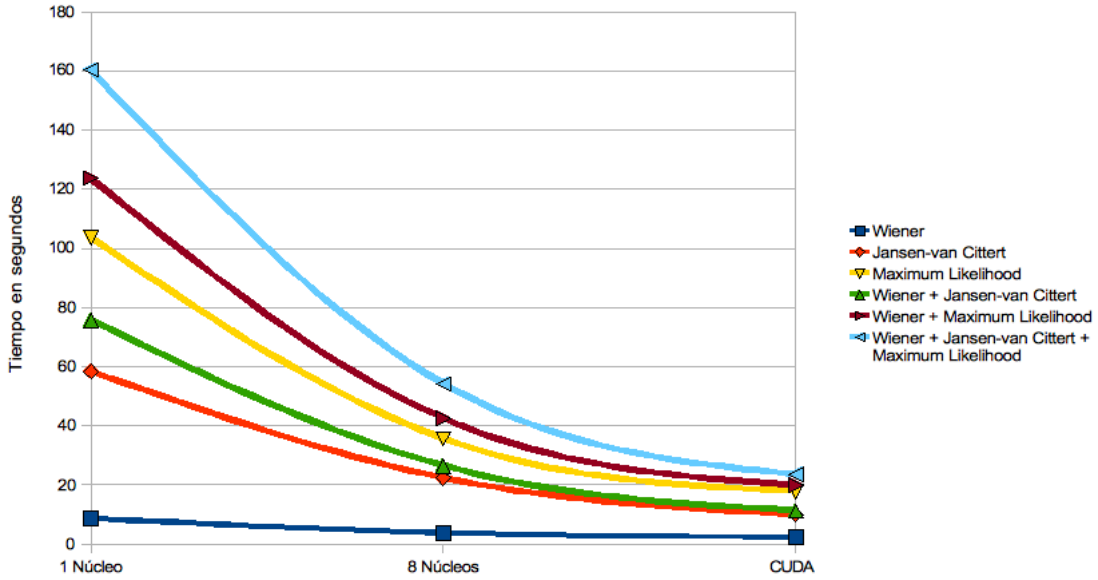

Figura 4.2: Comparativa de rendimiento en la deconvolución de una imagen, implementada sobre una CPU, multiprogramada sobre ocho núcleos y en una GPU mediante CUDA.

tiempo, destaca que el precio de un procesador con esa cantidad de núcleos de procesamiento suele superar al de una aceleradora de vídeo, por lo que se puede conseguir un aumento de rendimiento importante con una pequeña inversión.

Para acabar este apartado, se destacan un par de resultados de implementación sobre CPU y GPU. En primer lugar, se presentan los resultados de una implementación en CUDA de SIFT sobre GPU, disponible en [10]. En esta implementación se observa una mejoría cuando se utiliza una nVidia 8800GTX, sobre todo con tamaños de imagen pequeños, como puede verse en la figura 4.3. Aunque es de suponer que la implementación aún puede ser mejorada, comentan en el artículo que se ha probado sobre una mejor tarjeta gráfica, en concreto una GTX280, y los resultados son notablemente mejores, por lo que se comprueba el camino de futuro que esta tecnología tiene.

En la tabla 4.1, extraída de [11], se puede apreciar mejor el efecto de la GPU sobre la implementación de SURF. En ella se ha conseguido reducir a aproximadamente un tercio el tiempo total de cálculo. Es especialmente relevante el detalle de que la mayor parte de esta mejora proviene del cálculo del descriptor SURF, que ocupaba una gran parte del tiempo de CPU, mientras que en GPU se consigue minimizar este tiempo. Siguiendo la ley de Amdahl, disponible en [12], los creadores han conseguido una gran mejora en ese cálculo reduciendo considerablemente el tiempo total. Ahora sería necesario estudiar también si el resto de áreas son mejorables, ya que hay algunas en las que apenas se obtiene mejoría.

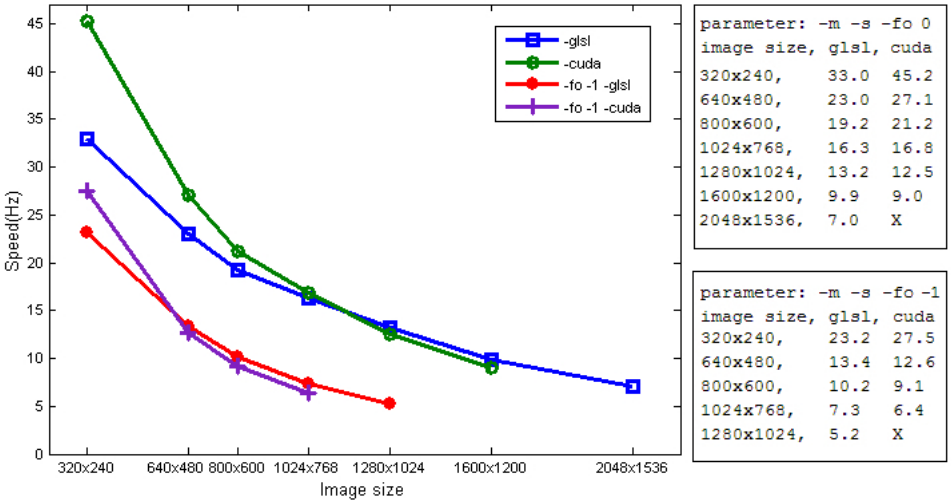

Figura 4.3: Comparativa de rendimiento para el algoritmo SIFT implementado en una CPU y utilizando CUDA sobre una tarjeta gráfica nVidia 8800GTX.

Función	GPU (ms)	CPU (ms)
Integral	16,0000	13,0000
Determinante del Hessiano	13,6800	39,0000
Descriptor SURF	6,1850	128,0000
Orientación	27,0000	27,0000
Total	62,8650	207,0000

Tabla 4.1: Tiempo empleado por varias rutinas del cálculo de un descriptor SURF en la implementación OpenSURF, se muestran los tiempos empleados si se utiliza la CPU o la GPU para cada procedimiento.

4.3 Arquitectura de la aplicación

En la figura 4.4 se muestra un esquema general de los paquetes (o *namespaces* de C++) en los que se ha dividido el diseño de la aplicación. En primer lugar cabe destacar que algunos paquetes tienen paquetes anidados, como por ejemplo el de algoritmos de búsqueda "search". Como se vio en el apartado 4.1.2 esto se ha hecho para mejorar la extensibilidad de la aplicación, permitiendo añadir nuevos métodos de búsqueda de forma sencilla.

De este modo, implementar un nuevo algoritmo de clasificación es tan simple como añadirlo al namespace correspondiente y hacer que la clase lo utilice o, simplemente, copiar el namespace en uno nuevo y que contenga la nueva funcionalidad.

Además, se separa la parte encargada del envío y recepción XMLRPC del resto, por lo que la consistencia interna y coherencia de módulos es mucho mejor y podría cambiarse este método por cualquier otro sin mayor problema.

Finalmente, los paquetes "application" y "data" contendrán los métodos propios de la aplicación y el acceso a bases de datos y/o ficheros, respectivamente. En esta versión de la aplicación, se utilizan ficheros, pero podría fácilmente cambiarse por una base de datos SQL o cualquier otro tipo, como ya se ha explicado.

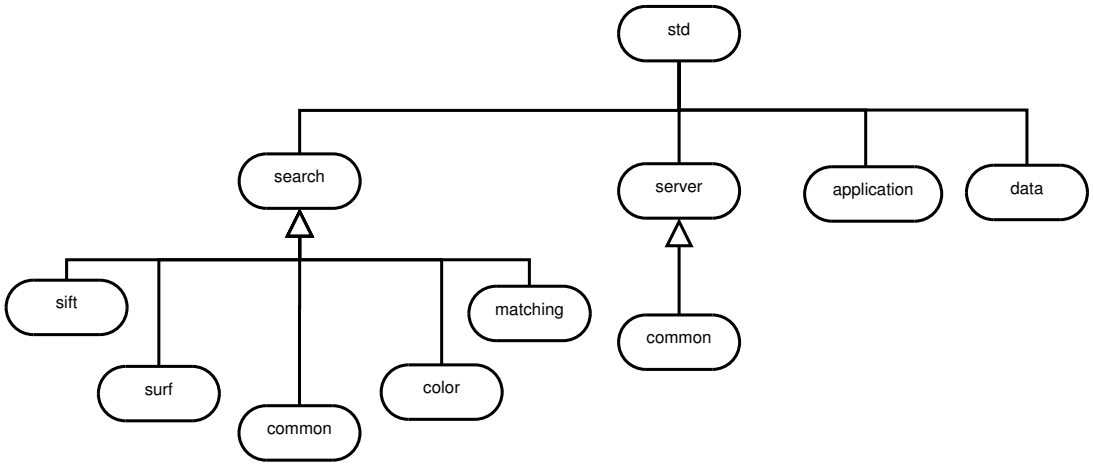

Figura 4.4: Dibujo de la arquitectura de paquetes de la aplicación. Se muestran los *namespaces* que se han implementado en C++.

5. Métodos de búsqueda

En esta sección se presentan y analizan los métodos de clasificación de imágenes que se han implementado en el proyecto, así como los clasificadores utilizados para optimizar y acelerar las búsquedas de un elemento en la base de datos.

5.1 Comparación píxel a píxel de dos imágenes

En este método, el objeto asociado a cada elemento es la misma imagen reescalada, en cualquier espacio de color.

Figura 5.1: Resultado de restar píxel a píxel dos imágenes con buena resolución. De izquierda a derecha: minuendo, sustraendo y resultado.

Figura 5.2: Resultado de restar píxel a píxel dos imágenes con poca resolución. De izquierda a derecha: minuendo, sustraendo y resultado.

El tamaño de la imagen reescalada es un factor importante: un tamaño pequeño hará que la gran parte de las imágenes tengan un alto parecido, como puede verse en la figura 5.2. Por el contrario, un tamaño grande de escalado ampliará las diferencias entre elementos, como se ve en 5.1. Por tanto, las imágenes menos parecidas tendrán una mayor diferencia al comparar.

Respecto al espacio de color, nos interesa uno que separe la crominancia de la luminancia, de modo que podamos comparar píxeles que son similares pero que tienen una mayor o menor iluminación. En la práctica, esto es muy útil debido a los cambios de luz que las cámaras y, sobre todo, videocámaras sufren con el tiempo.

Esto es debido a que los cambios de luz son efectos indeseables para la visión. Por su existencia, dos fotografías tomadas al mismo objeto con unos segundos de diferencia pueden ser diferentes por cuestiones tan sutiles como, por ejemplo, la climatología.

Según la aplicación, puede que nos interese utilizar un tamaño grande o pequeño, o un espacio de color u otro, todo dependerá del tipo de resultados que deseemos.

5.2 Histogramas de color

Un histograma consiste en un resumen de la información de color de la imagen, de modo que se represente la cantidad de cada color que contiene. Esto es muy utilizado para retoque fotográfico, donde se calcula el histograma como los de la figura 5.3 y se aplica algún efecto sobre la foto para que su histograma en cierto espacio de color tenga las propiedades deseadas.

Un histograma, para una imagen con un sólo canal de color, se compone de un vector que contiene la cantidad de píxeles que toman un cierto valor. Como la cantidad de valores distintos que un píxel puede tomar puede ser arbitrariamente grande, se suele utilizar un sistema de "cubetas". Estas cubetas contendrán la suma para un rango de valores de píxel. Por ejemplo, si nuestra información de color toma valores en el rango $[0, 255]$, se podrían utilizar 16 cubetas, de modo que la primera contenga el número de píxeles que toman un valor en el intervalo $[0, 15]$, la siguiente en $[16, 31]$ y así sucesivamente.

El cálculo de un histograma unidimensional es sencillo, se crea un vector con tantas posiciones como cubetas se quieran tomar; y a continuación, se cuentan los píxeles que caen en dicha cubeta. El resultado de hacer este proceso para cada canal de una imagen en RGB puede observarse en la figura 5.3.

Figura 5.3: Histogramas RGB unidimensionales a la derecha, imagen de entrada a la izquierda..

Otra forma de hacer histogramas para imágenes con varios canales de información, es hacer lo que se llama **histograma multidimensional**, consistente en hacer una matriz con tantas dimensiones como canales vayamos a combinar. Por ejemplo, para una imagen RGB podría hacerse un histograma tridimensional para los tres

canales simultáneamente como los de la figura 5.4, o podrían hacerse tres histogramas bidimensionales para las combinaciones RG, RB y GB, como los mostrados en la figura 5.5; todos ellos para la misma imagen de entrada mostrada en la figura 5.3.

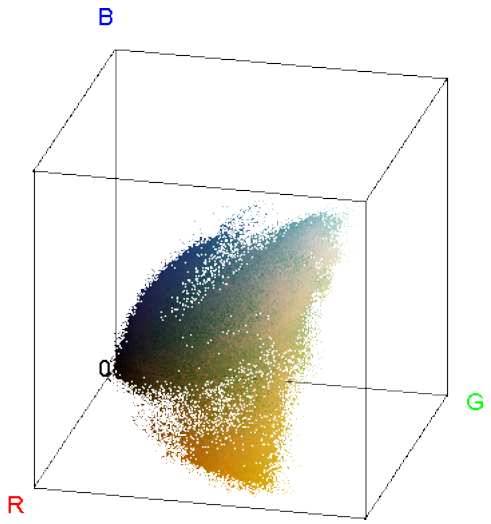

Figura 5.4: Histograma tridimensional en RGB calculado sobre la imagen de la figura 5.3.

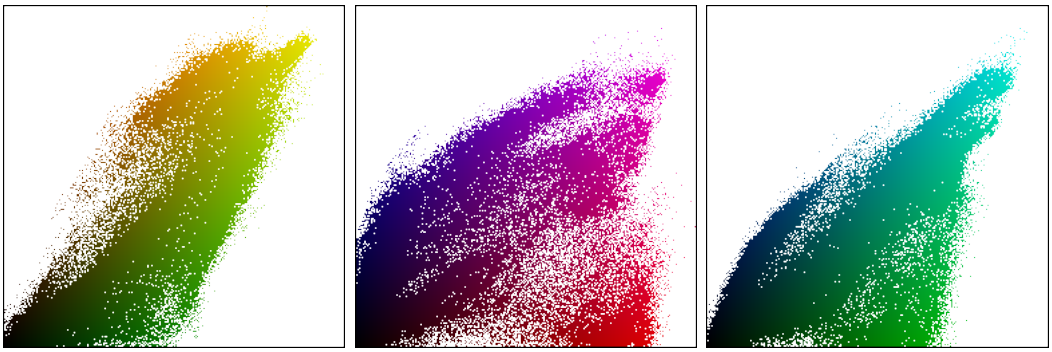

Figura 5.5: Histogramas bidimensionales de la imagen de la figura 5.3. De izquierda a derecha, histogramas para las combinaciones de canales: RG, RB y GB.

Como se verá en la sección 6.2, además de decidir la dimensionalidad del histograma, también es importante elegir bien tanto el número de cubetas como el espacio de color en el que se va a trabajar. Un menor número de cubetas hará que muchas imágenes tengan un gran parecido; mientras que un gran número de estas hará que sólo las imágenes con colores muy similares sean reconocidas.

Por otro lado, para el espacio de color hay numerosas posibilidades, cada una con sus defectos y sus virtudes. A modo de ejemplo, si se utiliza un espacio de color que separe la crominancia de la información de luminosidad, como YUV o HSV, se puede dar un menor peso a la luminosidad, para tener una cierta invarianza a los cambios de luz en las fotos, aunque a cambio puede estar perdiéndose información por la propia naturaleza de las cámaras, en la que una pequeña variación de luz puede dar lugar a colores muy diferentes, sobre todo en los extremos de los intervalos de representación.

Finalmente, según las pruebas que se estudian en el artículo [13], es un método sencillo pero muy efectivo para este tipo de tareas.

5.3 Características SIFT

SIFT es el acrónimo de Scale-Invariant Feature Transform, publicado en 1999 por David Lowe en el artículo referenciado en [14], define un método para encontrar características relevantes en una imagen proporcionando cierta invarianza a la escala, rotación y a una pequeña variación en la pose 3D.

El algoritmo de extracción de características (*features*) SIFT se basa en la idea de "espacio de escala". El **espacio de escala** de una imagen es una pirámide formada por todos los posibles reescalados de la imagen; y supone que, dada una *feature*, la respuesta del detector será máxima en un reescalado de entre todos los posibles, mientras que en los adyacentes en la pirámide se tendrá una menor respuesta.

Como obtener la pirámide completa sería un trabajo computacionalmente impracticable, se reduce el cálculo a sólo unos cuantos niveles discretizados, calculados como la convolución de la imagen original con un operador de suavizado Gaussiano, doblando en cada paso el nivel de suavizado, consiguiendo así un escalado efectivo del 50 %. Como se puede ver, uno de los principales problemas de este método es su lentitud, pues se han de realizar numerosos cálculos para encontrar la localización de los puntos importantes de la imagen, aunque hay implementaciones muy rápidas como se comenta en el apartado 4.2.3.

Así, el algoritmo se compone de cuatro pasos principales, bien resumidos en [15]:

1. **Cálculo del espacio de escala**. En primer lugar, se construye el espacio de escala aplicando una serie de suavizados Gaussianos sobre la imagen original, y se agrupan por "octavas". Cada octava corresponde a reducir el tamaño de la imagen a la mitad o, en este caso, a doblar el radio del suavizado. Tras esto, se restan las imágenes adyacentes de cada escala obteniendo una diferencia de Gaussianas (DoG), como puede verse en la figura 5.6.

2. **Localización de los puntos clave**. Cada posible punto de una *feature* es encontrado como el punto de máxima respuesta (máximo o mínimo) de la diferencia de Gaussianas, comparándolo con sus vecinos en la imagen y con los puntos vecinos de las imágenes adyacentes del espacio de escala, como se muestra en la figura 5.7.

   A continuación, es necesario filtrar los puntos para eliminar los puntos de bajo contraste y los que se encuentran en bordes, ya que la diferencia de Gaussianas produce una fuerte respuesta en torno a los bordes de la imagen. El resultado de este proceso puede verse en la figura 5.8.

3. **Asignación de la orientación**. Para dotar a los *keypoints* de independencia a la rotación, se aplica un post-proceso, consistente en calcular la mejor aproximación del vector gradiente a una de las direcciones discretizadas. Así, se compara la magnitud y dirección de dicho vector con unas cuentas direcciones discretizadas, habitualmente 36 direcciones con 10 grados de amplitud

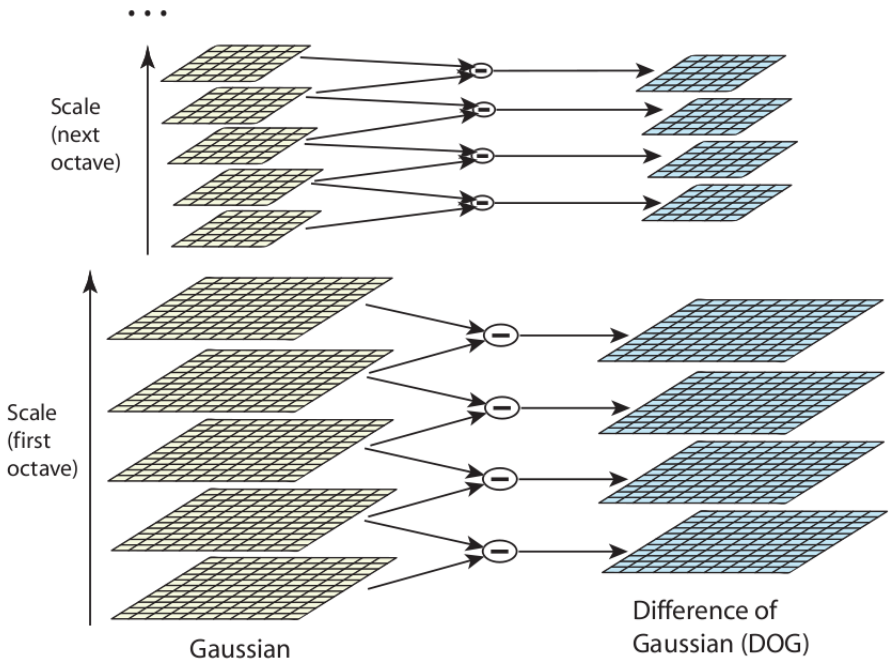

Figura 5.6: Combinación de imágenes suavizadas con suavizado Gaussiano para obtener la diferencia de Gaussianas en el algoritmo SIFT. Arriba, una octava menor, es decir, aplicando un mayor suavizado que abajo.

cada una, y se asigna la orientación (u orientaciones) que tienen una respuesta máxima.

4. **Cálculo del descriptor SIFT**. Ahora, para cada *keypoint* localizado, se calcula un descriptor que lo identifique unívocamente y que sea invariante a la posición, escala y rotación y a otras características indeseables, como la iluminación. Para ello, se crean histogramas de la orientación de los vectores gradiente cerca del punto, en regiones de $4x4$ píxeles, con 8 cubetas cada histograma. Esto se hace en $4x4 = 16$ histogramas alrededor del *keypoint*, teniendo en total 128 datos que son los que formarán el vector descriptor.

   Para conseguir invarianza a la iluminación, cada histograma es normalizado a la unidad, se le aplica un umbral de 0,2 y se vuelve a normalizar. En la figura 5.9 puede verse un ejemplo de este proceso, con $2x2 = 4$ histogramas alrededor del *keypoint*.

En la sección 6.3 se hace un análisis de parámetros para esta técnica en una aplicación de tipo buscador, y se propone utiliza el clasificador descrito en 5.5 para mejorar su rendimiento.

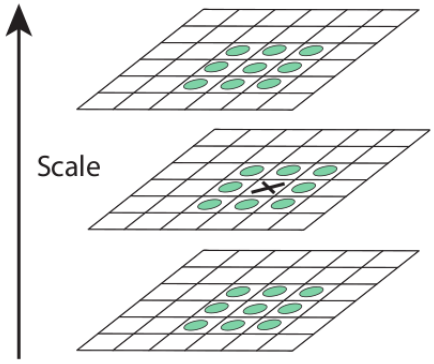

Figura 5.7: El punto marcado con la X se compara con cada uno de sus vecinos, marcados con un círculo, buscando el punto en el que la diferencia de Gaussianas es máxima, en un paso intermedio del algoritmo SIFT.

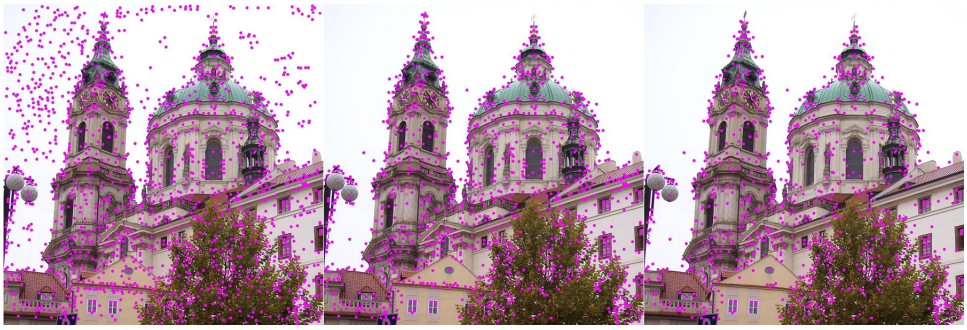

Figura 5.8: Puntos clave encontrados por el algoritmo SIFT en varias etapas del algoritmo. De izquierda a derecha: todos los puntos detectados, tras eliminar los de bajo contraste y eliminando también los que están en un borde.

5.4 Características SURF

En la actualidad, otras técnicas basadas en *features* están siendo estudiadas, como por ejemplo SURF (Speeded Up Robust Features), que ha demostrado tener una eficacia similar a SIFT, al tiempo que se reduce sustancialmente la carga computacional.

SURF es una técnica propuesta por Herbert Bay en 2006 en el artículo [16] inspirada en SIFT, pero que utiliza *wavelets* y una aproximación del determinante del Hessiano para conseguir un mejor rendimiento. Además, al contrario que SIFT, está libre de patentes y hay numerosas implementaciones muy eficientes, incluso multiprogramadas y en GPU, como la mostrada en el apartado 4.2.3; por ello se convierte en una opción muy interesante de cara al rendimiento de nuestra aplicación, y en la sección 6.4 se hace un estudio del uso de esta técnica.

En la figura 5.10 se muestra una comparación de los resultados que se obtienen aplicando SIFT y SURF sobre una misma imagen que, como se ve, son muy distintos.

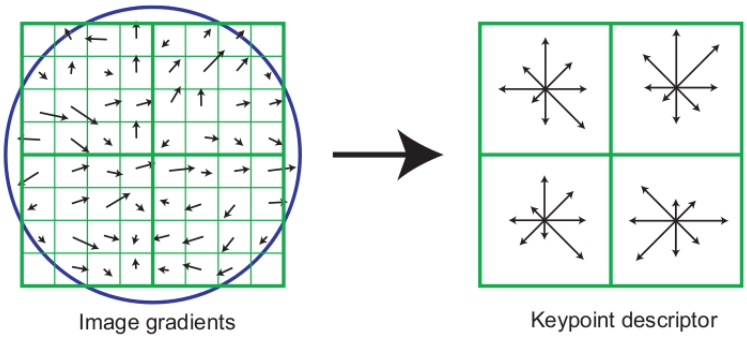

Figura 5.9: Los gradientes en torno a un punto son usados por SIFT para calcular el descriptor como un histograma normalizado de gradientes.

Figura 5.10: Resultado de aplicar los algoritmos SIFT y SURF sobre una misma imagen.

5.5 Clasificador para SIFT y SURF

En el estudio de características se ha considerado en primer lugar un algoritmo de clasificación en el que las características de la imagen de entrada se comparan con todas las almacenadas en la base de datos. Esto ha resultado en un enorme tiempo de ejecución, del orden de varios minutos por cada búsqueda, por lo que es necesario definir una estructura más compleja que permita un mejor acceso.

Se ha observado que el mayor problema ha sido el gran número de características que se extraen de cada imagen, aumentando exponencialmente el tiempo de búsqueda. Por ello, se ha hecho un ajuste de parámetros, para reducir el número y aumentar, en la medida de lo posible, las *features* obtenidas de cada imagen de la base de datos. Esto ha resultado en un tiempo de ejecución de en torno a 8 segundos por búsqueda, siendo el proceso de comparación el que mayor peso tiene sobre éste.

Para tratar de reducir este tiempo se ha optado por implementar un árbol de vocabulario, similar al descrito en [17], [18] y [19].

Este tipo de clasificador consiste en un árbol n-ario, en el que cada nodo intermedio contiene una *feature*, que representa a todas las que están contenidas en sus nodos hijos. La creación consta de varios pasos:

1. **Segmentación de las *features***. utilizando el algoritmo de las k-medias (una descripción detallada puede consultarse en [20]), donde el número "k" es el número de hijos que cada nodo del árbol tendrá.

2. **Creación de nodos hijos**. Cada nodo hijo es creado como uno de los subconjuntos de las k-medias, y su representante será el centroide de dicho subconjunto.

3. **Nodos solapado o puros**. Un nodo puede ser solapado, si alguno de sus hijos contiene *features* repetidas o, por el contrario, puede ser puro si ninguno de todos los subconjuntos de los nodos hijos son disjuntos. En principio, cualquier nodo es solapado, debido al funcionamiento del k-medias.

4. **Criterio de parada**. El criterio de parada para la creación del árbol es una cantidad mínima de *features* en un nodo hoja. Así, cuando un nodo tenga un número igual o menor al mínimo, no se continúa expandiendo.

5. **Poda**. Una vez creado cada nodo, se puede decidir la poda del mismo, si el índice de solapamiento es superior a un cierto valor (habitualmente 0.7). En este caso, el nodo se elimina y se vuelve a crear forzando a que ninguna *feature* esté repetida en los hijos. Esto se hace para evitar que, en los primeros niveles del árbol en los que habrá muchos subconjuntos solapados, se tenga que hacer un *backtracking*, descartando rápidamente un gran número de *features*.

6. Experimentación y resultados

En esta sección se a realizar un análisis de la influencia de los parámetros en el funcionamiento de cada uno de los métodos utilizados en nuestro proyecto, así como una comparativa global de los resultados obtenidos en cada uno de los métodos para diferentes galerías de prueba.

Antes de comenzar con el análisis de datos, conviene establecer una forma de comparar los resultados que se obtengan de la ejecución de los métodos. Supongamos que se realiza la consulta de una imagen sobre una base de datos, produciéndose varias imágenes como resultado. En una aplicación de esta clase, hay dos tipos de errores que nos interesa diferenciar:

1. **Falsos positivos**. Se trata de resultados que han sido devueltos incorrectamente, esto es, que se han reconocido como pertenecientes a la categoría consultada pero que no lo son.

2. **Falsos negativos**. Son imágenes que no son reconocidas como pertenecientes a la categoría consultada, o lo son con un bajo nivel de aceptación, cuando en realidad deberían ser aceptadas.

Para representar la variación de estos ratios para diferentes ajustes de las técnicas de comparación, usaremos unas gráficas en las que se muestran las curvas ROC (Receiver Operating Characteristic) que representan la tasa de falsos positivos frente a la tasa de falsos negativos. En estas curvas, se ha de maximizar el área que cada curva deja por encima, minimizando así el error global cometido. También es interesante conocer en qué punto la curva intersecciona con la recta $f(x) = x$. Esta intersección significa que, para ese ajuste de parámetros concreto, el número de falsos positivos y negativos se iguala, lo que puede ser interesante. Se utilizará como medida de bondad alternativa el área sobre la curva ROC.

6.1 Comparación píxel a píxel de dos imágenes

Para estas pruebas se ha utilizado la base de datos sencilla descrita en el apartado 2.3. La razón de esta elección es que, debido al pequeño tamaño de la muestra (100 imágenes), puede hacerse un estudio más amplio de todas las posibles combinaciones de los parámetros. En este caso, se ha probado con cinco espacios de color: Gris, YCrCb, HLS, HSV y RGB; y con tamaños de imagen reescalada entre 1x1 y 128x128.

En la figura 6.1 se muestra el resultado de estas pruebas, en las que se observa que, como era predecible, los tamaños pequeños de imagen suelen funcionar mejor para este tipo de tarea. Lo sorprendente es que el área máxima global, es decir el mejor resultado, se obtiene en el espacio de color RGB y que, además, en dicho espacio la diferencia entre usar unos tamaños y otros es pequeña, lo que indica que

el *matching* en este espacio de color puede ser una buena opción. El mejor resultado, con un EER del 13,3 %, se consigue con en el espacio RGB con tamaños de $52x52$ píxeles, seguido por el HSV, con un 16,3 %. Como era de esperar, los mayores errores se cometen analizando las imágenes en gris, con un EER del 19,7 %. Otra ventaja del espacio RGB es que el tamaño de las imágenes no presenta una gran influencia en la eficacia del método. Cualquier tamaño entre $4x4$ y $100x100$ presenta curvas muy parecidas.

Los anteriores resultados se han contrastado con otro *benchmark* más complejo, en el que se elimine la posibilidad de aciertos del método por la sencillez del problema. Las gráficas de este nuevo test están en la figura 6.2. Los resultados son notablemente peores que los de la figura 6.1, pero se mantiene que RGB es el mejor método de entre los analizados, con un EER del 34,6 % para un tamaño excepcionalmente grande, $461x461$; seguido del test en gris, con un EER del 37,6 % para un tamaño de $142x142$. Esos tamaños tan grandes para el mejor resultado se deben a que los resultados son muy similares entre unos tamaños y otros, al igual que pasaba con el *benchmark* sencillo.

En la figura 6.3 se muestra la gráfica de los tiempos que 100 búsquedas han tardado para cada uno de los tamaños y espacios de color. Estos tiempos incluyen la lectura de la imagen de entrada, su reescalado y la consulta en la base de datos. Es significativo que el tiempo se mantiene prácticamente constante independientemente del tamaño de las imágenes. Esto se debe a la optimización de la librería gráfica, y a que la mayor parte del tiempo de ejecución se emplea en otras tareas fuera de la propia comparación entre imágenes. Además, puede observarse cómo el espacio de color influye en el tiempo total, por lo que se deduce que una gran parte del tiempo de búsqueda es empleado en el cambio de espacio de color. Como era de esperar, la mayor eficiencia se obtiene usando el gris, que es capaz de realizar 100 búsquedas en menos de 1,5 segundos.

En la figura 6.4 se muestran los dos primeros resultados devueltos por el método de *matching*, ajustado con la mejor configuración encontrada, sobre la base de datos realista de 10200 imágenes cuando se le pasa una foto que no tiene almacenada.

6.2 Histogramas de color

Un histograma de color viene definido por dos parámetros principales: el número de dimensiones que tiene (derivado del espacio de color que se utilice), y el número de cubetas que se usan para cada dimensión.

A mayor número de cubetas, el tiempo de búsqueda y comparación será mayor, pero esto dependerá también de la dimensionalidad del histograma, puesto que hace aumentar proporcionalmente el número de operaciones simples que han de realizarse. Para estas pruebas, se ha utilizado la base de datos sencilla de la sección 2.3, y se ha ejecutado con la siguiente configuración:

1. **Espacios de color**. Se ha probado para los siguiente espacios de color: gris, HLS, HSV, RGB, YCrCb y XYZ. En los espacios que incluyen canales de luminosidad, ésta valdrá un 20 % y la crominancia el resto (un 40 % cada canal).

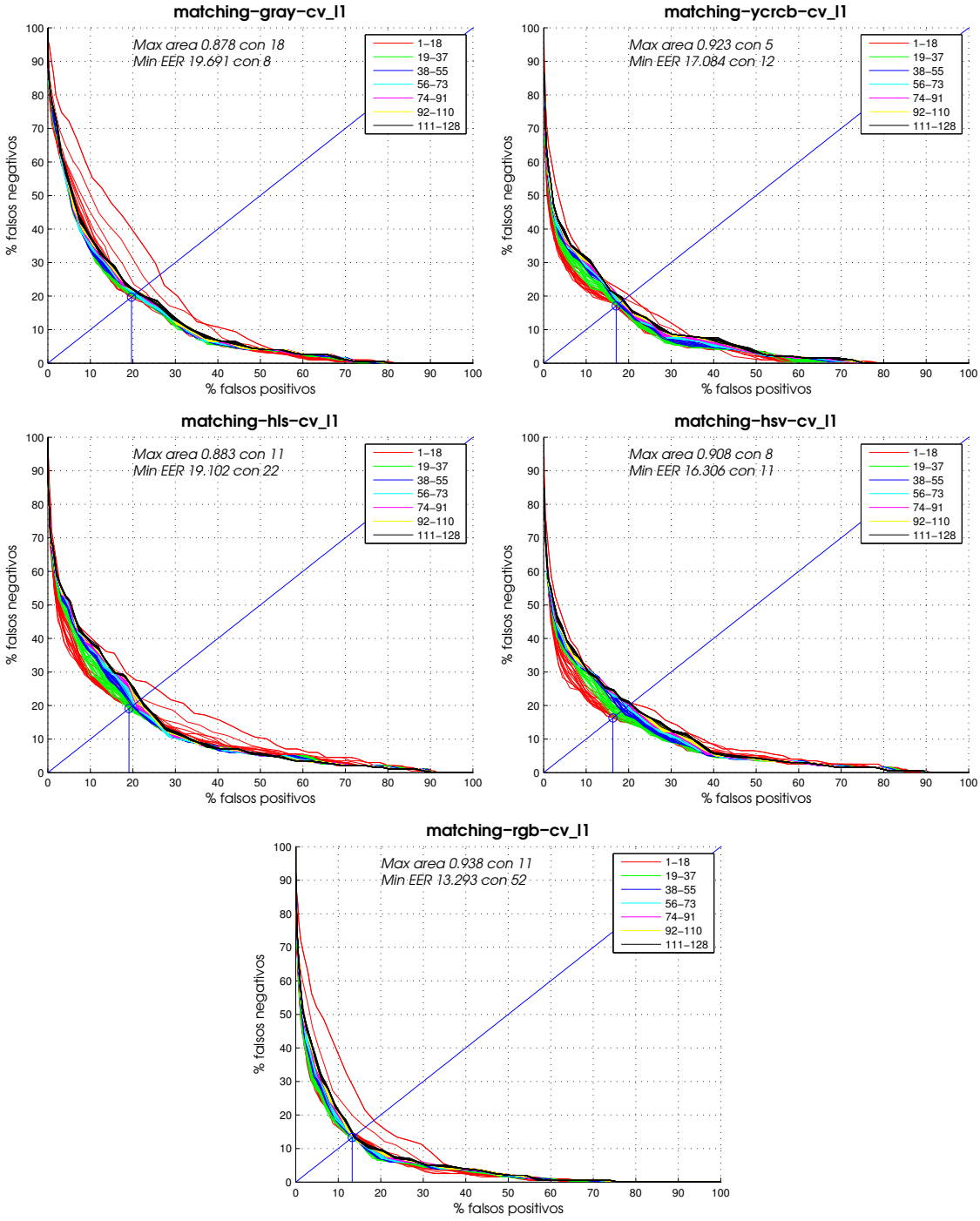

Figura 6.1: Resultados de la búsqueda por *matching* en la galería sencilla de 100 imágenes. De arriba abajo, de izquierda a derecha: gris, YCrCb, HLS, HSV y RGB. Para cada uno, se prueban tamaños de imagen entre 1 y 128. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

2. **Dimensionalidad**. Para cada uno de los espacios de color, se ha probado usando varios histogramas de una dimensión y, además, usando un histograma multidimensional. En los espacios de color que incluyen un canal para la luminosidad, éste se ha separado en un histograma unidimensional para dar

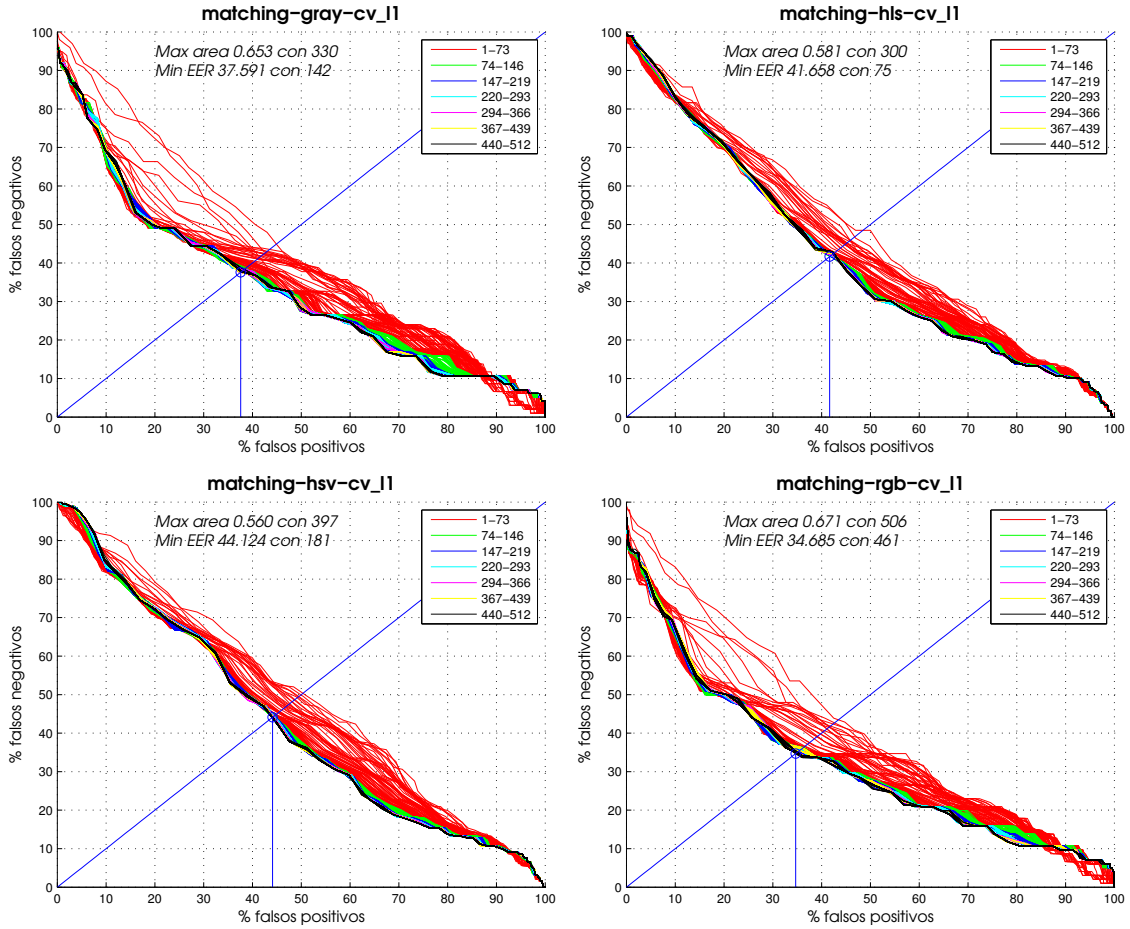

Figura 6.2: Resultados de la búsqueda por *matching* en la galería complicada de 50 imágenes. De arriba abajo, de izquierda a derecha: gris, HLS, HSV y RGB. Para cada uno, se prueban tamaños de imagen entre 1 y 512. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

un menor peso en la comparación a la luminosidad, y que se favorezcan las similitudes en la crominancia.

3. **Tamaños**. Para cada una de las configuraciones anteriores, se ha ejecutado con tamaños de histograma entre 1 y 128.

Como podemos observar en la figuras 6.5 y 6.6, los mejores resultados se han obtenido en los espacios de color HLS y HSV, con unos valores respectivos del EER del 19,1 % y el 19,6 % en una dimensión. Aunque el resultado en RGB multidimensional es ligeramente superior al resto, no se puede decir que exista una diferencia lo suficientemente significativa como para decantarse por uno u otro, por lo que se necesitarían más datos para poder extraer mejores conclusiones.

Para tomar una decisión con mejores datos, se ha querido comprobar que los resultados son fiables, pasando un *benchmark* con una base de datos de imágenes más compleja. Los resultados nuevos se muestran en la figura 6.7, y se ha encontrado que, para una base de datos compleja los espacios de color que separan la luminancia mejoran a RGB, por lo que se comprueba la hipótesis inicial de que, en un espacio

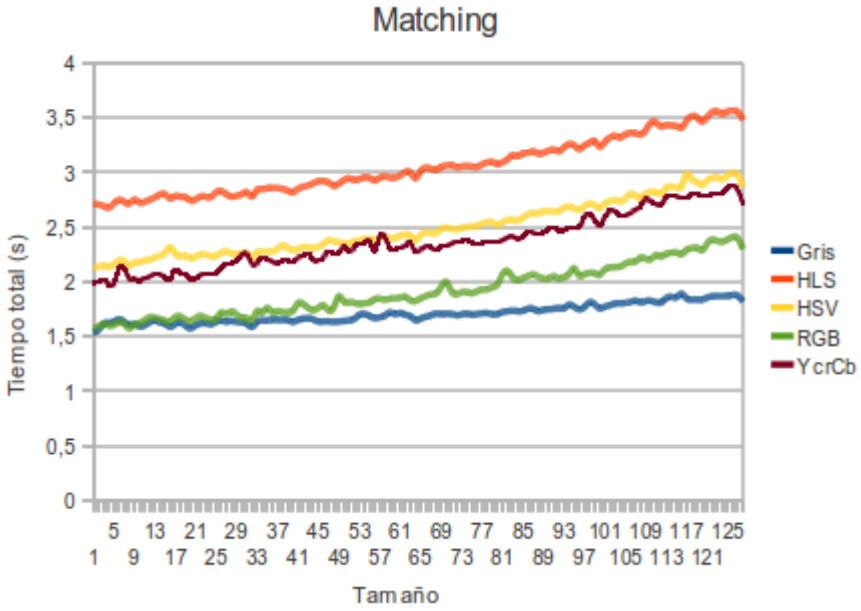

Figura 6.3: Tiempo para ejecutar 100 búsquedas con *matching* usando diferentes espacios de color y tamaños de las imágenes. Ejecución en un Intel Core 2 Duo a 2.1 Ghz.

Figura 6.4: Resultados de la búsqueda por *matching* en la galería realista de 10200 imágenes. De izquierda a derecha: imagen de entrada, y dos primeros resultados devueltos.

de búsqueda suficientemente complicado, separar la luminosidad de la crominancia es conveniente para descartar las variaciones de luz entre unas imágenes y otras, y así hacer un mejor emparejado de cada imagen. El mejor espacio de color ha resultado ser HLS, con un EER del 30,3 %, seguido de HSV con el 31 %, mientras que RGB no ha conseguido bajar del 32 %.

En la figura 6.8 se muestran las gráficas de tiempos para 100 búsquedas en cada espacio de color y con los diferentes tamaños de histograma probados, para histogramas uni y multidimensionales. Estos tiempos incluyen la lectura y procesamiento de la imagen consultada y la búsqueda en la base de datos. Se puede observar cómo los tiempos son ligeramente superiores al empleado en matching (figura 6.3), aunque sigue siendo aceptable para la mayoría de casos. Además, es interesante el hecho de que el tiempo sólo se dispara aumentando el tamaño de histograma en histogramas 3D (espacios de color RGB y XYZ), mientras que en histogramas 2D apenas hay diferencia entre un tamaño grande y uno pequeño. Por lo demás, el tiempo empleado en la búsqueda es muy similar al caso de matching por lo que, como se ha dicho anteriormente, es de suponer que la comparación de histogramas en sí está ocupando

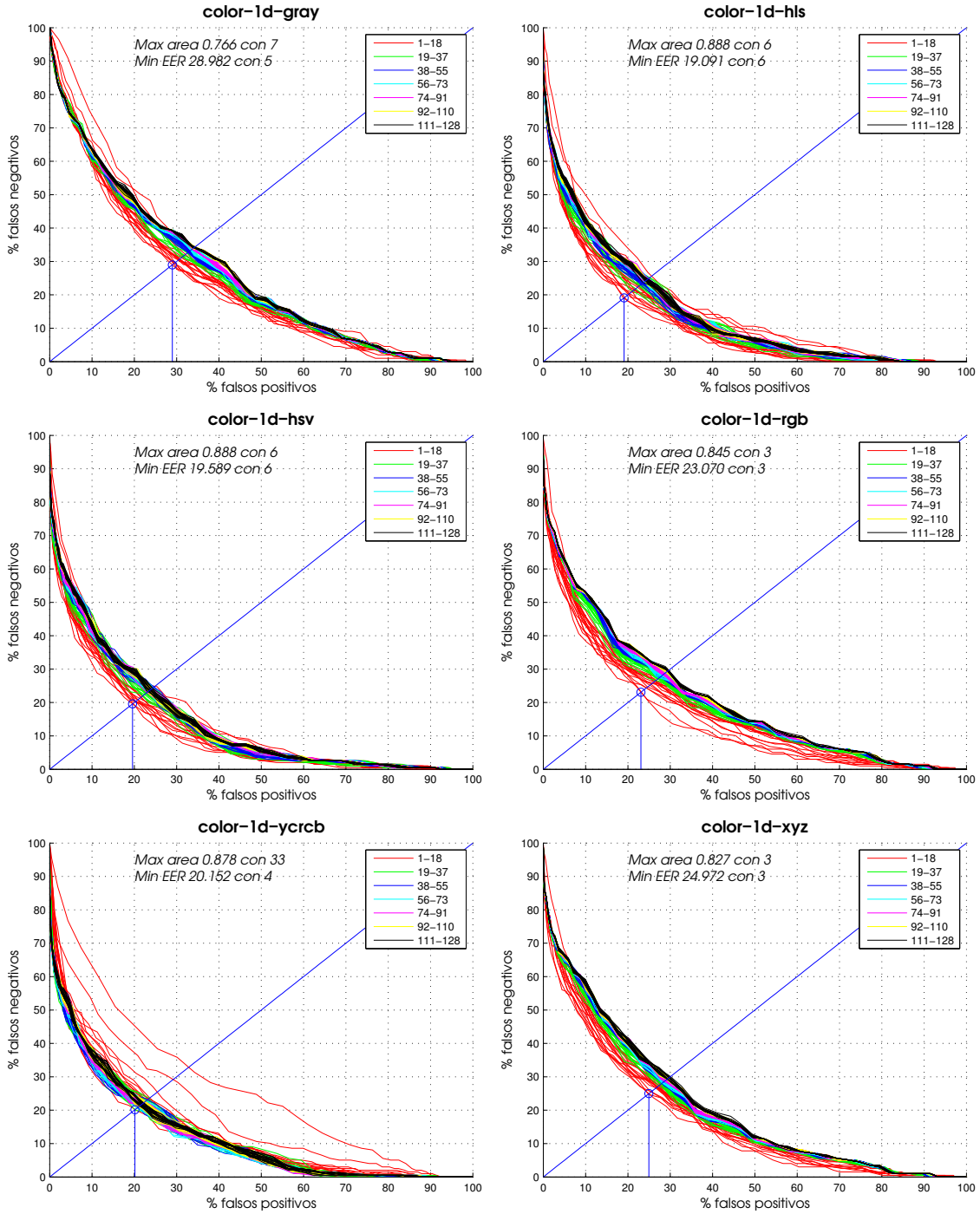

Figura 6.5: Resultados de la búsqueda por histogramas de color unidimensionales en la galería sencilla de 100 imágenes. De arriba abajo, de izquierda a derecha: gris, HLS, HSV, RGB, YCrCb y XYZ. Para cada uno, se prueban tamaños de imagen entre 1 y 128. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

un tiempo pequeño en comparación con el resto de cálculos (ley de Amdahl [12]).

Respecto a la tasa de errores, el mejor ajuste de parámetros obtiene un EER del 18,3 % para los histogramas de color ha sido el uso de histogramas multidimensio-

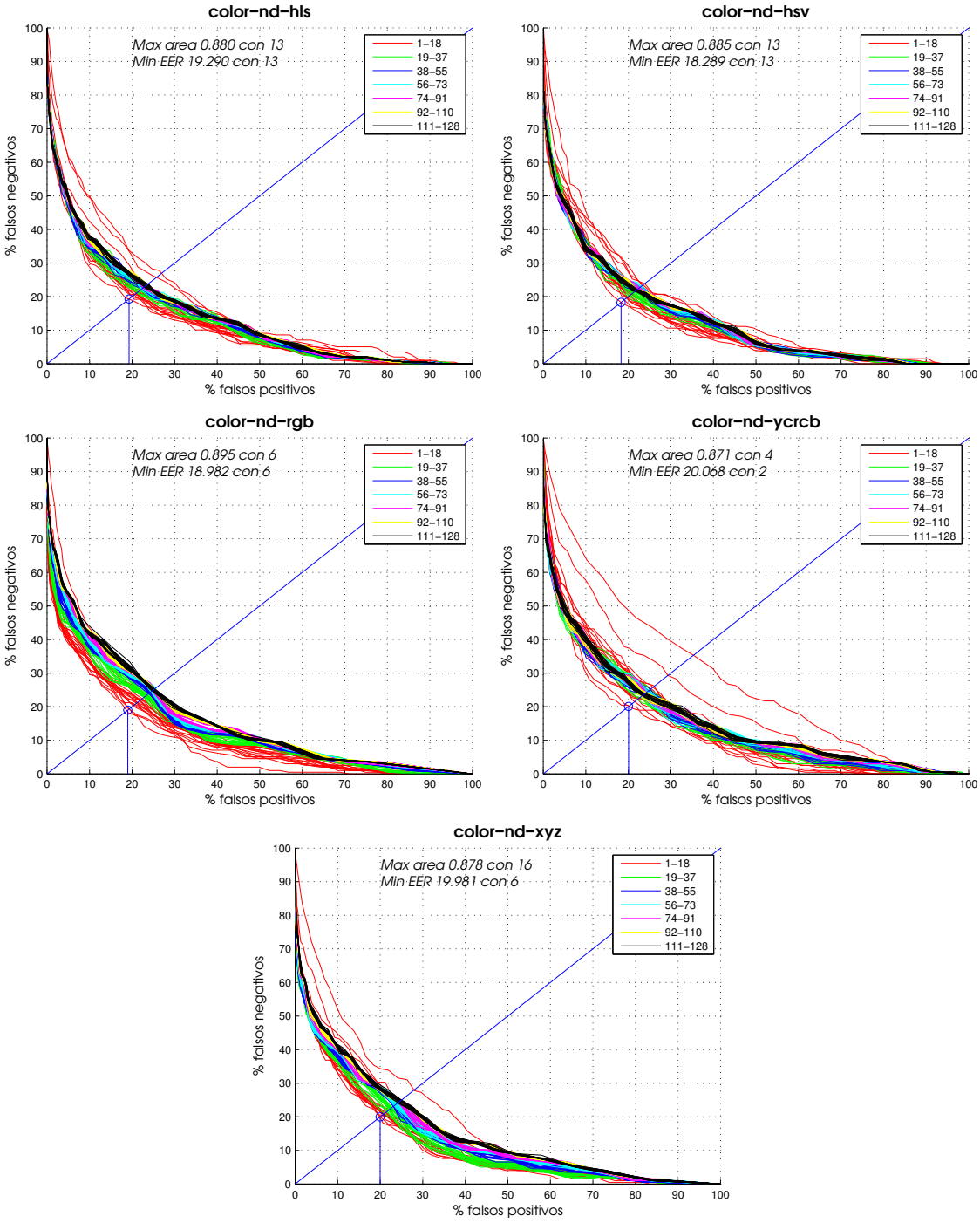

Figura 6.6: Resultados de la búsqueda por histogramas de color multidimensionales en la galería sencilla de 100 imágenes. De arriba abajo, de izquierda a derecha: HLS, HSV, RGB, YCrCb y XYZ. En YCrCb, HLS, HSV se separa el canal de luminosidad del resto. Para cada uno, se prueban tamaños de imagen entre 1 y 128. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

nales en el espacio de color HLS, con un tamaño de 13 cubetas por dimensión. En el *matching* de imágenes, el mejor ajuste consigue un EER del 13,3 %, utilizando un

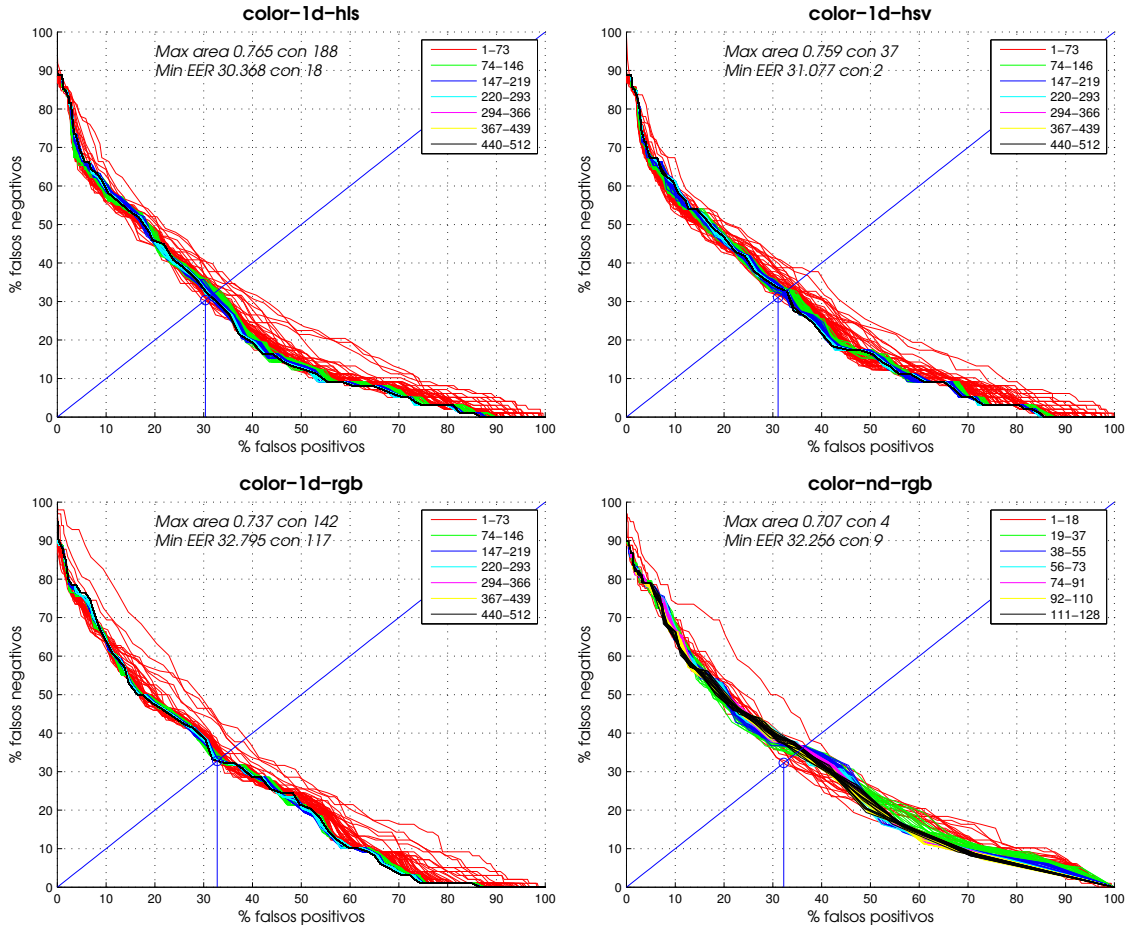

Figura 6.7: Resultados de la búsqueda por histogramas de color en la galería complicada de 50 imágenes. De arriba abajo, de izquierda a derecha: HLS, HSV, RGB unidimensionales y RGB multidimensional. Para cada uno, se prueban tamaños de imagen entre 1 y 512. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

tamaño de $52x52$ en RGB, claramente mejor que lo que consiguen los histogramas de color.

En la figura 6.9 se muestran los dos primeros resultados devueltos por el método de histogramas de color, ajustado con la mejor configuración encontrada, sobre la base de datos realista de 10200 imágenes cuando se le pasa una foto que no tiene almacenada.

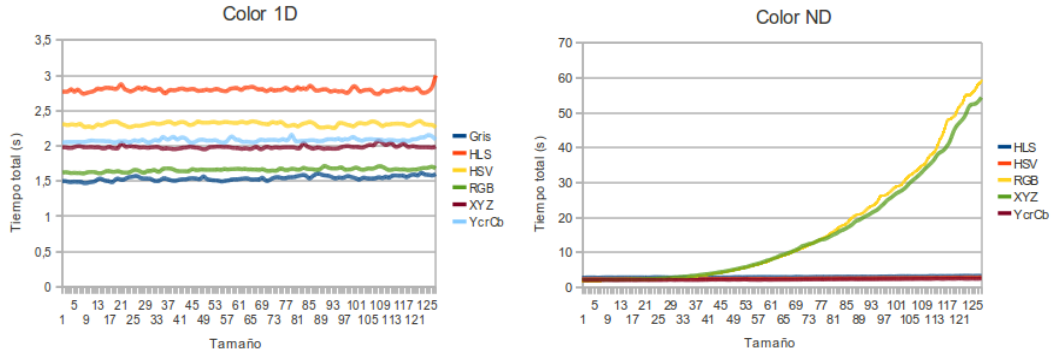

Figura 6.8: Tiempo empleado para ejecutar 100 búsquedas sobre la base de datos sencilla de 100 imágenes. El tiempo sólo incluye el procesamiento de la imagen de entrada y la búsqueda en la base de datos. De izquierda a derecha: histogramas unidimensionales (en gris, HLS, HSV, RGB, XYZ e YCrCb) e histogramas multidimensionales (en HLS, HSV, RGB, YCrCb y XYZ). Para cada uno se prueban tamaños entre 1 y 128.

Figura 6.9: Resultados de la búsqueda por histogramas de color en la galería realista de 10200 imágenes. De izquierda a derecha: imagen de entrada, y dos primeros resultados devueltos.

6.3 Características SIFT

En el estudio de búsqueda de imágenes usando características SIFT se ha probado en primer lugar el clasificador de tipo lista, en el que todas las características se almacenan en una lista en memoria, sobre la base de datos sencilla. Esta ejecución no se ha podido completar debido al enorme tiempo de ejecución, del orden de varios minutos por cada consulta, como se muestra en la tabla 6.1 marcado con el símbolo de infinito.

El principal problema de este clasificador de tipo lista para SIFT, es la gran cantidad de *features* que detecta en algunas imágenes, por encima de 2000. Por ello, se ha intentado reducir la cantidad de *features* detectadas y, si es posible, mejorar su calidad, mediante un ajuste de parámetros del algoritmo. Este ajuste ha resultado en un tiempo de ejecución más razonable, aunque aún muy grande para una aplicación realista: 8 segundos por búsqueda en una base de datos de sólo 100 imágenes.

Además se ha detectado que existe una sobrerrepresentación de algunas imágenes en el clasificador, ya que SIFT extrae un número muy variable de *features* según el tipo de escena de la imagen: en escenas sencillas, con pocos elementos representativos, puede obtener menos de 10 puntos característicos; mientras que en otro tipo

de escenas el número puede superar las 2000, como se ha dicho. Esta diferencia de tres órdenes de magnitud es un gran problema y es una de las causantes del bajo rendimiento de SIFT respecto a las técnicas basadas en histogramas y *matching*.

Para mejorar el tiempo de ejecución y la eficiencia, se ha utilizado el clasificador descrito en la sección 5.5. Aplicando esta técnica, se ha conseguido reducir el tiempo de búsqueda considerablemente, como se muestra en la tabla 6.1, aunque el funcionamiento del buscador ha sido notablemente peor, como se ve en la figura 6.10. Esto es debido a la imposibilidad de obtener buenos comparadores cuando sólo se dispone de un subconjunto de *features*, y en parte a la sobre-representación de algunas imágenes en el árbol, por el variable número de *features* que SIFT devuelve según el tipo de escena en la imagen.

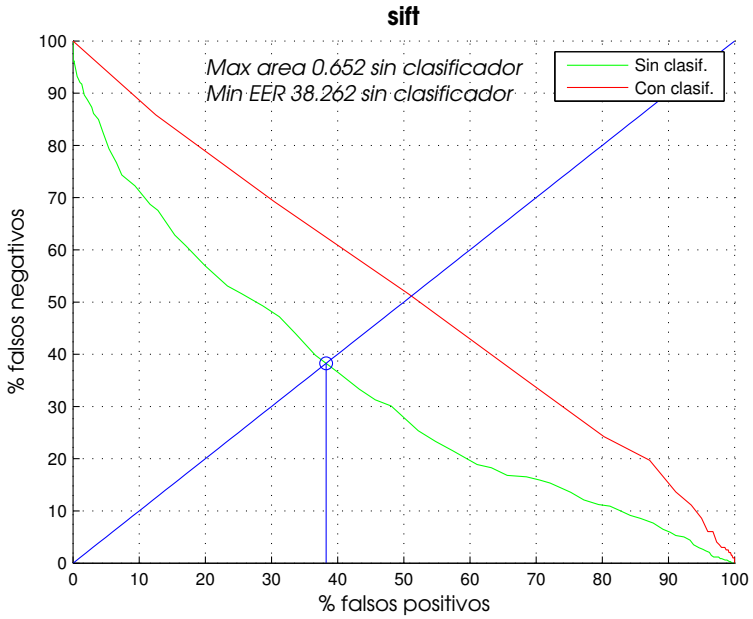

Figura 6.10: Resultados de la búsqueda por características SIFT en la galería sencilla de 100 imágenes. Se prueba con el clasificador tipo lista y con el tipo árbol. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

Método	Tiempo (segundos)
SIFT	∞
SIFT con ajuste de parámetros	849,14
SIFT utilizando el clasificador	76,99

Tabla 6.1: Tiempo empleado por SIFT para ejecutar 100 búsquedas sobre la base de datos sencilla de 100 imágenes. El tiempo sólo incluye el procesamiento de la imagen de entrada y la búsqueda en la base de datos. Se prueba con el clasificador tipo lista con y sin ajuste de parámetros del algoritmo y con el tipo árbol con parámetros ajustados.

Con SIFT sin el clasificador se ha conseguido un EER de 38,2 %, claramente inferior al obtenido con *matching* e histogramas de color, aunque puede ser una opción de futuro si se combina con otras técnicas.

6.4 Características SURF

Para intentar aliviar algunos de los problemas de SIFT, se ha intentado probar el método SURF, que promete, como mínimo, la misma efectividad combinado con un menor requerimiento de recursos *hardware*.

Una de las principales ventajas de SURF sobre SIFT, es la posibilidad de hacer un filtrado de *features* una vez han sido calculadas, ya que SURF nos da un valor de la bondad de cada una, calculado como el determinante del Laplaciano. Utilizando este valor, se ha decidido que siempre se utilicen las 100 mejores *features* (las de valor más alto) detectadas en cada imagen como descriptores de esa imagen. Las gráficas mostradas en la figura 6.11 corresponden a la ejecución de SURF sobre la base de datos sencilla.

Utilizando un clasificador de tipo lista, como el usado en la sección 5.3, los resultados son mejores que en aquélla, obteniendo un EER del 24,5 %, muy cercano al obtenido con los histogramas de color y, en menor medida, al que se consigue con *matching*.

Así, la eficacia de SURF ha superado notablemente a SIFT, llegando a obtener un buen rendimiento, aunque peor que los métodos más sencillos que se han implementado basados en *matching* e histogramas. Debido a que el tiempo empleado en las búsquedas es mucho mejor que el de SIFT, como se muestra en la tabla 6.2, el uso del clasificador no aporta tanta mejora en tiempo y sí empeora mucho la efectividad del método por la dificultad de encontrar comparadores que trabajen sobre esa estructura. SURF resulta un método muy interesante para una implementación que combine varios métodos, de forma que se aprovechen las mejores propiedades de cada uno.

Método	Tiempo (segundos)
SURF	304,47
SURF utilizando el clasificador	212,96

Tabla 6.2: Tiempo empleado por SURF para ejecutar 100 búsquedas sobre la base de datos sencilla de 100 imágenes. El tiempo sólo incluye el procesamiento de la imagen de entrada y la búsqueda en la base de datos. Se prueba con el clasificador tipo lista con el tipo árbol.

En la figura 6.12 se muestran los dos primeros resultados devueltos por el algoritmo SURF con el clasificador de tipo lista, sobre la base de datos realista de 10200 imágenes cuando se le pasa una foto que no tiene almacenada.

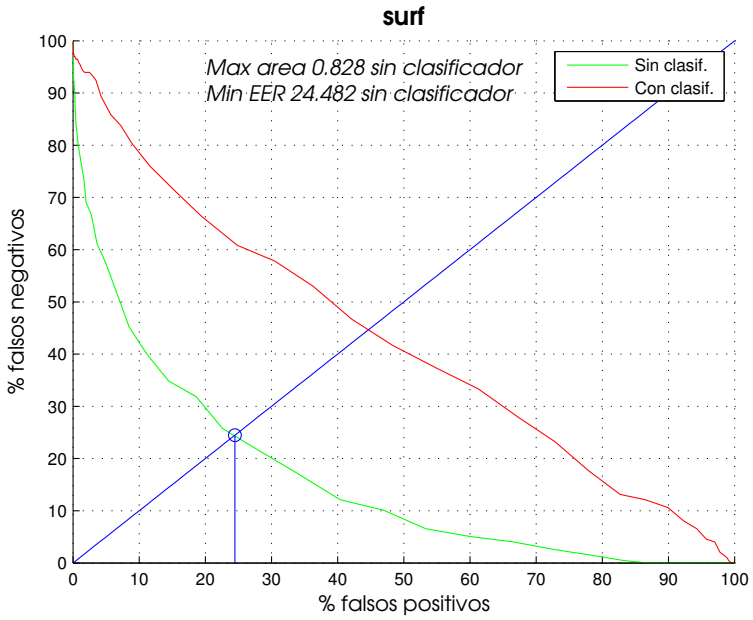

Figura 6.11: Resultados de la búsqueda por características SURF en la galería sencilla de 100 imágenes. Se prueba con el clasificador tipo lista y con el tipo árbol. Max area: máximo valor del área sobre la curva ROC. Min EER: mínimo valor del *Equal Error Rate*.

Figura 6.12: Resultados de la búsqueda por características SURF sin el clasificador en la galería realista de 10200 imágenes. De izquierda a derecha: imagen de entrada, y dos primeros resultados devueltos.

7. Conclusiones y vías futuras

Este proyecto fin de carrera ha abordado el complejo problema del reconocimiento y clasificación de categorías de imágenes, para su búsqueda por contenido de forma rápida y efectiva. Este es un problema muy complejo, incluso a un humano le resulta difícil decidir qué imágenes han de ser devueltas para una consulta concreta, por lo que automatizar el proceso es aún más difícil. Para ello se han implementado diversos métodos de comparación entre imágenes: *matching*, histogramas de color, SIFT y SURF.

El método de *matching* ha obtenido los mejores resultados. No obstante, esto se debe en muchos casos a una coincidencia en el contenido global de las imágenes ya que, con una base de datos suficientemente grande, es muy probable que tengamos muchas imágenes muy similares ya indexadas en nuestra base de datos y, por tanto, se puedan recuperar sin mucho esfuerzo un buen número de las mismas.

En cuando a los histogramas de color, se han obtenido resultados muy buenos, manteniendo el coste computacional muy bajo, por lo que se cree que puede ser una vía con mucho futuro para este campo. Además, podría aplicarse algún clasificador más sofisticado que rebaje el tiempo de acceso para bases de datos de varios millones de imágenes, como las que los buscadores reales manejan hoy en día.

Finalmente, las técnicas basadas en *features*, como SIFT y SURF, han demostrado no ser tan efectivas como las anteriores, en parte por su complejidad de representación. Aunque el principal escollo para utilizar estos métodos es el gran peso computacional que requiere extraer las características de la imagen de entrada, esto puede reducirse con implementaciones avanzadas con multiprogramación o en GPU, como se vio en la sección 4.2.3. Aun así queda el problema de la búsqueda de las *features* más similares de entre todas las almacenadas en la base de datos, problema que también puede ser minimizado mediante el uso de implementaciones optimizadas, haciendo que una aplicación de esta clase sea viable en un entorno realista.

A pesar de todo, SURF ha conseguido unos resultados bastante buenos, en un entorno muy complejo podría superar a los métodos sencillos basados en histogramas y *matching*, que apenas abstraen información de la imagen, por lo que un método basado en *features* sería más adecuado en esa situación.

Tras probar varios métodos de comparación de *features*, se ha conseguido un buen rendimiento que se aproxima al de los métodos de comparación de histogramas, aunque sigue permaneciendo por debajo de ellos, con un coste en recursos mucho mayor, debido al mayor tamaño de las features respecto a, por ejemplo, una simple imagen reescalada. No obstante, este método tiene el potencial de encontrar similitudes no triviales entre imágenes, donde el color y la forma global de la imagen pueden no funcionar. Es decir, se espera que tengan una capacidad de generalización muy superior.

Es por esto que SURF es nuestro principal candidato para ser combinado con otros métodos, dado que nos aporta las ventajas de un método basado en *features*, la alta

capacidad de generalización de objetos de una escena, con un más que aceptable rendimiento, y en una combinación con un método más rápido, por ejemplo, uno que utilice la información contenida en los metadatos asociados a las imágenes, puede tener un gran futuro.

Teniendo en cuenta la complejidad del problema, los resultados son muy prometedores, sobre todo con la posibilidad de combinar varios métodos. Esta combinación de métodos permitiría mejorar la eficacia y reducir los tiempos de búsqueda.

Hemos descubierto que la base de datos de prueba sobre la que se trabaja es absolutamente esencial, tanto para ayudar en el proceso de prueba y optimización, como para obtener ajustes que aumenten la fiabilidad de los resultados en entornos realistas. El diseño de una buena base de datos se hace más importante según aumente la complejidad de los métodos implementados. Un *benchmark* que nos ayude a decidir entre métodos u optimizar los ya implementados puede ser más importante que la propia implementación del método.

Así, habiendo cumplido el objetivo que se planteó para el proyecto: analizar, diseñar e implementar un buscador de imágenes que maximice la eficacia sin dejar de lado la eficiencia, se propone para un futuro la ampliación de la aplicación con nuevos clasificadores para los algoritmos estudiados y el estudio de otros extractores de características más ligeros, así como la combinación de diferentes métodos de comparación de imágenes. La combinación de métodos es la vía más prometedora, ya que permite extraer lo mejor de varios métodos, mejorando tanto la eficiencia de los clasificadores como los resultados que con ellos se obtienen, al usar mejores y más complejos sistemas de comparación.

BIBLIOGRAFÍA

[1] S.E. Madnick A Gupta, H-M Toong. A microcomputer-based image database management system. *IEEE Transactions on Industrial Electronics*, IE-34:83–8, 1987.

[2] Wikipedia. List of CBIR engines — Wikipedia, the free encyclopedia, 2010. [Online; accessed 09-April-2010].

[3] Tao-I Hsu, A. D. Calway, and R. Wilson. Texture analysis using the multiresolution fourier transform. In *In Scandinavian Conference on Image Analysis*, pages 823–830, 1993.

[4] Hui Zhang, Jason E. Fritts, and Sally A. Goldman. A fast texture feature extraction method for region-based image segmentation. In *16th Annual Symposium on Image and Video Communication and Processing, SPIE*, page 2005, 2005.

[5] Narendra Ahuja and Sinisa Todorovic. Extracting texels in 2.1d natural textures. In *In ICCV*, 2007.

[6] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, June 2006. **oral presentation**.

[7] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[8] Jorge Valverde Rebaza. Detección de bordes mediante el algoritmo de canny. `http://www.seccperu.org/node/348`, 2007.

[9] Francisco José Moreno Navas. Desarrollo de un proceso de deconvolución para imágenes de microscopía usando GPUs. 2009.

[10] Changchang Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). `http://cs.unc.edu/~ccwu/siftgpu`, 2007.

[11] Michael Cowgill. OpenSURF: GPU Enhancement. `http://cs264.org/projects/web/Cowgill_Michael/cowgill/Index.html`, 2009.

[12] Wikipedia. Ley de Amdahl — Wikipedia, la enciclopedia libre, 2010. [Online; accessed 09-April-2010].

[13] Rishav Chakravarti and Xiannong Meng. A study of color histogram based image retrieval. In *ITNG '09: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, pages 1323–1328, Washington, DC, USA, 2009. IEEE Computer Society.

[14] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[15] Wikipedia. List of CBIR engines — Wikipedia, the free encyclopedia, 2010. [Online; accessed 09-April-2010].

[16] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *In ECCV*, pages 404–417, 2006.

[17] David Nistér and Henrik Stewénius. Scalable recognition with a vocabulary tree. In *IN CVPR*, pages 2161–2168, 2006.

[18] Duy nguyen Ta, Wei chao Chen, Natasha Gelfand, and Kari Pulli. Surftrac: Efficient tracking and continuous object recognition using local feature descriptors. In *In IEEE Conf. on Computer Vision and Pattern Recognition (CVPR09*, 2009.

[19] V. S. V. S. Murthy, Swarup Kumar, and P. Sankara Rao. Content based image retrieval using hierarchical and k-means clustering techniques.

[20] Khaled Alsabti. An efficient k-means clustering algorithm. In *In Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.