Trabajo Fin de Grado
Grado en Ingeniería Informática

Reconocimiento de objetos en Android para aplicaciones de asistencia

Object recognition assistant application for Android

Autor

Alejandro Márquez Ferrer

Directores

Ana Cristina Murillo Arnal
Darío Suárez Gracia

Universidad de Zaragoza
Escuela de Ingeniería y Arquitectura

Noviembre 2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

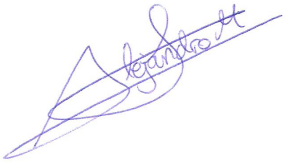
D./Dª. Alejandro Márquez Ferrer,

con nº de DNI 73134016L en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Reconocimiento de objetos en Android para aplicaciones de asistencia,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de noviembre de 2016



Fdo: Alejandro Márquez Ferrer

Reconocimiento de objetos en Android para aplicaciones de asistencia

Resumen

La capacidad de identificar y reconocer objetos es una tarea que la percepción del ser humano realiza sin problemas y de manera rutinaria. Sin embargo, crear un sistema informático capaz de igualar estas capacidades sigue siendo uno de los principales campos de investigación dentro de la disciplina de la visión por computador, VC. Esta razón unida a las posibilidades de ayuda que la VC puede ofrecer a personas con necesidades especiales han sido los acicates que han dado lugar a este trabajo. Además, en los últimos años los *smartphone* han experimentado una gran evolución en términos de potencia, haciendo que cada persona tenga un "mini-ordenador" en el bolsillo. Por ello, se ha enfocado el trabajo hacia esta plataforma y se han evaluado distintos algoritmos reconocedores para implementarlos en un pequeño prototipo de asistencia al usuario que, en tiempo real, sea capaz de identificar qué objetos que aparezcan en la cámara y listar los alérgenos que contiene.

Se han investigado dos tipos de algoritmos distintos para el reconocimiento de los objetos: uno basado en *local features*, que puede ser ejecutado tanto en local como en modo cliente-servidor, y otro basado en técnicas más novedosas, concretamente *Deep Learning*, que debido a que es más exigente en términos de recursos sólo podrá ser ejecutado en modo cliente-servidor.

Como punto de partida, se ha creado una base de datos de objetos propios de un supermercado. A continuación, se ha implementado un reconocedor de objetos basado en *local features*, y se ha investigado y evaluado qué algoritmos de reconocimiento funcionaban mejor con esa base de datos. Una vez elegido el que se ha considerado más óptimo, se ha implementado ese mismo reconocedor en modo cliente-servidor. Después, se ha implementado el reconocedor basado en *Deep Learning*, y se han evaluado sus resultados. Por último, se han comparado el rendimiento y los resultados obtenidos entre las distintas técnicas y arquitecturas, y se han extraído conclusiones acerca de ellos: ventajas, desventajas de cada uno y posibles mejoras.

Finalmente, se ha desarrollado un pequeño prototipo, consistente en una aplicación de asistencia cuya función es reconocer los objetos de la base de datos creada a través de la cámara del dispositivo e indicar la lista de alérgenos que contienen, todo ello en tiempo real.

Agradecimientos

Quisiera en primer lugar dar las gracias a mis directores de proyecto Ana Cris y Darío por toda la ayuda prestada a lo largo del trabajo, el interés mostrado en el mismo, y las cosas que gracias a ellos he aprendido.

Por supuesto también agradecer a mis padres sus ánimos a lo largo de toda esta carrera, apoyándome en todo momento con su paciencia y consejos. No os preocupéis, que tras este trabajo ya podré tener tiempo por fin para jugar con la gata.

A mi novia Nuria, que prácticamente desde que me conoce me ha tenido que aguantar en mis momentos de desesperación y agobio con la carrera. A cambio, prometo arreglarte el ordenador y cualquier aparato electrónico que se te estropee, que para eso soy ingeniero informático.

Y por último, pero para nada menos importante, a mis amigos, en especial a mis compañeros de clase: compañeros de batallas, de desesperaciones, pero también de risas y muy buenos momentos. Sin vosotros, este camino hubiese sido mucho más duro.

¡Muchas gracias a todos!

Índice

1. Introducción

Este primer capítulo describe el contexto en el que se desarrolla el trabajo, así como la motivación del mismo. A continuación, se habla del trabajo relacionado con este tema. Después, se hace una breve introducción a los objetivos de alto nivel del trabajo y a las tecnologías utilizadas, finalizando con la explicación de la estructura que presenta el resto de la memoria.

1.1. Contexto y motivación

La visión por computador es una rama de la inteligencia artificial que se basa en métodos por los cuales los computadores puedan adquirir, procesar, analizar y entender imágenes digitales del mundo real para producir información. Mientras que la inteligencia artificial busca que las máquinas imiten las formas de pensar de la mente humana (aprendizaje, resolución de problemas...), la visión por computador trata de reproducir en éstas la manera que tienen los seres humanos de, mediante el sentido de la vista, comprender el mundo real.

Durante los últimos años y gracias a las nuevas tecnologías, la visión por computador ha experimentado un gran auge y se han desarrollado numerosas aplicaciones. Una de las principales, y en la que se va a centrar este trabajo, es el reconocimiento de objetos.

Actualmente, existen multitud de reconocedores de objetos en nuestra vida cotidiana, como por ejemplo los lectores de matrículas, reconocedores de rostros, clasificadores de objetos, aplicaciones de asistencia... Además, el hecho de que la inmensa mayoría de personas posee un *smartphone* o *tablet* con Internet, unido a la evolución que han experimentado estos aparatos en cuestión de prestaciones, hacen que puedan ejecutar reconocedores sin necesidad de ningún dispositivo adicional.

Es precisamente en este marco donde se enmarca este sistema. Se han investigado, implementado y evaluado distintos algoritmos y arquitecturas para el reconocimiento de objetos, y se ha desarrollado un pequeño prototipo de asistencia al usuario. Este prototipo le permitirá reconocer una serie de objetos propios de un supermercado y obtener la lista de alérgenos que lo componen, todo esto en tiempo real.

1.2. Trabajo relacionado

Muchos investigadores están intentando optimizar los algoritmos de reconocimiento, en la mayoría de los casos, para poder ejecutarlos desde dispositivos móviles y en tiempo real. Algunas investigaciones se centran en ejecutar todos los cálculos en el dispositivo, como por ejemplo [1], que consiste en una aplicación móvil de guía de museos mediante reconocimiento de objetos. Otras, sin embargo, implementan un sistema distribuido, de tal manera que los cálculos se realicen entre el cliente y el servidor [2].

Este proyecto se centra en el reconocimiento de objetos para asistencia al usuario, y se han

comparado ambos enfoques, tanto solo cliente, como cliente/servidor, para analizar cual es la mejor aproximación en términos de precisión y rendimiento. Relacionado con este tema está esta publicación [3], donde hablan del desafío que supone reconocer productos de un supermercado debido a que las imágenes tomadas ahí tienen ruido mientras que las de la base de datos obtenidas de Internet son limpias e incluso a veces generadas por ordenador.

En el dominio de la asistencia a personas con dificultades visuales existen ya algunas aplicaciones en el mercado. A continuación, se detallarán algunas de ellas:

- **OrCam** [1]: OrCam es una cámara que se coloca en la montura de las gafas y permite reconocer objetos y textos. Mediante un gesto de pulsación, el usuario indica el objeto a reconocer, y OrCam le comunica por voz, gracias a los auriculares que incorpora, el resultado del reconocimiento.
- **Medicamento Accesible Plus** [2]: Esta aplicación Android (también disponible en iOS) permite, entre otras cosas, reconocer el código de barras de cualquier medicamento y leer su información en voz alta al usuario, lo cual es muy útil para personas con problemas de vista.
- **KNFB Reader** [3]: Aplicación disponible para iOS y Android que permite reconocer textos en multitud de formatos y lugares gracias a algoritmos de visión por computador, permitiendo reproducirlos por voz. Al igual que la anterior, también está orientada a personas con déficit visual.

1.3. Objetivos

Los objetivos generales de este proyecto son:

- Crear una pequeña base de datos de objetos típicos de supermercado y de los alérgenos que contienen.
- Estudiar y evaluar distintos tipos de algoritmos de reconocimiento de objetos basado en *Local Features* utilizando solo la plataforma Android, y elegir el más preciso y rápido posible con respecto a la base de datos creada.
- Implementar el algoritmo elegido anteriormente tanto para ser ejecutado en local, como en arquitectura cliente/servidor, y evaluar el rendimiento de ambas arquitecturas.
- Estudiar e implementar otro algoritmo de reconocimiento de objetos más exigente en términos de recursos, basado en *Deep Learning*, e implementarlo en arquitectura cliente/servidor.

[1] Página oficial: `http://www.orcam.com/`

[2] Enlace a la aplicación en Google Play: `https://play.google.com/store/apps/details?id=com.technosite.medicamentoaccesible&hl=es/`

[3] Página oficial: `http://www.knfbreader.com/`

- Evaluar y comparar el rendimiento y resultado de los distintos algoritmos y arquitecturas utilizados, analizando las ventajas, desventajas y resultados de cada uno.
- Creación de un prototipo que permita reconocer objetos de la base de datos mencionada en tiempo real, y proporcione al usuario información acerca de los alérgenos que estos contienen.

1.4. Distribución temporal de las tareas

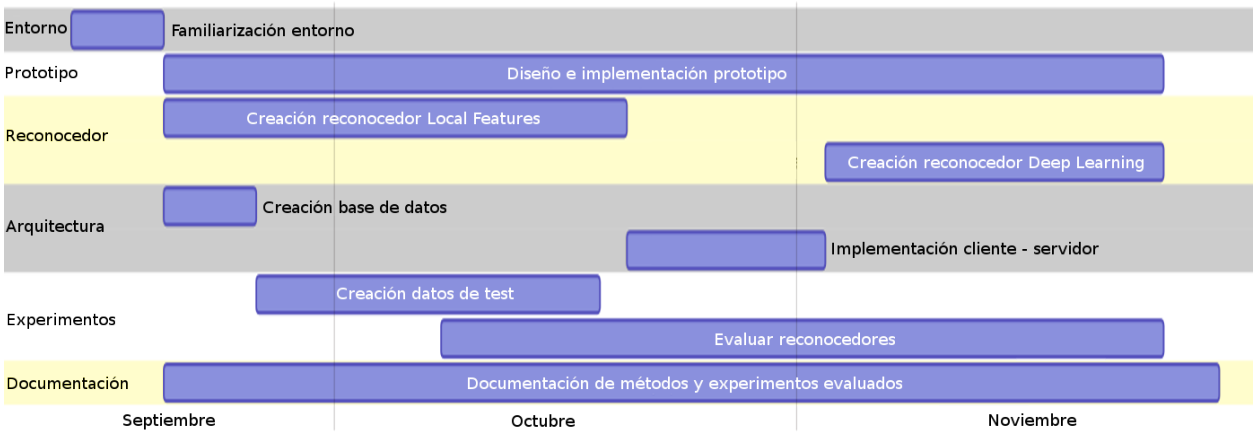


Figura 1: Diagrama de Gantt de las tareas realizadas

1.5. Tecnologías utilizadas

Para llevar a cabo la implementación de los objetivos citados anteriormente, es necesario el uso de varias herramientas y tecnologías. En la tabla 1 de esta sección aparecen enumeradas las diferentes herramientas y tecnologías empleadas durante el desarrollo del sistema. La tabla se encuentra agrupada por categorías, mostrándose en primer lugar las tecnologías relacionadas con la implementación del sistema en local. A continuación, se especifican las tecnologías utilizadas para la comunicación entre el cliente y el servidor. Después, se muestran las herramientas empleadas para implementar el reconocedor mediante *Deep Learning*. Esto va seguido por las utilidades usadas para procesar los resultados en bruto obtenidos en el sistema. Por último, aparecen las herramientas de desarrollo y las de ofimática y documentación, que sirven de apoyo para la realización del proyecto.

Categoría	Tecnologías y Herramientas	
Implementación del sistema	Android	
	CrystaX's Android NDK	https://www.crystax.net/en/android/ndk
	C++	
	OpenCV	http://opencv.org/
Comunicación Cliente-Servidor	C++	
	Protocol Buffers	https://developers.google.com/protocol-buffers/
Reconocimiento Deep Learning	Caffe [4]	http://caffe.berkeleyvision.org/
Procesamiento de resultados	Python	
Herramientas de desarrollo	Eclipse IDE	
	Android Studio	
	GitHub	
	Google Drive	
Ofimática y documentación	Sublime Text	
	LaTeX	https://www.latex-project.org/
	Overleaf	https://www.overleaf.com/

Tabla 1: Tecnologías y herramientas utilizadas en el proyecto

1.6. Estructura de la memoria

La memoria del trabajo se compone de seis capítulos y seis anexos, que se especifican a continuación:

- **Capítulo 1:** Introducción. Se hace una introducción general del proyecto, mostrando el contexto y la motivación del mismo, el trabajo relacionado, los objetivos del proyecto y las tecnologías utilizadas.
- **Capítulo 2:** Reconocimiento de objetos. Se abordan los aspectos teóricos de las dos técnicas utilizadas para el reconocimiento de objetos en este proyecto, que son el reconocimiento basado en *Deep Learning* y el reconocimiento basado en *local features*.
- **Capítulo 3:** Diseño del sistema. Se describe el sistema realizado durante el proyecto, lo que incluye también detalles de su implementación y de las distintas arquitecturas utilizadas.
- **Capítulo 4:** Implementación del prototipo. Se comenta de forma más general el prototipo creado, al igual que se hace una descripción de su mapa de navegación y sus pantallas.
- **Capítulo 5:** Experimentos y resultados. Se explican los experimentos llevados a cabo, las optimizaciones realizadas, y los resultados obtenidos para cada prueba.
- **Capítulo 6:** Conclusiones y trabajo futuro. Se muestran las conclusiones obtenidas de la realización del proyecto, el trabajo que se puede llevar a cabo para mejorar y continuar el sistema y la opinión personal del estudiante.
- **Anexo A:** Manual de instalación del sistema. Se dan instrucciones de cómo instalar el sistema (tanto la aplicación como el servidor) así como de sus requisitos.

- **Anexo B:** Estructura de la base de datos. Se explica cómo está estructurada la base de datos utilizada por el sistema.
- **Anexo C:** Diagrama de clases del sistema: Se incluye un diagrama de clases del sistema implementado.
- **Anexo D:** Descriptor/Detector ORB. Se detallan las particularidades de este algoritmo de extracción y descripción de puntos de interés.
- **Anexo E:** Procesamiento en Redes neuronales convolucionales (CNN). Se explica más en detalle cómo funcionan las redes neuronales convolucionales.
- **Anexo F:** Resultados adicionales de los experimentos. Se muestran todos los resultados de las pruebas realizadas a lo largo del proyecto.

2. Reconocimiento de objetos

En esta sección se abordarán los aspectos teóricos de las dos técnicas que se han utilizado para reconocer objetos en este proyecto: el reconocimiento mediante *Local features* y el reconocimiento mediante *Global features*, haciendo uso de *Deep Learning*. En ambos casos, existirá una base de datos con imágenes de objetos y su información correspondiente. El objetivo del sistema de reconocimiento es decir si la imagen tomada por el usuario corresponde con alguno de los objetos de la base de datos.

2.1. Reconocimiento mediante *Local Features*

Una *local feature*, o característica de interés local, es un patrón de la imagen que debido a una serie de cualidades, puede ser utilizada como punto de referencia de esa imagen. En la publicación *Local Invariant Feature Detectors: A Survey* [5] se define como "Un patrón de la imagen que difiere de sus vecinos inmediatos". La principal característica de una buena *feature* es que sea única, y por lo tanto, no pueda confundirse con otras *features*. Hay muchas propuestas para obtener *local features* en la literatura, que pueden estar basadas en contornos, esquinas, regiones... En este trabajo, nos hemos centrado en el uso de *local features* que consisten en un punto de interés descrito por el contenido de una zona alrededor de dicho punto. Uno de los primeros detectores/descriptores de la literatura que supuso grandes avances en sistemas de reconocimiento basados en *local features* es el algoritmo de SIFT [8].

Un sistema de reconocimiento de objetos utilizando puntos de interés consiste en detectar dichos puntos en una imagen y compararlas con las de las imágenes de la base de datos. El objeto de la base de datos cuyos puntos de interés presenten más similitud con los de imagen dada, será identificado como el que aparece en esa imagen. Este sistema de detección y comparación de puntos no solo se utiliza para el reconocimiento de imágenes, sino que también se emplea para, por ejemplo, la creación de imágenes panorámicas [6], navegación de robots [7]... Podemos dividir el reconocimiento basado en puntos de interés en cuatro fases:

- **Detección de puntos de interés:** Consiste en identificar aquellos puntos que cumplen la característica nombrada anteriormente. Existen numerosas técnicas y algoritmos para detectarlos, como por ejemplo el detector FAST [10] u ORB [11]. Sin embargo, un buen detector de *local features* debería cumplir todas o la mayoría de estas características:
  - **Robusto:** que sea independiente de las transformaciones geométricas (escala, rotación...) y fotométricas (brillo, exposición...).
  - **Preciso:** que sea preciso a la hora de localizar los puntos de interés.
  - **Eficiente:** que sea rápido, sobre todo orientado a las aplicaciones en tiempo real.
  - **Repetible:** que se obtengan los mismos puntos cada vez que se ejecute.

En la figura 2 se pueden observar distintos puntos de interés extraídos de una imagen.

Figura 2: Detección de las *local features* de una imagen

- **Descripción de puntos de interés:** Una vez obtenidos los puntos, el siguiente paso es obtener el descriptor de cada uno de ellos. Existen multitud de algoritmos y soluciones para ello, pero la idea en todos es obtener de cada *feature*, la información de la región vecina de ella, información que deberá ser invariante de las transformaciones de la imagen. Podemos agrupar los descriptores de puntos de interés más extendidos en dos tipos, según el tipo de dato utilizado en su descripción: descriptores binarios o no binarios. Los primeros son generalmente más eficientes, pero menos precisos que los segundos. Algunos de los descriptores no binarios más conocidos son por ejemplo SIFT [8] y SURF [9], que utilizan vectores de valores reales como descriptores. Ejemplos de descriptores binarios, que utilizan un vector de valores binarios como descriptor, son ORB [11] o BRISK [12]. En el anexo D se explicará ORB en detalle, ya que es el que se utilizará en el proyecto.

  En la figura 3[4] se pueden observar las *local features* detectadas en una imagen, y el descriptor de una de ellas, obtenido mediante el algoritmo SIFT [8], que está basado en los gradientes de la imagen.

[4]Fuente imagen:`http://www.codeproject.com/Articles/619039/Bag-of-Features-Descriptor-on-SIFT-Features-with-O`

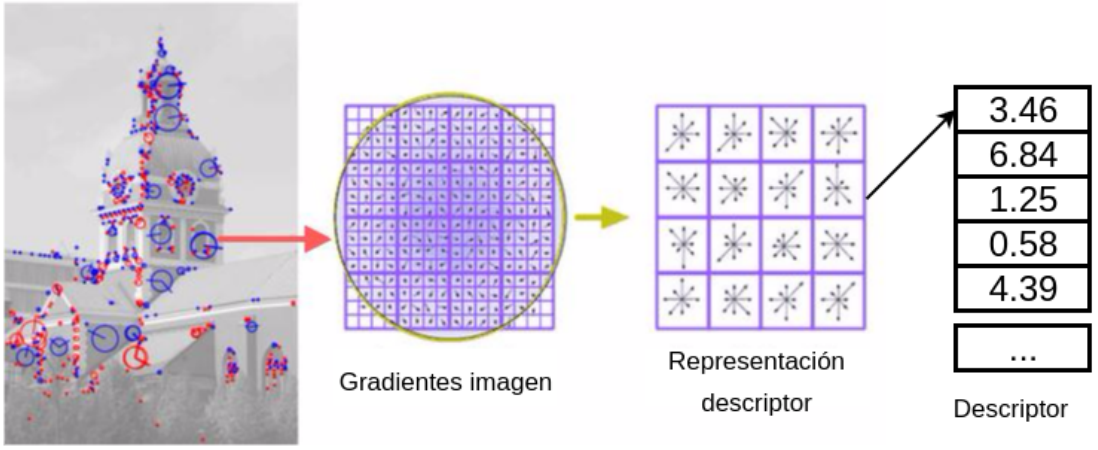


Figura 3: A la izquierda, los puntos de interés detectados en una imagen de ejemplo. En el centro, una rejilla que representa los valores y direcciones del gradiente del nivel de gris en cada pixel alrededor del punto de interés descrito. A su derecha, la representación del descriptor del punto, que representa la distribución de los valores del gradiente alrededor del punto, y por ultimo el vector de números reales que codifica este descriptor.

- **Cálculo de correspondencias entre puntos de interés:** Una vez obtenidos los descriptores de la imagen, el último paso es compararlos con los descriptores de la base de datos. La idea general es establecer una función de similitud que a partir de dos descriptores, nos indique su nivel de semejanza.

  Existen varios algoritmos para identificar correspondencias entre descriptores, es decir, buscar el más parecido o *vecino mas cercano (nearest neighbour)*, y se distinguen principalmente por el tipo de distancia o similitud para comparar dos descriptores. Por ejemplo, para descriptores no binarios se suele utilizar la distancia Euclídea, mientras que trabajando con descriptores binarios se suele utilizar la distancia Hamming:

$$d^{HAD}(\mathbf{i},\mathbf{j}) = \sum_{k=0}^{n-1} [i_k \neq j_k] \tag{1}$$

  Donde d^{HAD} es la distancia Hamming entre los códigos binarios $\boldsymbol{i}$, $\boldsymbol{j}$; k es el índice de bit, entre 0 y el número de bits (n), o longitud del descriptor.

  Para mejorar los resultados, se puede establecer un umbral o *threshold* que indique el nivel mínimo de semejanza entre correspondencias. Si una correspondencia no llega a ese nivel, se descarta. Otra opción más robusta y extendida, que es la utilizada en este trabajo, consiste en el uso del ratio entre los dos descriptores más parecidos encontrados. Si la distancia al más cercano es menor que la distancia al segundo más cercano, multiplicada por un factor

reductor, se da la correspondencia como buena. En caso contrario se descarta. El algoritmo es

```
/* Indicador de si la correspondencia es buena                    */
bool accept;
si dnn1 < dnn2 * 0,7 entonces
    accept = true; ;
en otro caso
    accept = false ;
devolver accept;
```

Algoritmo 1: Algoritmo de correspondencias según el ratio entre los dos vecinos más cercanos donde dnn_1 es la distancia al descriptor más cercano, y dnn_2 es la distancia al segundo descriptor más cercano.

Para realizar comparaciones entre muchos puntos de muchas imágenes, se suelen utilizar técnicas aproximadas, basadas en el uso de estructuras de datos que permitan una búsqueda más eficiente, como pueden ser árboles. Por ejemplo FLANN [13] es un método muy extendido para conseguir correspondencias entre *features* de manera eficiente. Sin embargo, debido a que el número de objetos del proyecto es muy pequeño, no se ha considerado necesario implementar este método.

- **Elección del objeto más parecido**: Una vez comparados todos los descriptores de los puntos de interés, la imagen de la base de datos cuyo nivel de semejanza respecto a la imagen enviada por el usuario sea el más alto indicará el nombre del objeto. Esta similitud entre imágenes puede ser sencillamente el número de correspondencias de puntos de interés buenas encontradas, como se utilizará en este trabajo. Sin embargo hay otras técnicas más sofisticadas que tienen en cuenta la distribución espacial de los puntos de interés, o co-ocurrencias de los mismos [14, 15]. No se ha visto necesario implementar ninguno de estos métodos ya que las imágenes del modelo siempre eran limpias y no era parte central del proyecto conseguir un algoritmo de correspondencias robusto, sino una manera rápida de conseguir una similitud. También es posible aplicar un filtro para que el reconocedor sea más robusto. Por ejemplo, calculando restricciones geométricas entre dos vistas (como la homografía o la matriz fundamental [16]) mediante el algoritmo RANSAC [17]. Este algoritmo consigue eliminar los valores atípicos de un modelo, y sería posible filtrar aquellas correspondencias que no se ajustasen a la transformación geométrica. Sin embargo, debido al coste computacional que esto requiere, y a que el reconocedor está pensado para ejecutarse en tiempo real, se ha decidido no implementar este filtro en este primer prototipo.

  En la figura 4 se pueden observar las correspondencias entre un mismo objeto en dos imágenes distintas.

Figura 4: Puntos de interés y correspondencias buenas entre dos imágenes.

Las ventajas que tiene el empleo de *local features*, en concreto usando descriptores binarios, para el reconocimiento de objetos son:

- Robustez a oclusiones, debido a que mientras haya puntos de interés visibles, el reconocedor podrá hacer correspondencias con los objetos de la base de datos (que son imágenes limpias).
- Son eficientes, pudiendo ser utilizados para reconocimiento en tiempo real. Esto es gracias al uso de descriptores binarios.

2.2. Reconocimiento mediante *Deep Learning*

Se ha definido como *Deep Learning* [18] (en español, aprendizaje profundo), a un tipo de aprendizaje automático que cumple las siguientes características:

- Utiliza una cascada de varias capas con unidades de procesamiento no lineales, en las que cada capa sucesiva toma como entrada la salida de la anterior.
- Está basado en el aprendizaje no supervisado de múltiples niveles de características o representaciones de los datos. Las características de más alto nivel derivan de las de más bajo nivel, formando una representación jerárquica.
- Aprende múltiples niveles de representaciones que corresponden a diferentes niveles de abstracción, los cuales forman una jerarquía de conceptos.

En el caso de este proyecto, el método de reconocimiento de objetos mediante *Deep Learning* se basa en el uso de redes neuronales denominadas Redes neuronales convolucionales (CNN, de sus

siglas en inglés). El sistema de reconocimiento basado en estas técnicas tienes dos fases: extracción de *features* de una imagen, y clasificación utilizando esas *features*.

En el trabajo, se parte de una red CNN ya entrenada (dataset de entrenamiento: [20]) para reconocimiento de un conjunto de objetos. Si se quisiera reconocer objetos de ese conjunto de entrenamiento, la red daría directamente una clasificación. En este caso, se quiere utilizar las *features* aprendidas de manera no supervisada durante el entrenamiento de dicha red, para clasificar y reconocer otro tipo de objetos. Este tipo de redes CNN recopilan mucha información general de imágenes (más detalle de cómo funcionan estas redes en el anexo E, y se ha demostrado que las *features* que aprenden se pueden transferir con buenos resultados a otras aplicaciones [4], simplemente utilizando la salida de capas intermedias de la red como *features* de una imagen. Esta es la base del sistema utilizada en el proyecto.

En más detalle, el proceso seguido en este trabajo para reconocer un objeto utilizando una CNN entrenada previamente como extractor de características es:

1. Se obtiene como *feature* de la imagen el resultado de una capa intermedia (fc7 de la red publicada por AlexNet [20]) de las neuronas de clasificación, la cual consiste en 4096 valores *floats* que corresponden con la respuesta combinada de la imagen de entrada a las funciones evaluadas por todas las capas de la red hasta llegar a dicha capa intermedia.

2. Se obtiene el mismo tipo de *feature* para la imagen enviada por el usuario.

3. Se compara el vector obtenido de esa imagen, con todos los vectores obtenidos en las imágenes de la base de datos y, siguiendo el razonamiento de que imágenes similares tendrán valores similares, se elige la más parecida (mediante distancia euclídea). El algoritmo sería el siguiente

```
float min_distance = ∞;
int index = 0;
mientras index < numero(imagenesBD) hacer
    float actual_distance = distancia_euclidea( escena, imagenesBD[index] );
    si actual_distance < min_distance entonces
        min_distance = actual_distance;
    fin
    index++;
fin
devolver index;
```

Algoritmo 2: Algoritmo para la obtención del objeto más parecido mediante distancia euclídea

donde *escena* son los descriptores de la imagen enviada por el usuario, *numero(imagenesBD)* devuelve el número de imágenes que contiene la base de datos e *imagenesBD[index]* se corresponde al descriptor de la imagen de la base de datos con índice *index*.

A diferencia del reconocimiento de objetos basado en *local features*, podemos considerar estas características como *global features*, ya que son descriptores de toda la imagen, no de una zona o

punto concreto. Una de las desventajas de este tipo de descripción de imagen es que, pese a que resulta más compacto (un descriptor por imagen solamente), también resulta más sensible a la oclusión y al ruido de fondo.

Otra desventaja es que requieren de una máquina potente capaz de cargar la red y obtener el resultado, razón por la cual actualmente es prohibitivo para *smartphones* y *tablets*. Actualmente, es motivo de investigación de los principales fabricantes el cómo poder ejecutar *Deep Learning* en un *smartphone* o *tablet*. Este tipo de algoritmos de reconocimiento también posee ciertas ventajas: por ejemplo, permite trabajar por lotes, es decir, procesar varias imágenes al mismo tiempo, sin que esto afecte al rendimiento o también permite aprovechar la GPU de las tarjetas gráficas, lo que supone un plus en rapidez.

3. Diseño del sistema

En este capítulo se hablará del diseño del sistema. Primeramente se mostrará un pequeño esquema del diseño del sistema. A continuación, se comentará la base de datos y después la implementación del resto del sistema, lo cual incluye la descripción de las distintas arquitecturas y de los algoritmos de reconocimiento.

3.1. Diseño de la aplicación

En la figura 5 se muestra un diagrama simplificado del sistema. Esto incluye las dos posibles arquitecturas: local, y cliente-servidor.

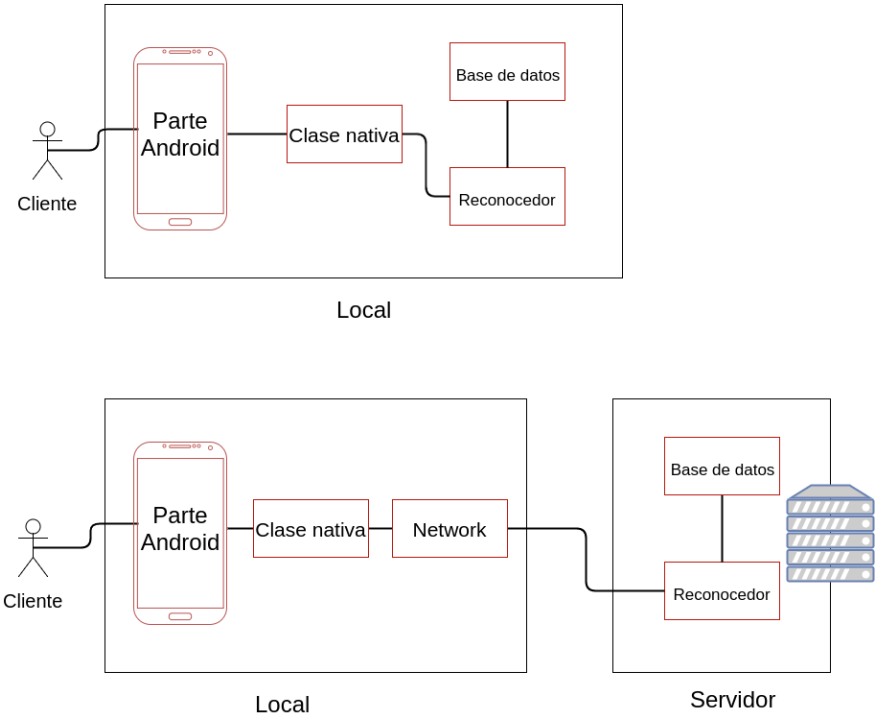


Figura 5: Representación simplificada del sistema con las arquitecturas local y cliente/servidor.

3.2. Descripción de la base de datos

La base de datos final consta de 50 tipos de objetos. Cada objeto está compuesto por su nombre, que servirá de identificador, una lista de imágenes del objeto (representando distintas vistas) con su nombre de vista (arriba, abajo, cara frontal...) y una lista de alérgenos que lo componen, en caso de haber alguno. Además, incluirán un campo extra, que tiene únicamente fines experimentales y no influye en el funcionamiento del prototipo, y es un indicador de si el objeto es "fácil" o "difícil" de reconocer. En la figura 6 se puede observar un diagrama representando a un objeto.

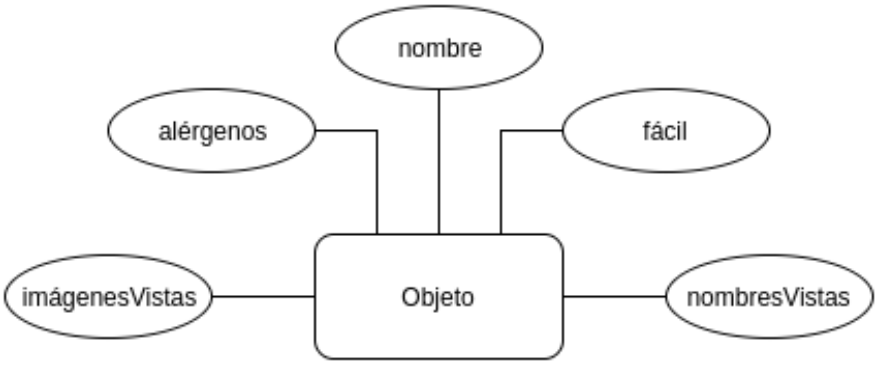


Figura 6: Representación de cada objeto de la base de datos.

En total, en la base de datos están almacenadas estas 54 imágenes de resoluciones variables y obtenidas de diversas páginas de Internet. Esto se ha hecho así para no poblar la base de datos con fotos de la misma cámara con lo que se grabará, haciendo más difícil (y más real) el reconocimiento. La dificultad añadida que plantea el tomar imágenes de Internet es que son en su mayoría imágenes limpias, e incluso a veces generadas por ordenador. La figura 7 muestra todas las imágenes de la colección.

Figura 7: Imágenes que componen la base de datos.

La lista de los posibles alérgenos es la siguiente:

- Altramuces
- Apio
- Cacahuetes
- Gluten
- Dióxido de azufre
- Frutos secos
- Huevos
- Leche
- Marisco
- Moluscos
- Mostaza
- Pescado
- Sésamo
- Soja

La base de datos ha sido programada en C++, cargando los ficheros en memoria al inicio de la aplicación.

3.3. Implementación de la aplicación

3.3.1. Descripción general

La aplicación está diseñada para Android mezclando lenguaje Java (interfaz y acceso a la cámara del móvil) y C++ (procesamiento de la imagen y base de datos). El objetivo de hacerlo así es desacoplar ambas partes para poder reutilizar fácilmente el código correspondiente al procesamiento de la imagen en cualquier otro dispositivo (PC, iOS...) sin tener que modificarlo apenas y poder optimizar esta parte. OpenCV dispone de una librería escrita en Java para Android, pero es menos eficiente y posee menos características y funciones que la escrita en C++.

El sistema estará compuesto por varias clases:

- **Clase nativa**: Es la clase que sirve de puente entre el código Java y el código C++.
- **Reconocedor**: Existen dos reconocedores, uno para reconocer objetos basado en *local features* y otro basado en *Deep Learning*. Ambos poseerán la lista de objetos correspondientes al tipo de reconocedor que son, y sus principales funciones serán cargar la lista de objetos de la base de datos, extraer los descriptores de una imagen, y reconocer un objeto a partir de una imagen. Además, estarán formados por:
  - **Reconocedor basado en *local features*:** un detector y un descriptor de *features*, y un *matcher* de descriptores.
  - **Reconocedor basado en *Deep learning*:** un puntero a la red neuronal ya entrenada, con la cual obtener los descriptores de la imagen.
- **Objeto**: Representa un objeto de la base de datos. Está formado por su nombre, la lista de nombres de las vistas que lo compone, la lista de alérgenos que contiene y si pertenece a los objetos fáciles de reconocer. Además, dependiendo del reconocedor al que correspondan, estarán formados por:
  - **Objetos para reconocedor *local features*:** una lista de puntos de interés de cada una de sus vistas, una lista de matrices de descriptores para cada una de sus vistas, y una lista con las cuatro esquinas de cada una de sus vistas.
  - **Objetos para reconocedor *Deep Learning*:** una lista de descriptores para cada una de sus vistas.
- **Resultado**: representa el resultado de aplicar cualquiera de los dos reconocedores a una imagen. Está compuesto por el nombre y la vista del objeto identificado, el número de puntos de interés y de correspondencias que se han encontrado, la lista de alérgenos que contiene

ese objeto y una lista con las esquinas del rectángulo de la imagen en la que se encuentra el objeto (si lo hay).

- **Network**: utilizado para la comunicación cliente servidor. Está compuesto por un *socket*, la dirección IP y el puerto del servidor, un indicador de si la conexión está activa y otro del tipo de reconocedor que está ejecutando el servidor, además de funciones para enviar y recibir datos del servidor. Más adelante, la subsección 3.3.3 explica el funcionamiento de esta clase.
- **Resultado serializado:** clase generada por *Google Protocol Buffer* que representa un objeto resultado serializado y listo para enviarse entre cliente y servidor.

Además de estas clases, se ha creado otras clases auxiliares para medir tiempos, depurar el programa, testear tipos de datos, y varias clases para ejecutar experimentos, que se detallarán en el capítulo 5.

En el anexo C se encuentra el diagrama de clases del sistema.

3.3.2. Descripción de los algoritmos de reconocimiento

Como ya se ha comentado anteriormente, se han implementado dos reconocedores, uno basado en *local features* y el otro en *Deep Learning*. El funcionamiento de ambos es el mismo, lo único que cambia es la manera que tienen de reconocer objetos. El funcionamiento general sería el siguiente:

1. El reconocedor recibe una imagen.
2. El reconocedor extrae los descriptores de esa imagen.
3. El reconocedor compara los descriptores de la imagen recibida, con los descriptores de las imágenes de la base de datos (que estaban cargados previamente). Aquella imagen de la base cuya correspondencia sea más alta, indicará la vista del objeto al que más se parece. Si la correspondencia es lo suficientemente buena (se pasa un filtro), se marca como objeto encontrado.
4. En caso de haber encontrado objeto, devuelve el resultado (nombre del objeto, nombre de la vista del objeto, lista de alérgenos si los tiene). Si no lo ha encontrado, simplemente indica que no hay objeto.

En el diagrama 8 se puede ver el funcionamiento de este proceso:

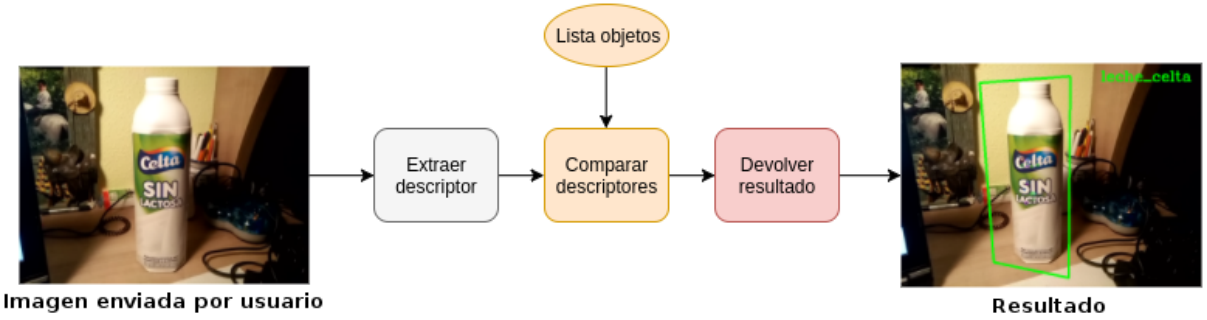


Figura 8: Diagrama del funcionamiento general del reconocedor de objetos.

A continuación, se ahondará en las diferencias de los dos reconocedores:

- **Reconocedor basado en *local features***:

  Este reconocedor tiene guardada la lista de objetos que, como se ha indicado antes, contiene los puntos clave, descriptores y esquinas de cada una de las imágenes que la componen. Cuando recibe la imagen, mediante el algoritmo ORB [11] extrae sus puntos clave y descriptores. A continuación, calcula las correspondencias entre la imagen, y todas las imágenes de la base de datos. Mediante la *distancia de Hamming* se obtienen las correspondencias entre puntos. Sin embargo, para dar por válida una correspondencia, se aplica un filtro basado en el algoritmo *K-vecinos más cercanos* [5], el cual ha sido anteriormente explicado (1). El objeto que más correspondencias buenas tenga con la imagen dada, será reconocido como objeto de la imagen. Sin embargo, aún se aplican dos filtros más. El primero, establecer que haya más de cuatro correspondencias buenas. El segundo, calcular la matriz de homografía entre la imagen y el objeto de la base de datos con más correspondencias buenas. Esta matriz se calcula mediante el algoritmo RANSAC [17], e indica la correspondencia entre cada punto de interés del objeto de la base de datos con su correspondiente en la imagen. De esta manera, y a partir de las esquinas de la imagen del objeto de la base de datos, se pueden obtener los vértices de un polígono que rodee al objeto encontrado en la imagen. El filtro consiste en que si ese polígono resultante no es convexo [6], significa que las correspondencias eran malas, y por lo tanto no existe correspondencia.

  A continuación, la figura 9 muestra un diagrama explicando el proceso:

[5] Explicación `http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_knn/py_knn_understanding/py_knn_understanding.html`

[6] Polígono convexo `https://es.wikipedia.org/wiki/Pol%C3%ADgono_convexo`

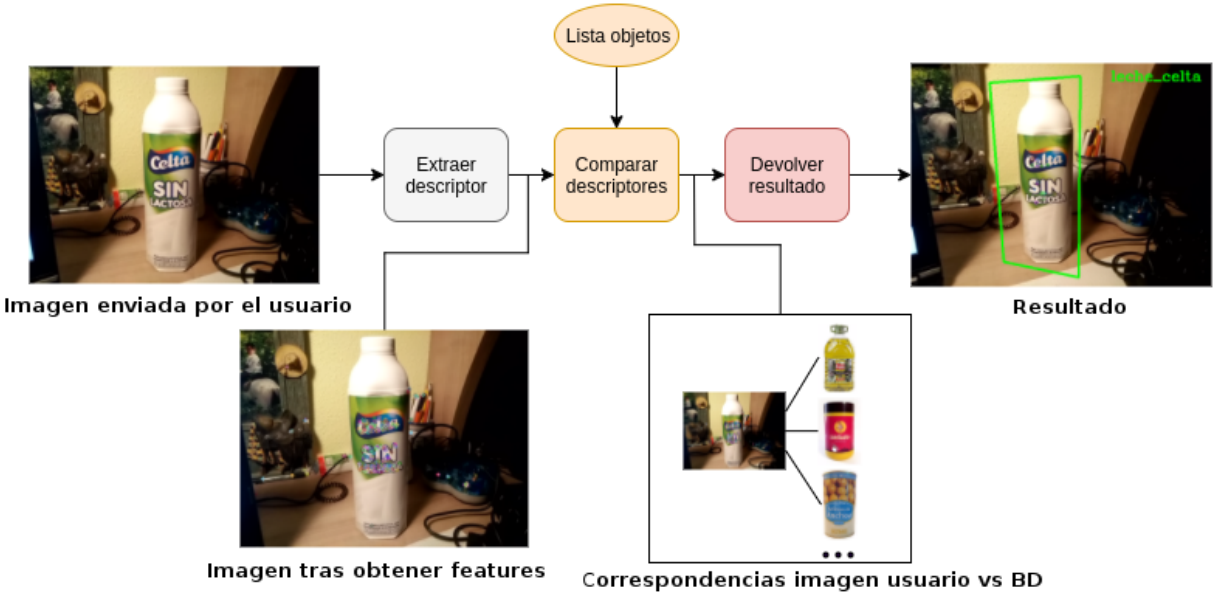


Figura 9: Diagrama del funcionamiento exclusivo del reconocedor de objetos basado en *local features*.

- **Reconocedor basado en *Deep learning***:

  Este reconocedor tiene guardado previamente la lista de objetos que, además de los parámetros comunes (nombre, etc), tienen un descriptor por imagen consistente en una lista de 4096 números reales, valores otorgados por la capa de neuronas fc7 de la red neuronal convolucional. El proceso para reconocer un objeto mediante esta red es extraer ese mismo descriptor de la imagen de entrada. Después, comparar ese vector con los vectores de descriptores de la base de datos, buscando el más parecido bajo el criterio de la distancia euclídea (2). Si la mejor distancia obtenida es menor que cierto valor umbral, se dará por válida la correspondencia. A diferencia del reconocedor de *local features*, este, al estar basado en *global features*, no puede establecer una correspondencia entre puntos de una imagen con la de la base de datos. Por lo tanto, con este reconocedor no se puede obtener el rectángulo rodeando al objeto.

  A continuación, la figura 10 muestra el diagrama explicando este proceso :

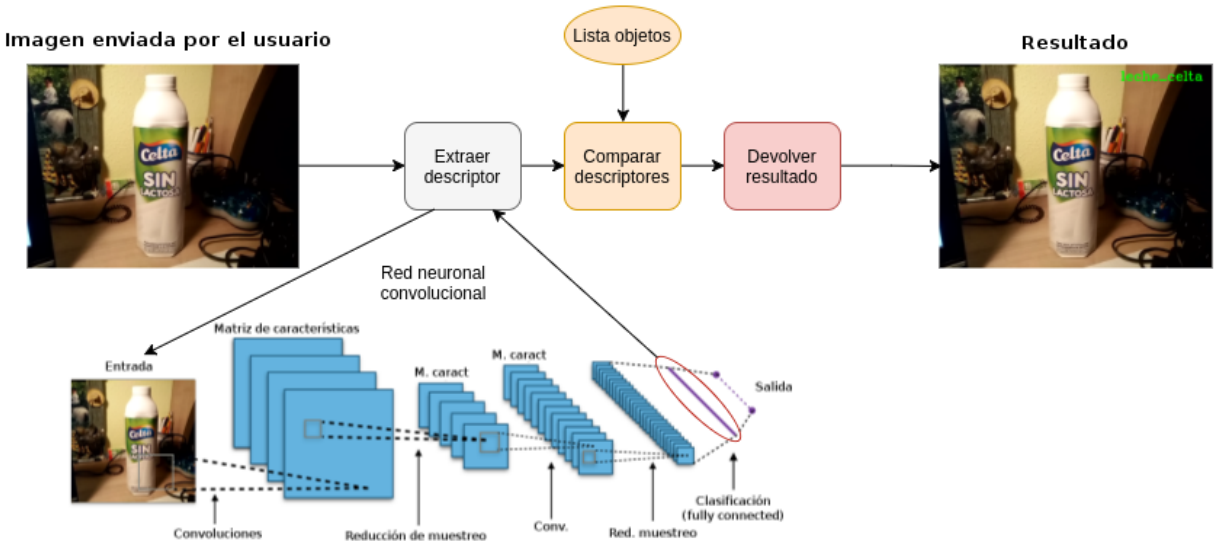


Figura 10: Funcionamiento exclusivo del reconocedor de objetos basado en *Deep Learning*.

El coste en tiempo y la precisión de cada uno se verá en la sección 5 de experimentos y resultados
.

3.3.3. Descripción de las arquitecturas

Como ya se ha dicho anteriormente, el reconocedor basado en *Deep Learning* no puede ser ejecutado en el dispositivo Android, al ser muy exigente en términos de memoria y recursos. Es por ello que se han implementado dos arquitecturas: modo local (*local features*), y modo cliente-servidor (ambas).

- **Modo local**:

  El modo local, como su propio nombre indica, no necesita conexión a Internet y es ejecutado de manera local en el dispositivo Android. Para poder ejecutar este modo y que funcione correctamente, es necesario que las imágenes de la base de datos se encuentren en el almacenamiento interno del dispositivo. El funcionamiento está explicado en la anterior subsección 3.3.2. El coste en tiempo de este modo dependerá en gran medida de la capacidad de cálculo del *smartphone*.

  A continuación, un diagrama mostrando su funcionamiento 11:

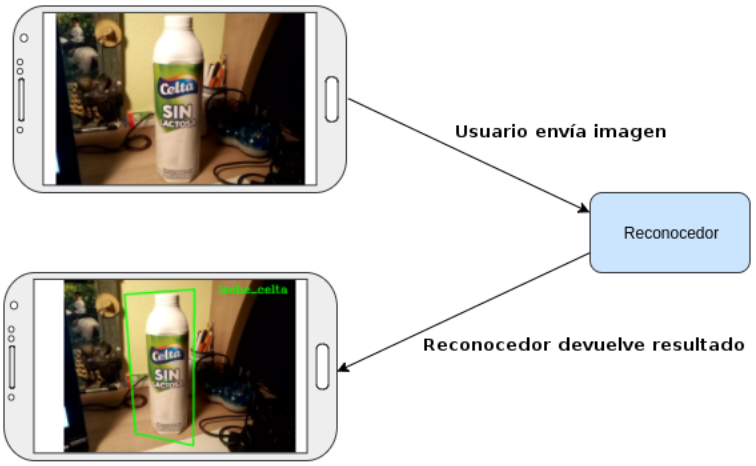


Figura 11: Funcionamiento del reconocedor de objetos en modo local.

- **Modo cliente-servidor**:

  En este modo, existe un servidor con los reconocedores de objetos, y el cliente se limita a enviarle peticiones con las imágenes y a recibir el resultado. Con el fin de que la comunicación sea lo más rápida posible, y siguiendo la idea de que el código pueda ser exportado con facilidad a otros sistemas operativos, se ha decidido implementar la conexión mediante *sockets* [7] bajo el protocolo TCP [8] en C++.

  En modo remoto, el cliente se conecta con el servidor, que está esperando conexiones. Cuando el servidor recibe una conexión, crea un hilo para atender al cliente, y le comunica qué tipo de reconocedor tiene en marcha. Una vez establecida la conexión, el cliente procede a enviarle la imagen al servidor. Ese proceso consiste en comprimir la imagen para reducir la latencia, enviar al servidor el tamaño de la imagen para que sepa cuando dejar de esperar más *bytes*, y enviar la imagen. Si el servidor está en modo *Deep learning*, antes de comprimir la imagen la reduce al tamaño 256x256, que es el tamaño con el que trabaja ese algoritmo. Así, disminuye aún más el tamaño de la imagen, y, por lo tanto, el tiempo de comunicación. Después, el cliente espera respuesta del servidor. Mientras tanto, el servidor una vez haya recibido la imagen, procede a descomprimirla y aplicar el reconocedor correspondiente sobre ella. Una vez obtenga el resultado, mediante los *Protocol Buffers* serializa el resultado y junto a su tamaño en *bytes* se envía de vuelta al cliente. Al recibir el resultado, el cliente lo "deserializa" para finalmente mostrarlo por pantalla. La conexión entre el cliente y el servidor se cierra cuando éste vuelva a la pantalla inicial, minimice o cierre la aplicación.

  La ventaja de este modo es que es más escalable, puesto que el servidor, que es el que realiza los cálculos, puede ser muchísimo más potente que un dispositivo móvil. Además, de esta manera el dispositivo no tiene por qué tener almacenada la base de datos en su interior, con el consecuente ahorro de memoria. Sin embargo, un inconveniente es que la red ha de ser rápida,

[7] Definición de socket: `https://es.wikipedia.org/wiki/Socket_de_Internet`
[8] Protocolo TCP `https://es.wikipedia.org/wiki/Transmission_Control_Protocol`

puesto que el coste de todo el proceso dependerá en gran medida de su velocidad, al igual que de la velocidad de reconocimiento del objeto.

A continuación, el diagrama 12 explica todo este proceso:

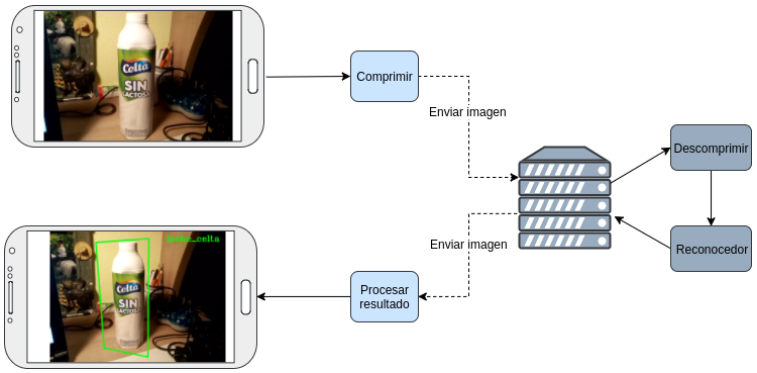


Figura 12: Reconocedor de objetos en modo cliente-servidor.

Por último, se mostrarán dos diagramas de secuencia. Uno referente a la conexión entre cliente y servidor en la figura 13 y otro referente a un envío de imagen en la figura 14.

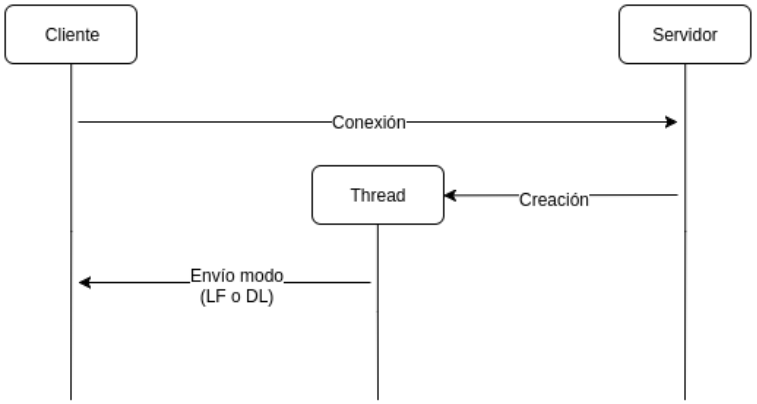


Figura 13: Diagrama de secuencia correspondiente al establecimiento de conexión entre cliente y servidor.

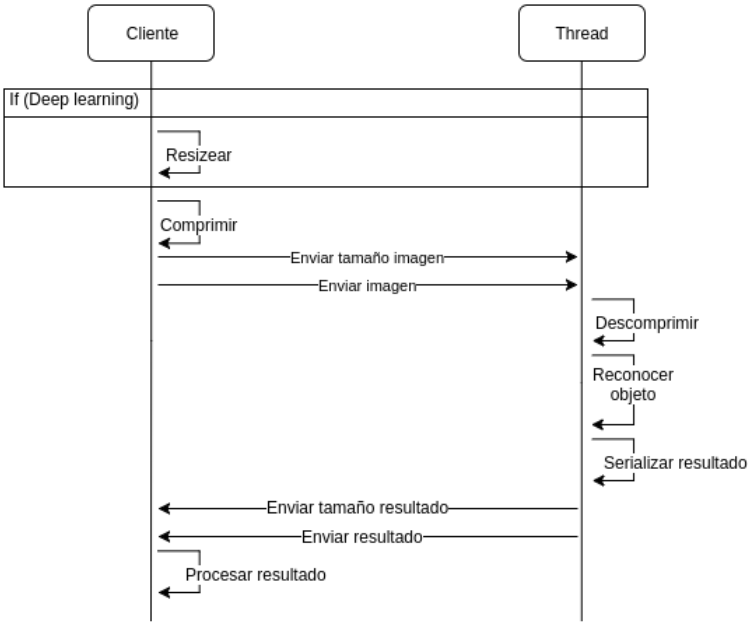


Figura 14: Diagrama de secuencia correspondiente a la comunicación para el procesamiento de una imagen en modo cliente - servidor.

Los costes de tiempo entre una arquitectura y otra se mostrarán en la sección 5 (experimentos y resultados).

4. Implementación del prototipo

En este capítulo se describe de una forma más general el prototipo creado, así como una descripción de su mapa de navegación y sus pantallas.

4.1. Descripción del prototipo

El prototipo es muy simple. Consta de una pantalla con una barra en la cual se encuentra el título, y cuatro botones para las cuatro acciones que se pueden realizar: ejecutar el reconocedor local, ejecutar el reconocedor remoto, ejecutar el test local y ejecutar el test remoto. El tipo de reconocedor remoto dependerá del servidor, será éste el que le indique al cliente cuál está usando.

Para funcionar, la aplicación necesita permisos de lectura/escritura en el almacenamiento (para cargar las imágenes y escribir los resultados de los test, respectivamente); cámara (para acceder a ella) e Internet. En versiones posteriores a Android 6.0, el usuario tendrá que confirmar manualmente estos permisos. En caso contrario, no funcionará la aplicación.

Cuando el usuario la inicia, se cargan los objetos en *background*. Mientras esto sucede, la pantalla aparece en blanco.

En la figura 15 se puede observar la pantalla principal de la aplicación.

Figura 15: Pantalla principal de la aplicación.

Ambos botones del reconocedor, al pulsarlos llevan a la misma pantalla: la visualización de la

cámara. En ella aparecerá también el nombre del objeto reconocido (o si no ha reconocido ninguno, el mensaje "No object"), la lista de alérgenos que contiene, y si el reconocedor está basado en *local features*, un recuadro rodeando al objeto. En caso de que no aparezca ningún tipo de texto, significa que ha habido algún error en la conexión con el servidor, y entonces simplemente se muestra la imagen de cámara en tiempo real.

En cuanto al modo reconocedor remoto, la conexión con el servidor se establece al pulsar el botón, y es cerrada al volver a la pantalla principal, minimizar o cerrar la aplicación.

En la figura 16 se puede observar la pantalla del reconocedor.

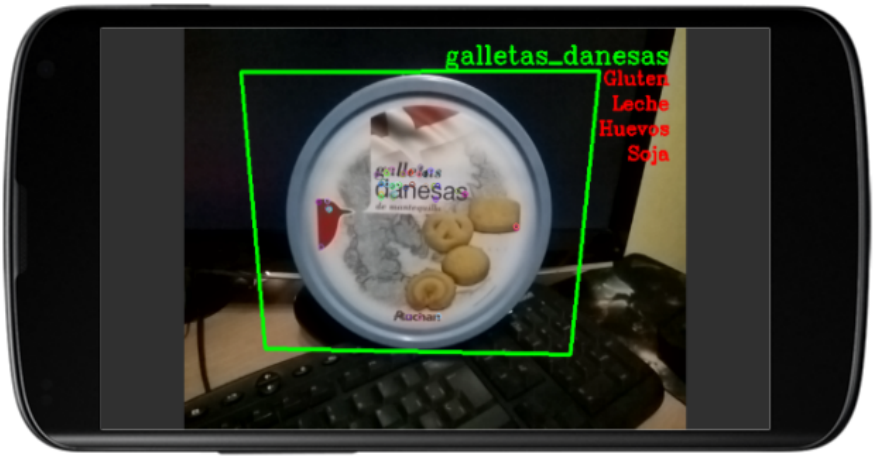


Figura 16: Pantalla del reconocedor de objetos.

Los botones referentes al test no producen ningún cambio visual. Lo que hacen es ejecutar la evaluación del reconocedor correspondiente mediante las imágenes de prueba (esto se comentará más en detalle en la sección 5). Estos botones por lo tanto tienen función únicamente para experimentar.

Se pensó también en incluir un botón en la barra principal que llevase a una lista de alérgenos, donde el usuario pudiera marcar a los que él es alérgico. Después, al reconocer el objeto, se resaltarían en otro color los peligrosos para él. Sin embargo la limitación en tiempo del proyecto hizo que se decidiera priorizar la realización de más experimentos antes que añadir funcionalidades adicionales al prototipo.

En la imagen 17 se puede observar el mapa de navegación. Éste indica que desde la pantalla principal se puede ir a la pantalla del reconocedor, mediante los botones "Cámara" y "Remoto" (reconocedor local y remoto, respectivamente). Desde la pantalla del reconocedor, es posible volver a la pantalla principal, pulsando el botón de "atrás" que incluyen todos los móviles Android.

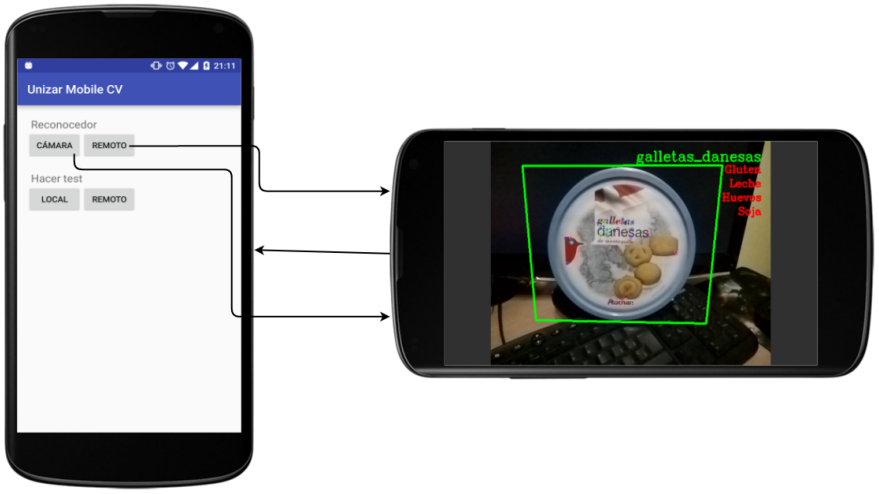


Figura 17: Mapa de navegación del prototipo.

5. Experimentos y resultados

En este capítulo se hablará la preparación de la batería de imágenes de test, los experimentos que se han realizado en el sistema, las optimizaciones que se han llevado a cabo, y finalmente, los resultados obtenidos. En esta sección no se incluyen todas las pruebas realizadas, sólo las más relevantes. El total de pruebas realizadas se encuentra en el Anexo F. Las imágenes de los resultados obtenidos aquí se encuentran en tamaño reducido por razones estéticas. Estas imágenes también se encontrarán en el anexo a tamaño completo.

5.1. *Setup* de la batería de test

Primeramente, se ha creado una base de datos de imágenes de test. Esta base de datos está formada por 150 imágenes, 3 por objeto existente. Para ello, se han tomado fotos de dichos objetos en distintas posiciones (una sosteniendo el objeto con la mano; otra con oclusión, sosteniendo el objeto con la mano y tapándolo parcialmente con el dedo; y otra sobre una mesa de escritorio). En la imagen 18, un ejemplo de las tres fotos de test de un objeto.

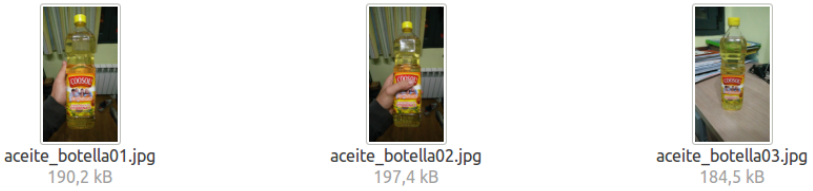


Figura 18: Imágenes de test del objeto "aceite_botella".

Estas imágenes son más grandes que las tomadas por la cámara durante el reconocimiento en tiempo real (640x1137 vs 640x480). Sin embargo, para hacer las pruebas sirven, ya que sólo interesa saber las diferencias de tiempos de las distintas optimizaciones.

Para la implementación de los experimentos se han creado varias clases nombradas anteriormente en el capítulo 3.3.1. Estas clases son:

- **TestImage**: Representan un objeto de test. Este objeto estará formado por el nombre del objeto que representa, el nombre del fichero de su imagen, la imagen en formato *Mat* en blanco y negro, y la imagen en el mismo formato en color.
- **Tester**: Es la clase que se encarga de ejecutar los experimentos. Posee como atributo una lista de objetos de test, con los cuales evaluará la precisión de los algoritmos reconocedores. Su funcionamiento consiste en:

  1. Crear una matriz de confusión vacía [9]. Una matriz de confusión es una herramienta que permite evaluar el desempeño de los reconocedores. Cada fila representa un objeto real,

[9] Matriz de confusión: `https://es.wikipedia.org/wiki/Matriz_de_confusi%C3%B3n`

y cada columna un objeto predicho. Para que una matriz de confusión indique que el reconocedor es 100 % perfecto para los objetos de test, debería tener sólo ocupada la diagonal. Esto indicaría que para cada objeto, en efecto predice ese objeto y no otro.

2. Pasar cada imagen de test por el reconocedor correspondiente y obtener el resultado predicho.
3. Comparar el nombre del objeto del resultado dado por el reconocedor con el resultado real.
4. Colocar el resultado de la comparación en el lugar correspondiente de la matriz. Si el reconocedor ha acertado, éste debería colocarse en la diagonal.
5. Repetirá todo el proceso 5 iteraciones para obtener la media de esas 5 ejecuciones.

Además, se miden los tiempos que tarda el reconocedor en ejecutarse, y se escribe la matriz de confusión, junto con otros datos relevantes (media de puntos, media de *matches*, tiempo...) en 4 ficheros de texto. Uno de ellos contiene los datos de test de toda la batería de imágenes, y los otros tres, las baterías de objetos sin oclusión, objetos con oclusión y objetos en la mesa.

- **Utilidades**: Esta clase contiene métodos utilizados a lo largo del proyecto de propósito general, como por ejemplo escribir por consola mensajes o convertir variables de distintos tipos a *string*.
- **Timer**: Clase utilizada para almacenar tiempos de ejecución identificados por etiquetas. De esta manera se pueden medir tiempos de determinadas partes del código, y almacenarlas independientemente del resto de tiempos guardados. También permite imprimir todos los tiempos almacenados, con junto con el número de ejecuciones de esa parte de código, la media de tiempo para ese número de ejecuciones y la desviación estándar.

Además, para poder interpretar ese resultado, se ha creado un *script* en Python, que a partir de la salida generada por el *Tester*, genera tres imágenes: la matriz de confusión, la tabla de verdaderos/falsos positivos - verdaderos/falsos negativos [10] y la tabla de precisión, exactitud y exhaustividad [11] (*recall*).

5.2. Elección del algoritmo de reconocimiento basado en *local features*

Una vez creada la batería de imágenes de test, el siguiente paso fue investigar qué algoritmo de reconocimiento basado en *local features* era más adecuado para el proyecto. Debido a su naturaleza en tiempo real, el algoritmo debía cumplir esta característica: **rápido, pero preciso**. Se eligieron los siguientes descriptores para su evaluación: ORB, BRISK y FAST (este último al ser sólo detector, se combinó con ORB y BRISK para obtener los descriptores). Además, de los descriptores ORB y BRISK se probaron distintas configuraciones aparte de las configuraciones por defecto en OpenCV. La razón de escoger estos algoritmos únicamente es que se trata de descriptores binarios, mucho

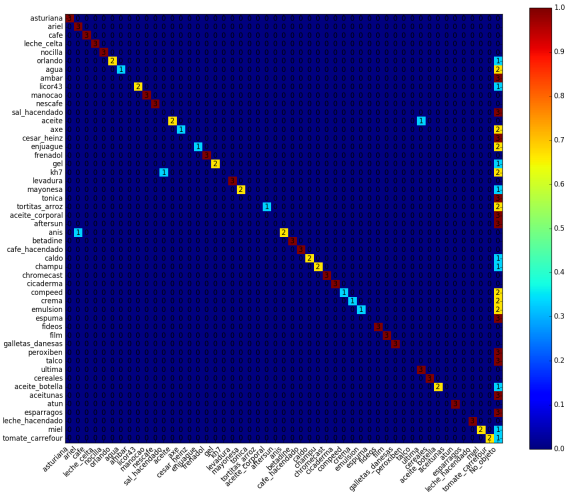
[10] Explicación verdadero/falso positivo/negativo: `https://es.wikipedia.org/wiki/Curva_ROC`
[11] Precisión y recall: `https://es.wikipedia.org/wiki/Precisi%C3%B3n_y_exhaustividad`

más rápidos que no binarios como SIFT o SURF. Información al respecto se puede encontrar en numerosos artículos [11] [21].

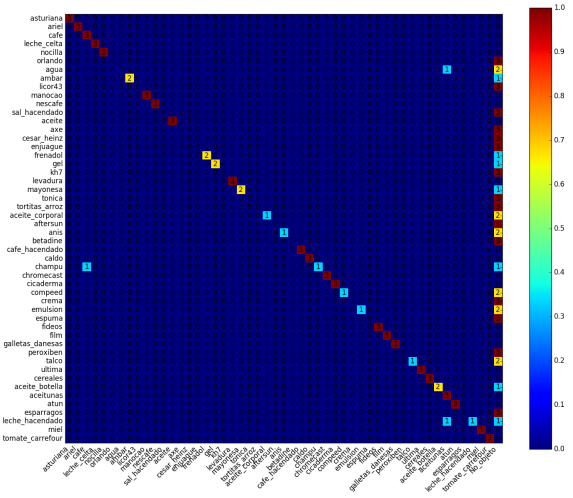
Por último, para desarrollar los experimentos se han utilizado tres máquinas distintas:

- ***Smartphone* OnePlus X**: Sistema operativo Android. 3GB RAM. Procesador 2.3GHz Qualcomm Snapdragon 801 (cuatro núcleos). [12]
- **Portátil MSI GE62 6QE**: Sistema operativo Ubuntu 16.04. 8GB RAM. Procesador Intel Core i7-6700HQ (3.50 GHz). Durante el proyecto, se referirá a esta máquina como "portátil". [13]
- **Servidor remoto**: Sistema operativo CentOS 6.8. 64GB RAM. Procesador Intel Core i7-6700K (4 GHz). 2x GPU Nvidia Titan X [14]. CUDA instalado [15]. Durante el proyecto, se referirá a esta máquina como "servidor".

Primero se ha evaluado ORB y BRISK. El resultado de la evaluación entre ambos, ejecutados en el *smartphone*, se observa en la figura 19.



(a) ORB (72)
Tiempo: 48.2 segs
Aciertos: 88
Fallos: 4
No encontrados: 58
Media puntos: 500
Media *matches* buenos: 27

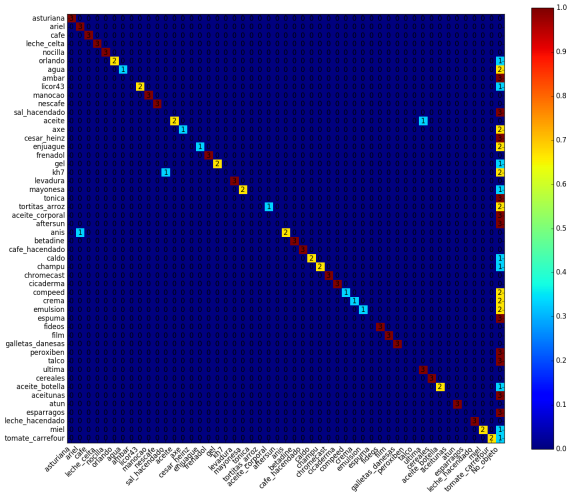


(b) BRISK (96)
Tiempo: 502.7 segs
Aciertos: 86
Fallos: 3
No encontrados: 61
Media puntos: 1855
Media *matches* buenos: 63

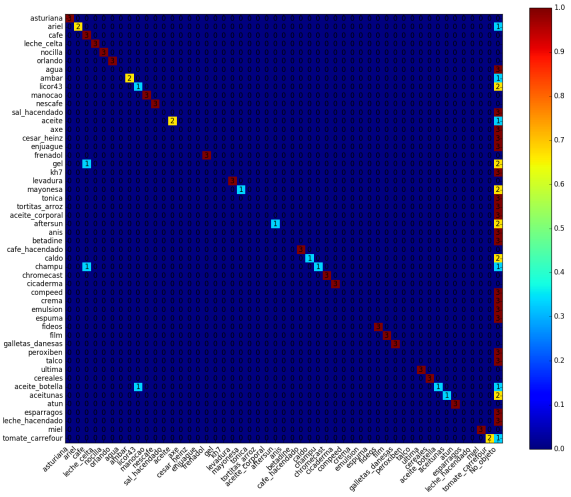
Figura 19: ORB vs BRISK (Android)

[12] OnePlus X `https://oneplus.net/es/x`
[13] MSI GE62 6QE `https://www.msi.com/Laptop/GE62-6QE-Apache-Pro-.html#hero-overview`
[14] Nvidia Titan X `http://www.nvidia.es/graphics-cards/geforce/pascal/titan-x/`
[15] CUDA `http://www.nvidia.es/object/cuda-parallel-computing-es.html`

ORB obtiene mejor precisión que BRISK (88 aciertos vs 86) con un menor tiempo de ejecución. Esta gran diferencia de tiempo se debe a que BRISK encuentra muchos más puntos, y como el algoritmo para calcular las correspondencias es linear para el número de puntos, incrementa mucho el tiempo de ejecución. Para intentar bajar este tiempo, se han retocado los valores por defecto que presenta el descriptor BRISK, aumentando un umbral relacionado con el algoritmo AGAST [22], que tiene que ver con la detección de esquinas. A mayor umbral, menor número de puntos, siendo por defecto 30. En la imagen 20 se observan los resultados de los mismos reconocedores, salvo que ahora BRISK tendrá un umbral de 85.



(a) ORB (72)
Tiempo: 48.2 segs
Aciertos: 88
Fallos: 4
No encontrados: 58
Media puntos: 500
Media *matches* buenos: 27

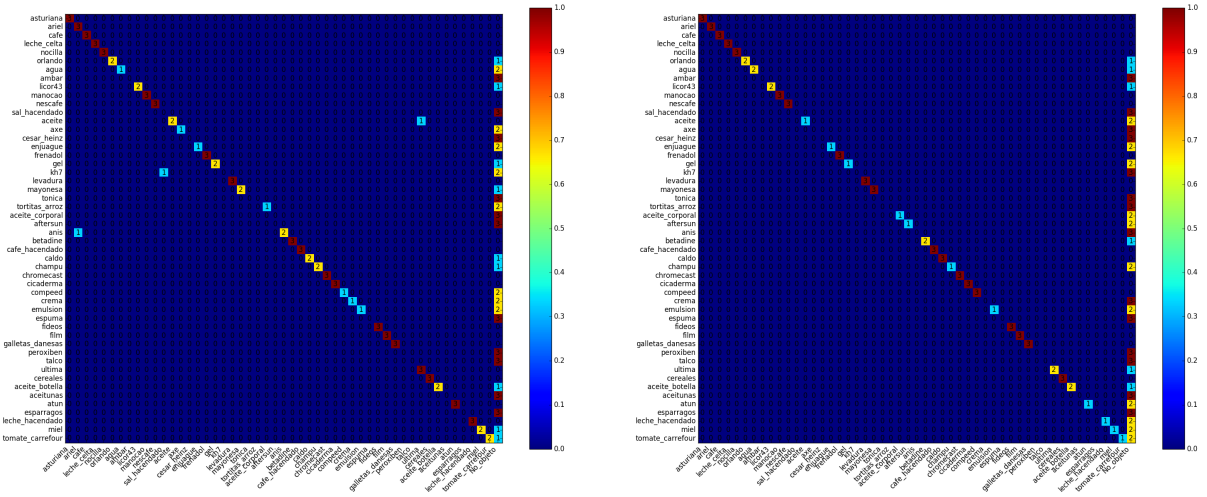


(b) BRISK (umbral 85) (102)
Tiempo: 44.4 segs
Aciertos: 72
Fallos: 3
No encontrados: 75
Media puntos: 331
Media *matches* buenos: 23

Figura 20: ORB vs BRISK (umbral 85) (Android)

Se observa que el tiempo ha mejorado considerablemente, debido a la disminución de puntos de interés detectados. Sin embargo, se ha perdido precisión, y ésta sigue siendo menor que la del descriptor ORB (72 aciertos vs 88), por lo que se decidió por este último. Para concluir con la elección del reconocedor que se utilizará a lo largo del proyecto, se ha decidido comparar dos configuraciones distintas del descriptor ORB. Una la de por defecto, y la otra cambiando el número de puntos que ORB obtiene internamente de otro descriptor [16], para ver si mejora la precisión. En la imagen 21 se puede observar el resultado:

[16] Ver parámetro WTA_K aquí `http://docs.opencv.org/trunk/db/d95/classcv_1_1ORB.html`



(a) ORB (72)
Tiempo: 48.2 segs
Aciertos: 88
Fallos: 4
No encontrados: 58
Media puntos: 500
Media *matches* buenos: 27

(b) ORB (WTA_K 4) (87)
Tiempo: 71.3 segs
Aciertos: 80
Fallos: 0
No encontrados: 70
Media puntos: 500
Media *matches* buenos: 30

Figura 21: ORB vs ORB (WTA_K 4) (Android)

Se observa que no solo el tiempo es peor, sino que la precisión también lo es (88 vs 80 aciertos). Pese a que no da falsos positivos (no reconoce erróneamente ningún objeto como otro), la diferencia de tiempo no compensa.

Las pruebas del detector FAST junto con los descriptores BRISK y ORB han resultado muy malas en cuestión de tiempo y precisión. Es por ello que no se incluyen en esta sección. Estos resultados se encuentran en el anexo F, sección F.1.

En resumen, vistos los resultados obtenidos, se ha escogido ORB como descriptor final para el reconocedor basado en *local features*.

5.3. Experimentos sobre la configuración del modo cliente-servidor

Una vez elegido el descriptor, el siguiente paso era la implementación de ese mismo reconocedor pero en modo cliente-servidor. Como ya se ha comentado en la sección 3.3.3, la comunicación ha sido implementada mediante *sockets* y protocolo TCP. En la primera versión, el cliente enviaba al servidor la imagen sin comprimir, el servidor la procesaba, y enviaba al cliente otra imagen sin comprimir con el resultado por pantalla. Esto era extremadamente ineficiente en términos de tiempo, debido a la comunicación red. Para la segunda versión, el cliente enviaba un entero indicando el tamaño de la imagen codificada en *jpeg*, así como la imagen codificada; el servidor descodificaba esa imagen, la procesaba, y la imagen resultado la codificaba de nuevo y se la enviaba al cliente.

Haciendo esto se logró mejorar en gran medida el tiempo de comunicación, siendo ya aceptable para usarse en tiempo real. Por último, para la versión final se decidió usar los *Protocol Buffers* de Google, un mecanismo que permite serializar objetos. El proceso ahora consistía en que el cliente enviaba la imagen comprimida al servidor, el servidor la descomprimía y procesaba, pero en vez de devolver la imagen procesada, devolvía el objeto resultado, haciendo uso de estos *Protocol Buffers*. El cliente procesaba el resultado y lo mostraba por pantalla. El resultado: reducir enormemente el tamaño del objeto a enviar desde el servidor al cliente. En esta tabla de tiempos 2 se pueden observar las diferencias de tiempos y tamaños de objetos enviados durante las distintas versiones en la aplicación real (imágenes obtenidas directamente de la cámara). (cliente: Android, servidor: portátil, datos: cámara, red: 100Mbps misma red, calidad *jpeg*: 90):

Versión	**Tiempo total (segs.) (media 100 iter.)**	**Tiempo reconocedor (segs.) (media 100 iter.)**	**Tamaño obj enviado (kB)**	**Tamaño obj recibido (kB)**
Local	0.326	0.326	0	0
Sin comprimir imagen ni al enviar ni al recibir	1.025	0.122	921.6	921.6
Comprimiendo imagen a enviar y al recibir	0.457	0.122	≈175.6	≈175.6
Comprimiendo imagen a enviar, recibiendo protobuf	0.328	0.122	≈175.6	≈0.067

Tabla 2: Comparativa de tiempos y tamaños de objetos enviados/recibidos de las distintas versiones realizadas. (cliente: Android, servidor: portátil, datos: cámara, red: 100Mbps misma red)

El nivel de compresión de la imagen era un factor a tener en cuenta, puesto que puede afectar a la precisión del reconocimiento. Se comprobaron los resultados para la batería de imágenes de test, comprimiéndola con distintos factores: 100 (menos pérdida de calidad posible), 90 y 80. Los resultados que se obtuvieron se pueden observar en la tabla 3 (imágenes en el anexo F.2).

Calidad de imagen jpeg	Aciertos	Fallos	No reconocidos
100 % (78)	88	2	60
90 % (81)	92	2	56
80 % (84)	86	2	62

Tabla 3: Comparación del reconocedor ORB con imágenes comprimidas con distintas calidades. (cliente: Android, servidor: portátil, datos: test)

Se observa que el número de aciertos es mayor en el reconocedor con calidad de compresión 90 %. Esto probablemente se deba a que esa pérdida de calidad en estos casos está haciendo que el reconocedor detecte mejores puntos de interés para identificar ese objeto. Además, el tiempo también mejora considerablemente (ver tiempos en anexo F.4, tabla 15). Con la calidad de la imagen al 80 % se nota un descenso notorio en la precisión del reconocedor, pese a que el tiempo sea menor también. Es por ello que se ha decidido optar por implementar un nivel de compresión con calidad 90 %.

5.4. Experimentos sobre la configuración del reconocedor *Deep Learning*

En cuanto al reconocimiento basado en *Deep Learning* también se han realizado varias pruebas sobre distintas configuraciones del mismo. Se parte de una red neuronal ya entrenada y, como se ha indicado en la sección 3.3.2, el procedimiento consiste en extraer los descriptores de la imagen y compararlos con los descriptores previamente extraídos de los objetos. La capa de la que se extraen los descriptores, como también se ha dicho, es la capa denominada fc7. El primer paso fue ejecutar la batería de imágenes de pruebas con este reconocedor. El resultado fue el que se ve en la imagen 22:

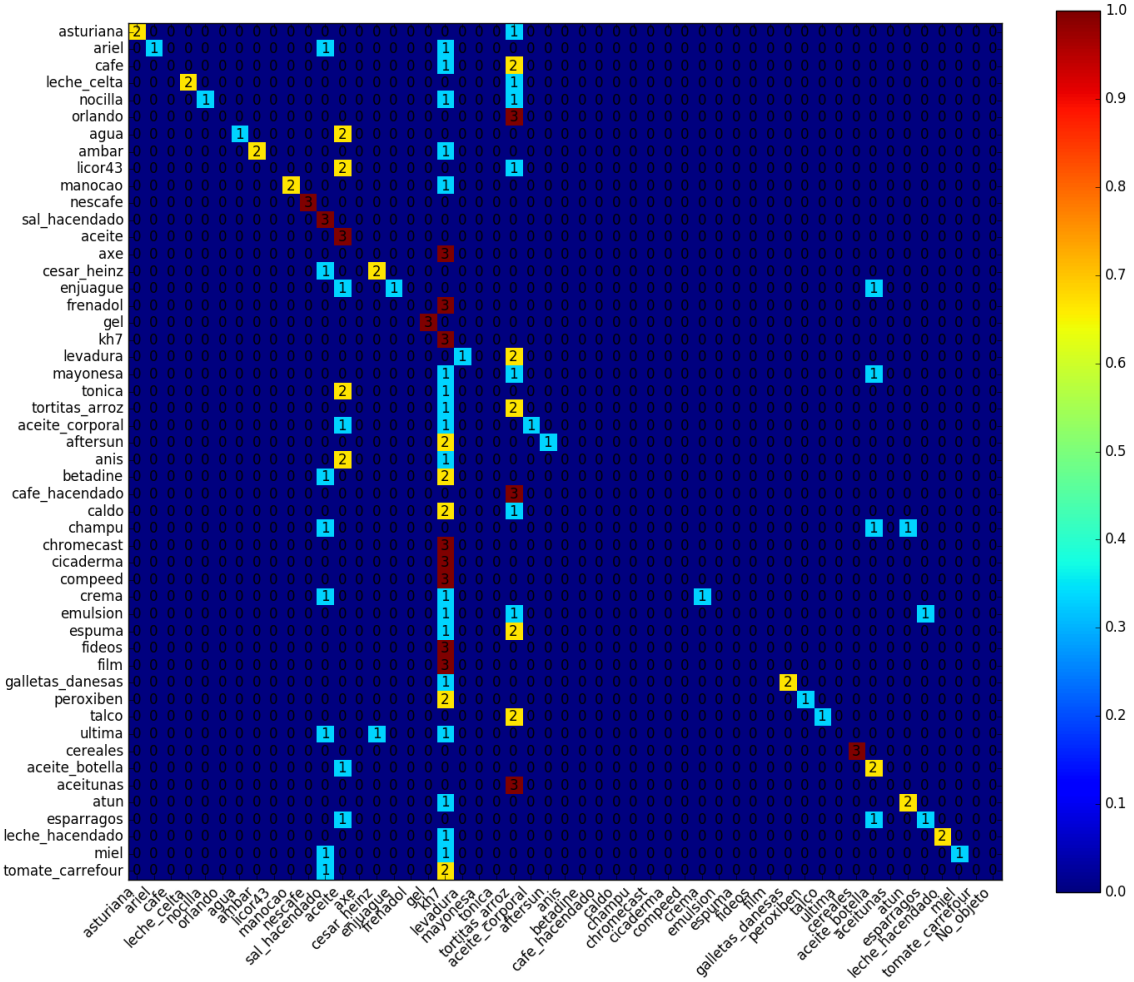


Figura 22: Matriz de confusión obtenida del reconocedor *Deep Learning*. (108)

Tal y como se puede observar, el resultado no es nada bueno. Además, se ve como hay objetos "predominantes", los cuales el reconocedor predice en la mayoría de las ocasiones, aún cuando no se corresponde al objeto real. Para analizar más a fondo este problema, se separó esta matriz de confusión en otras tres: una para los objetos sin oclusión, otra para los objetos con oclusión, y otra para los objetos en la mesa. En la tabla 4 se pueden ver los resultados del reconocimiento de esos tres tipos de imágenes de test (las imágenes de las matrices se encuentran en la sección F.3 del

anexo F).

Tipo de imagen	Aciertos	Fallos
Sin oclusión (111)	26	24
Con oclusión (114)	13	37
Objeto en la mesa (117)	11	39

Tabla 4: Resultados de los tres tipos de imágenes de test comparando descriptores capa fc7

Se puede ver cómo los mejores resultados se obtienen con la perspectiva sin oclusión. Sin embargo, con la perspectiva con oclusión y con objetos en la mesa, la precisión del reconocedor es mucho menor. Esto concuerda con la teoría de este tipo de reconocedores: al estar basados en *global features*, son muy afectados por la oclusión, ya que no distinguen fondo de objeto y obtienen sus descriptores a partir de toda la imagen entera.

También, para ver si se mejoraban los resultados, se programó el reconocedor para utilizar los descriptores de la capa fc8 en lugar de la fc7, es decir, la siguiente capa a la que se estaba usando. El resultado se puede visualizar en la tabla 5 (imágenes en la sección F.3 del mismo anexo, figura 120).

Tipo de imagen	Aciertos	Fallos
Sin oclusión (123)	26	24
Con oclusión (126)	17	33
Objeto en la mesa (129)	13	37

Tabla 5: Resultados de los tres tipos de imágenes de test comparando descriptores capa fc8

Se observa que los resultados para las imágenes sin oclusión son los mismos, pero sin embargo existe una pequeña mejora en los otros dos tipos. Sin embargo, como resultado global, sigue sin ser bueno.

Otra hipótesis que se pensó como posible causa de que el reconocedor tuviese tan poca precisión, es la calidad de las imágenes, tanto las de la base de datos, como las de test. La razón de ello es que al tener resoluciones tan variables (algunas son cuadradas, otras horizontales, otras verticales muy alargadas...) y el hecho de que este algoritmo antes de tratar con ellas, las escala a tamaño 256x256, se deformen los objetos y se pierda mucha calidad en el proceso.

Para comprobar si así mejoraban los resultados, se creó una base de datos "alternativa", con imágenes de la base de datos cuadradas, y algunas de las imágenes de test limpias (fondo blanco, objeto en el centro y cuadradas también). Se han seleccionado 5 objetos al azar, a los cuales se les ha hecho fotos limpias. Las imágenes se ven en la figura 23:

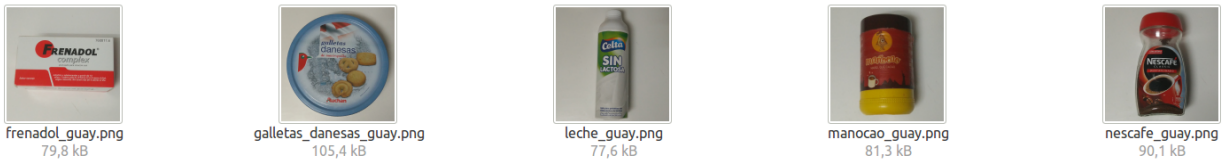


Figura 23: Imágenes de objetos limpias.

Al ejecutar el reconocedor, se obtienen los resultados mostrados en la tabla 6

	Aciertos	Fallos
Objetos limpios	5	0

Tabla 6: Resultado de pasar las imágenes "limpias" por el reconocedor *Deep Learning* (156)

Se puede comprobar como el reconocedor ha acertado todos los objetos. Por lo tanto, es asumible pensar que el problema está el formato de las imágenes, ya que cuadrando todas las imágenes de la base de datos, y usando imágenes limpias como imágenes de test, el reconocedor parece no fallar.

En cuanto a tiempos, se realizaron dos optimizaciones. La primera de ellas tiene que ver con la compresión de la imagen. Ya que el algoritmo *Deep Learning* requiere que la imagen tenga tamaño 256x256, y que la red es uno de los principales cuellos de botella en el modo cliente-servidor, se ha decidido comprimir la imagen en el lado del cliente, enviando una imagen mucho menor al servidor ($\approx$175.6 kB$\rightarrow\approx$12.5 kB). La otra optimización tiene que ver con el modo de procesamiento que utiliza Caffe para ejecutar el algoritmo. Puede realizar los calculos con la CPU o con la GPU siendo el resultado de ambas configuraciones el mostrado en la siguiente tabla 7:

Configuración	CPU (segs)	GPU (segs)
Cargar imágenes de la base de datos	4.95	9.3
Cargar red	0.23	0.39
Obtener descriptores de una imagen (reconocedor)	0.13	0.006
Total reconocedor	0.33	0.20

Tabla 7: Diferencia de tiempos ejecutando reconocedor en modo CPU o GPU (cliente: Android, servidor: servidor)

Como se puede ver, leer las imágenes de la base de datos, obtener sus descriptores y guardarlos en memoria, tarda casi el doble usando GPU en vez de CPU. También tarda más el cargar la red en memoria. Sin embargo, obtener los descriptores de la imagen que el cliente envía al servidor va mucho más rápido utilizando la GPU. Al ser las dos primeras operaciones que se ejecutarán al iniciar el servidor únicamente, y ser esta última la más crítica, pues es la que tiene que ejecutarse en "tiempo real", se ha dejado la configuración usando la GPU.

5.5. Evaluación de la memoria

Se ha comparado también la memoria que utiliza cada uno de los descriptores. El resultado ha sido:

- **Reconocedor basado en *local features***:
  - **Lista de objetos:** $\approx$ 12 MB
- **Reconocedor basado en *Deep learning***
  - **Lista de objetos:** $\approx$ 570 MB
  - **Red:** $\approx$ 220 MB

Los resultados han sido medidos calculando las diferencias entre la memoria utilizada por el servidor cuando no se cargaba ningún reconocedor, con la memoria utilizada cuando se cargaban cada reconocedor y la red. Aquí se ratifica que el algoritmo *Deep Learning* es demasiado exigente en términos de prestaciones para que sea viable ejecutarlo en local, puesto que ya sólo en términos de memoria es inasumible ya que sólo los terminales de gama alta tienen más de un gigabyte de memoria.

5.6. Comparativa de tiempos finales

En esta subsección se mostrarán y compararán los tiempos finales de los tres tipos de reconocedores: los reconocedores basados en *local features* (local y cliente servidor) y el reconocedor basado en *Deep Learning*. Así mismo detallarán los tiempos de las partes de cada uno.

- **Modo local basado en *local features*.**
  - **Cliente:**
    - **Cargar los objetos:** 6.23 segs.
    - **Procesar la imagen:** 0.325 segs. Descomposición en la tabla 8 y figura 24.

	Tiempo medio (segs) (100 iter.)	**Desviación típica**
Extraer *features*	0.095	0.02
Hacer *matches*	0.2	0.03
Procesar resultado	0.13	0.002
Total	**0.325**	**0.04**

Tabla 8: Coste en tiempo del reconocedor en local

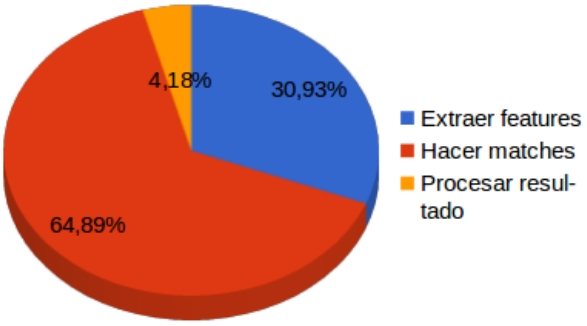


Figura 24: Porcentaje de tiempos del cliente en modo local con reconocedor basado en *local features*.

- **Modo cliente-servidor basado en *local features*.**
  - **Cliente:**
    - **Proceso total:** 0.325 segs. Descomposición en la tabla 9 y la figura 25.

	Tiempo medio (segs) (100 iter.)	**Desviación típica**
Codificar y enviar	**0.057**	**0.01**
-Codificar	0.056	0.01
-Enviar	0.001	0.001
Esperar, recibir y procesar resultado	**0.259**	**0.09**
-Recibir	0.257	0.09
-Deserializar resultado	0	0
-Procesar resultado	0.001	0.001
Total	**0.325**	0.09

Tabla 9: Coste en tiempo del cliente en modo cliente-servidor con reconocedor basado en *local features*.

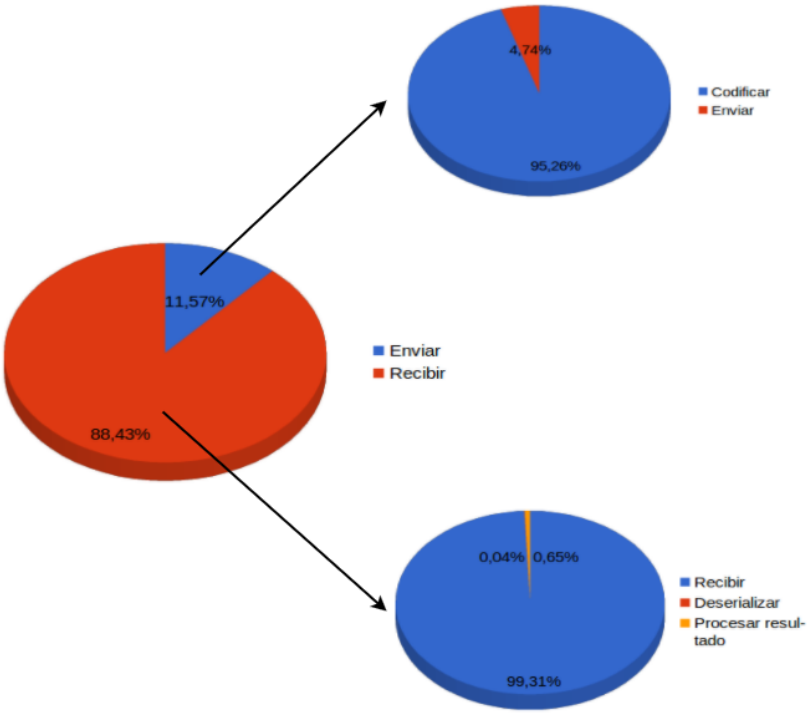


Figura 25: Porcentaje de tiempos del cliente en modo cliente-servidor con reconocedor basado en *local features*.

- **Servidor:**
  - **Cargar objetos:** 0.618 segs.
  - **Recibir, procesar y devolver la imagen:** 0.355 segs. Descomposición en la tabla 10 y la figura 26

	Tiempo medio (segs) (100 iter.)	**Desviación típica**
Recibir imagen	**0.2**	**0.08**
Decodificar	**0.005**	0.002
Procesar total	**0.149**	0.02
-Extracción de *features*	0.006	0
-Hacer *matches*	0.142	0.02
-Procesar resultado	0	0
Serializar	**0**	**0**
Enviar	**0**	**0**
Total	**0.355**	**0.08**

Tabla 10: Coste en tiempo del servidor en modo cliente-servidor con reconocedor basado en *local features*.

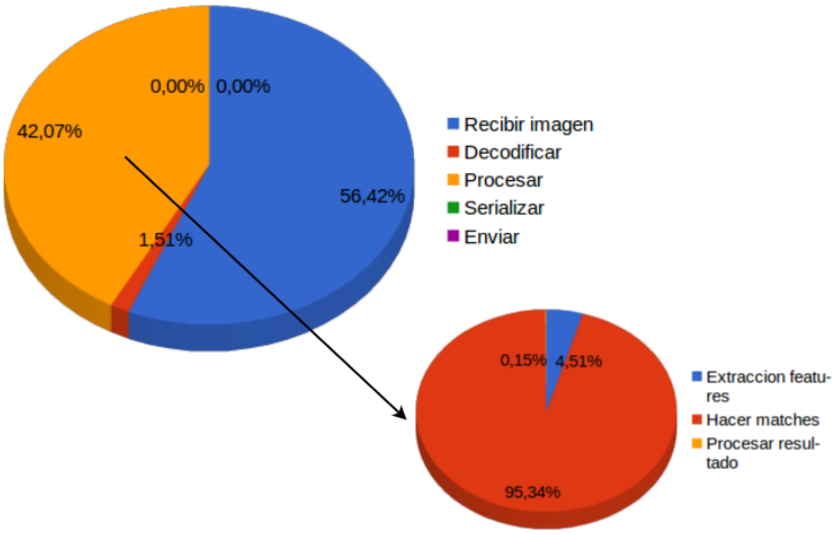


Figura 26: Porcentaje de tiempos del servidor en modo cliente-servidor con reconocedor basado en *local features*.

- **Modo cliente-servidor basado en *Deep Learning*.**
  - **Cliente:**
    - **Proceso total:** 0.346 segs. Descomposición en la tabla 11 y la figura 27.

	Tiempo medio (segs) (100 iter.)	**Desviación típica**
Redimensionar, codificar y enviar	**0.04**	**0.005**
-Redimensionar	0.01	0.003
-Codificar	0.028	0.004
-Enviar	0.001	0.001
Esperar, recibir y procesar resultado	**0.306**	**0.03**
-Recibir	0.304	0.03
-Deserializar resultado	0	0
-Procesar resultado	0.002	0.003
Total	**0.346**	**0.03**

Tabla 11: Coste en tiempo del cliente en modo cliente-servidor con reconocedor basado en *Deep Learning*.

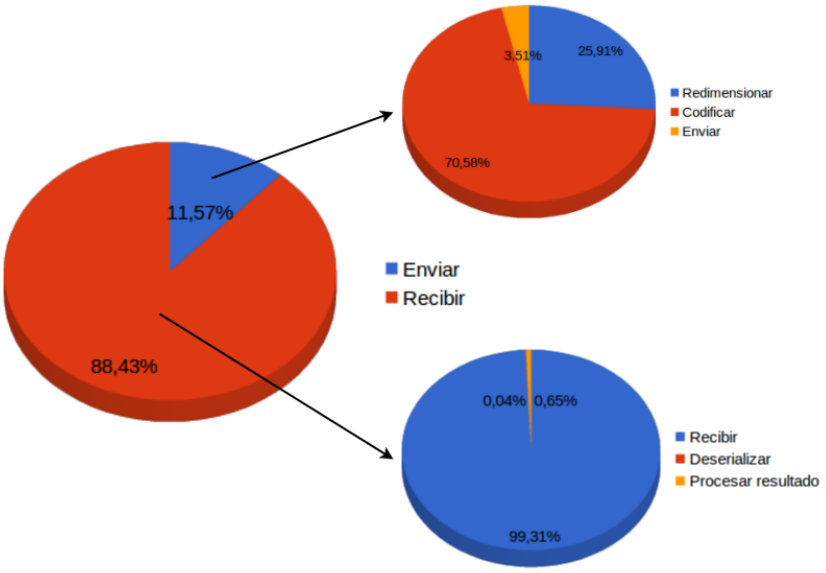


Figura 27: Porcentaje de tiempos del cliente en modo cliente-servidor con reconocedor basado en *Deep Learning*.

- **Servidor:**
  - **Cargar objetos:** 0.618 segs.
  - **Recibir, procesar y devolver la imagen:** 0.375 segs. Descomposición en la tabla 12 y la figura 28

	Tiempo medio (segs) (100 iter.)	**Desviación típica**
Recibir imagen	**0.167**	**0.04**
Decodificar	**0.002**	**0.001**
Procesar total	**0.205**	**0.006**
-Extracción de *features*	0.006	0.005
-Hacer *matches*	0.198	0.004
-Procesar resultado	0	0
Serializar	**0**	**0**
Enviar	**0**	**0**
Total	**0.375**	**0.04**

Tabla 12: Coste en tiempo del servidor en modo cliente-servidor con reconocedor basado en *Deep Learning*.

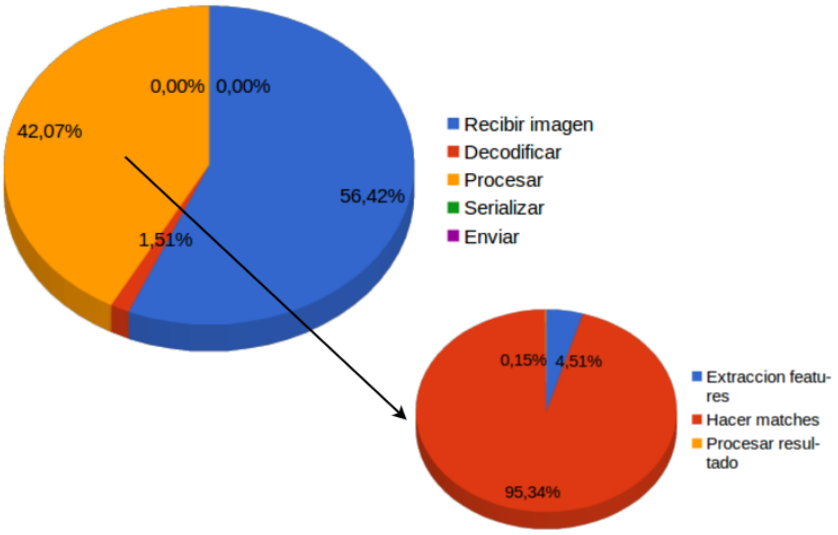


Figura 28: Porcentaje de tiempos del servidor en modo cliente-servidor con reconocedor basado en *Deep Learning*.

Por último, en la tabla 13 se mostrarán juntos los tiempos totales de cada uno.

	Local features	***Deep learning***
Local	0.325	/
Cliente - servidor	0.325	0.346

Tabla 13: Tiempos totales (en segundos) de cada reconocedor (media 100 iteraciones).

En vista de los resultados obtenidos, tanto de precisión como de tiempos, se podría decir que el reconocedor adecuado para esta tarea y que mejor funciona es el reconocedor basado en *local features*. En cuanto a la arquitectura a usar, ambas tienen sus ventajas y desventajas. El modo local no depende de la red, pero sin embargo requiere tener los objetos almacenados en el teléfono. Además, utiliza más memoria, puesto que carga en ella los objetos al iniciarse. Por otro lado, el modo cliente-servidor depende totalmente de la red, siendo esta uno de los principales cuellos de botella. Sin embargo presenta ventajas, como por ejemplo que si se quiere actualizar la base de datos, se puede hacer de manera transparente al usuario, sin que este tenga que actualizarla. Y en cuanto a términos de escalabilidad, esta es mejor en el modo cliente-servidor, puesto que aunque el coste para cada uno de los algoritmos de reconocimiento es linear en el número de imágenes, el servidor tarda mucho menos procesando las imágenes. Con lo cual, a más imágenes, mayor será la diferencia de tiempos entre ambas arquitecturas.

6. Conclusiones y trabajo futuro

En este último capítulo se harán unas conclusiones acerca del proyecto, se hablará de posibles líneas futuras para continuarlo, y se elaborará una opinión personal sobre el trabajo en sí, y sobre lo que le ha rodeado.

6.1. Conclusiones

Se ha logrado cumplir el objetivo del proyecto: implementar un reconocedor visual de objetos que se pueden encontrar en un supermercado para la plataforma Android, y que indique a los usuarios los posibles alérgenos que contengan dichos objetos. Se han evaluado dos algoritmos distintos de reconocimiento: uno basado en *local features*, que puede ser ejecutado tanto en local como en remoto, y otro basado en *Deep Learning*, que debido a su coste computacional solo puede ser ejecutado en remoto. Dentro de estos algoritmos, se han evaluado distintas versiones, buscando un equilibrio entre precisión y eficiencia.

Gracias a los resultados obtenidos se puede ver que, con la base de datos actual, el reconocedor que mejor funciona es el basado en *local features*. Esto es más notorio en la precisión, ya que funciona bastante mejor que el basado en *Deep Learning*. Se ha comprobado que esto se debe al formato de las imágenes de la base de datos y de la batería de test, ya que presentan resoluciones variables y las imágenes de test tienen mucho ruido de fondo. Probando con cinco imágenes sin ruido de fondo, la precisión ha sido total. En cuanto a la arquitectura, la conclusión depende de varios factores. Como ya se ha comentado en el capítulo anterior, la calidad y velocidad de la red es uno de ellos. Si se dispone de una red rápida, vale la pena implementar el modo cliente-servidor, puesto que las actualizaciones de la base de datos se realizarán solo del lado del servidor (el cliente no tendrá que actualizar nada), la aplicación en sí ocupará menos, y a más objetos haya en la base de datos, la diferencia con el modo local será más grande. Sin embargo, para una cantidad de objetos pequeña, el modo local funciona prácticamente igual que el modo cliente-servidor, con la ventaja de que funciona *offline* y que de que los tiempos son más constantes (el tiempo que se tarda en enviar y recibir una imagen al servidor depende de muchos factores, como número de clientes conectados, nivel de utilización de la red, etc).

En cuanto al modo *Deep Learning*, se ha comprobado que, efectivamente, es muy afectado por la oclusión y el ruido de fondo. Tiene también potencial en este proyecto, con por ejemplo optimizaciones que se nombrarán en la subsección 6.2, pero será necesario investigar sus resultados.

6.2. Trabajo futuro

Refiriéndose a líneas futuras sobre este proyecto, se pueden nombrar algunas que se han pensado durante el transcurso del proyecto, pero que estaban fuera del alcance del mismo.

Lo primero, en cuanto a lo ya existente, mejorar los algoritmos de reconocimiento. Para mejorar

el reconocedor basado en *local features*, se podría implementar algunas las mejoras nombradas en la sección 2.1, como por ejemplo, hacer más robusta la función de similitud entre imágenes mediante *Bag of words[14]* o filtrar las correspondencias mediante RANSAC [17]. En cuanto al algoritmo de *Deep Learning*, habría que adecuar la base de datos. Para ello, habría que tomar imágenes cuadradas de los objetos, y de un tamaño fijo. Otra posible mejora sería que apareciese un recuadro en la pantalla del móvil, se indicase al usuario que para reconocer un objeto lo sitúe en dicho recuadro, y sólo procesar la imagen de ese área. De esta manera se recortaría mucho ruido de fondo, lo cual ayuda al algoritmo de reconocimiento. También, en cuanto a la interfaz, se podría mejorar haciéndola más atractiva de cara al usuario (lo de ahora es un prototipo simplemente), y como se comenta en la sección 4.1 permitir al usuario seleccionar los alérgenos a los que es vulnerable, y que éstos se destacasen durante el reconocimiento en otro color.

En cuanto a las posibles aplicaciones del sistema o de parte del mismo en el mundo real, se cree que podría tener varias utilidades prácticas. Por ejemplo, si se implementa el sistema en un supermercado, con una base de datos de sus productos, los clientes podrían descargarse la aplicación y conectarse al servidor mediante la red del supermercado, permitiéndoles reconocer productos y leer sus alérgenos sin necesidad de consultar la etiqueta del mismo. Esto podría ser de ayuda a gente con problemas de vista que tengan dificultades para leer la letra de los ingredientes, o simplemente por comodidad. También podrían consultar la información del producto desde casa. Otra posibilidad, juntando el campo de la realidad aumentada, sería que una vez reconocido un producto, apareciese un cuadro junto a él con la información del producto. Además, si se implementa en un supermercado, se puede aprovechar la capacidad del algoritmo de *Deep Learning* evaluado en el proyecto de trabajar con bloques de imágenes sin apenas impacto en su rendimiento. Los clientes podrían ir enviando las imágenes no al servidor, sino a una cola, que una vez se llegase a cierto tamaño o cierto tiempo, las procesase todas de golpe. De esta manera se podría ahorrar tiempo con respecto al reconocedor basado en *local features*. También se podría utilizar la parte del reconocedor para crear una aplicación de asistencia para personas invidentes, que dijesen por voz el objeto reconocido, así como información acerca de él. Estas son solo algunas de las posibles ideas que se han ido barajando a lo largo del proyecto.

6.3. Opinión personal

La realización de este proyecto me ha generado una opinión personal muy positiva, tanto durante el transcurso del proyecto, como del resultado final. Este proyecto me ha permitido aplicar distintos conocimientos adquiridos a lo largo del grado, y adquirir experiencia en el campo de la investigación.

Desde un punto de vista tecnológico, me ha permitido aprender muchas tecnologías de las cuales hasta ahora sabía poco o nada. Una de mis motivaciones detrás del proyecto era poder aprender y experimentar con tecnologías que no había tocado hasta ahora. Por ejemplo, mis conocimientos en Android u OpenCV eran muy limitados, y este proyecto me ha permitido ampliarlos en gran medida. A su vez, he aprendido de cero tecnologías y herramientas como son Android NDK para

mezclar código nativo en C++ con Java, cosa que considero muy interesante, Caffe, Python, CMake, Google Protocol Buffers... En resumen, un conjunto de tecnologías amplio que, pese a complicar la curva de aprendizaje, me ha dejado muy satisfecho.

En lo personal, me ha permitido también tocar un área que está ahora muy en auge y en la cual tengo mucho interés: la visión por computador, y en concreto, los algoritmos basados en *Deep Learning*. Gracias al proyecto he podido comprender este área más en profundidad y descubrir las dificultades que entraña, lo cual considero que es muy importante y enriquecedor para mi futuro como ingeniero informático.

En mi opinión, el Trabajo de Fin de Grado es una parte clave en el desarrollo de un ingeniero informático, y mi caso no ha sido una excepción. Mi experiencia con la interacción con los directores de proyecto ha sido muy buena, así como con los conocimientos adquiridos durante el mismo, dando como resultado una base que espero que pueda servir para la implementación de alguna aplicación de reconocimiento de objetos de cara a la comunidad.

A. Manual de instalación del sistema

A.1. Requisitos previos aplicación:

- **Android Studio:** Descargar desde aquí `https://developer.android.com/studio/index.html?hl=es-419`.
- **Android SDK:** Incluido durante la instalación de Android Studio.
- **CrystaX's Android NDK:** Descargar desde aquí `https://www.crystax.net/en/android/ndk`
- **OpenCV 3.1 for Android:** Descargar desde aquí `http://opencv.org/downloads.html`
- **Google Protocol Buffers:** Descargar desde aquí `https://github.com/julienr/protobuf-android`

A.2. Requisitos previos servidor:

- **OpenCV 2.4.12:** Descargar desde aquí `http://opencv.org/downloads.html`
- **Google Protocol Buffers:** Descargar desde aquí `https://developers.google.com/protocol-buffers/docs/downloads`
- **Caffe:** Descargar desde aquí `http://caffe.berkeleyvision.org/installation.html`
- **Cuda:** Descargar desde aquí `https://developer.nvidia.com/cuda-75-downloads-archive`

A.3. Instalación

El código del sistema está subido a la plataforma GitHub, desde donde se puede clonar. El enlace al repositorio es el siguiente: `https://github.com/AMarquez94/UnizarNativeOpenCV`.

Se recomienda clonar el repositorio desde el IDE Android Studio, ya que permite automáticamente añadirlo como proyecto. Una vez clonado, habrá que editar algunos parámetros y rutas para importar correctamente todos los módulos y paquetes necesarios. La lista de cambios a realizar es la siguiente:

- En el fichero **local.properties** situado en la raíz del proyecto, hay que cambiar el parámetro **sdk.dir** por la ruta en la que se encuentre el SDK de Android.
- En el fichero **gradle.properties** situado también en la raíz del proyecto, hay que cambiar el parámetro **ndkDir** por la ruta en la que se encuentre el NDK descargado.
- En el fichero **Android.mk**, situado en la ruta *(raíz_proyecto)/app/src/main/jni*, cambiar las siguientes líneas:

- En la variable **OPENCVROOT** (línea 9) escribir la ruta en la que esté instalado OpenCV para Android.
- En la llamada a **import-add-path** (línea 43), sustituir la ruta existente por la ruta del directorio en el que se encuentre *Google Protocol Buffers* descargado.

Para compilar el código escrito en C++ y poder integrarlo con la aplicación, hay que ejecutar un comando especial. Se recomienda hacerlo de la siguiente manera. En Android Studio, ir al menú *File → Settings → Tools → External Tools*. allí, pulsar el símbulo "+" verde (Add). Aparecerá una pantalla como la que se muestra en la figura 29.

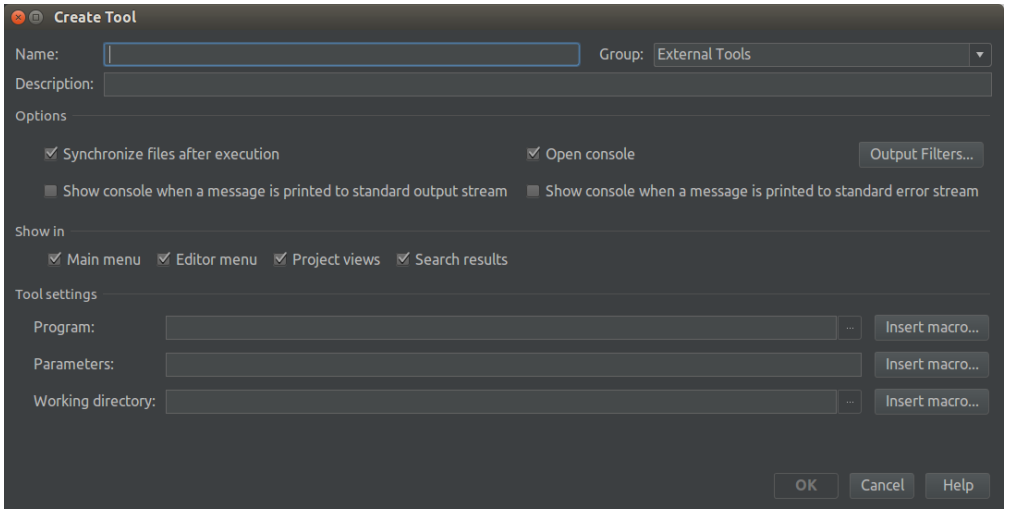


Figura 29: Pantalla de "External tools".

En esta pantalla, habrá que rellenar los datos como se indica:

- **Name:** ndk-build
- **Group:** Android Tools
- **Description:** (Opcional) Android Tool - NDK ndk-build tool
- **Options:** Marcar *Synchronize files after execution* y *Open console*. Desmarcar el resto.
- **Show in:** Marcar todas.
- **Program:** Insertar ruta al ejecutable ndk-build de ndk.
- **Parameters:** Insertar el siguiente parámetro (todo en la misma línea):

  ```
  NDK_PROJECT_PATH=$ModuleFileDir$/app/build/intermediates/ndk
  NDK_LIBS_OUT=$ModuleFileDir$/app/src/main/jniLibs
  NDK_APPLICATION_MK=$ModuleFileDir$/app/src/main/jni/Application.mk
  APP_BUILD_SCRIPT=$ModuleFileDir$/app/src/main/jni/Android.mk V=1
  ```

- **Working directory:** $SourcepathEntry$

El resultado una vez completado será algo similar a la imagen 30.

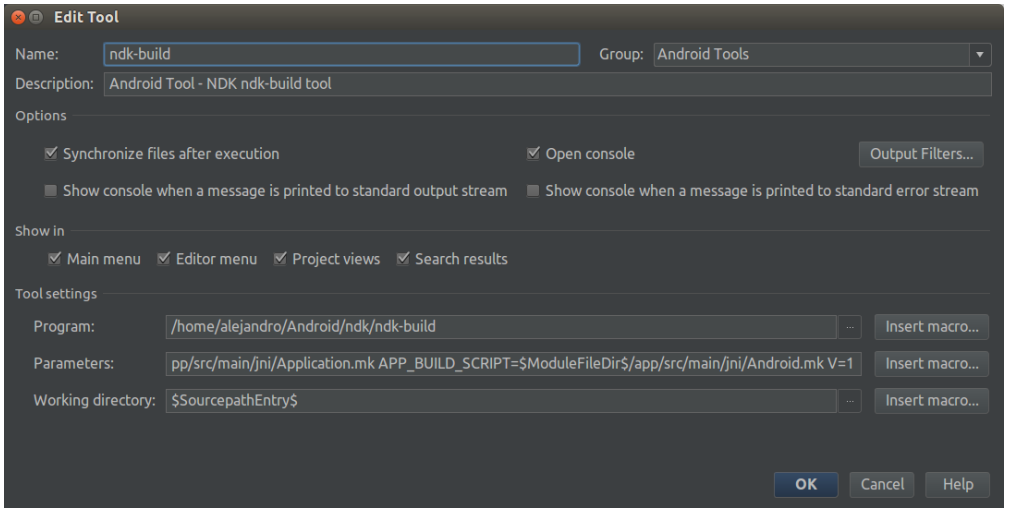


Figura 30: Pantalla de "External tools" una vez rellenada.

Después de guardar el resultado, al pulsar en Android Studio el botón derecho sobre algún archivo o directorio, aparecerá una nueva opción denominada "Android Tools", con un desplegable: "ndk-build". La primera vez que se importe este código, y cada vez que se modifique el código nativo, habrá que pulsar sobre el directorio raíz del proyecto y ejecutar este comando. De esa manera, se compilarán las librerías que conectan la parte Java con la parte nativa dentro de la *apk* generada, y ya se podrá instalar la aplicación en el móvil.

En cuanto a la instalación del servidor, su código también está en GitHub y el repositorio es el siguiente: `https://github.com/AMarquez94/UnizarNativeOpenCVServer`. Para compilarlo habrá que cambiar las líneas correspondientes en el fichero *CMakeLists.txt* a las rutas donde esté instalado *Caffe* y *Cuda*. Una vez compilado, generará el fichero binario **NativeOpenCV**. El comando para ejecutarlo es:

```
NativeOpenCV numPuerto (dl/kp)
```

Donde *numPuerto* será el número de puerto en el que el servidor se quedará escuchando, *kp* es que el servidor usará el reconocedor basado en *local features* y *dl* que el servidor usará el reconocedor basado en *Deep Learning*.

La base de datos usada en el proyecto se puede descargar utilizando este enlace: `https://drive.google.com/open?id=0B4Il89Zv6adpby1XTU1nc3BIMlk`

A.4. Requisitos aplicación

- *Smartphone* con cámara.

- Sistema operativo Android 14 o superior.
- Conexión a Internet (opcional).

B. Estructura de la base de datos

B.1. Objetos

Como ya se ha mencionado en el capítulo 3.2, la base de datos está compuesta por 50 objetos. Cada uno de esos objetos estará compuesto por un nombre, un indicador de si es fácil o no de reconocer, las imágenes que componen sus vistas, los nombres de dichas vistas, y la lista de alérgenos que componen el objeto. La base de datos se estructura de la siguiente forma:

Consta de un directorio por objeto. En cada directorio habrá una serie de imágenes, que se corresponderán a las vistas del objeto, y un fichero *info.txt*. Este fichero contendrá toda la información necesaria para crear una instancia de la clase objeto en el reconocedor. La sintaxis del fichero *info.txt* es la siguiente:

```
nombre_objeto
easy
num_vistas
(nombre_imagen_vista nombre_vista)+
(nombre_alergeno)*
```

Donde *nombre_objeto* será el nombre del objeto, *easy* será un booleano indicando si el objeto es de los fáciles de reconocer o no, *num_vistas* será el número de vistas que posee el objeto (mínimo una), *nombre_imagen_vista* será el nombre del fichero imagen conteniendo la vista con nombre *nombre_vista* (esta dupla se repetirá tantas veces como vistas haya) y *nombre_alergeno* será una lista con los alérgenos que contiene el objeto (de 0 a 14).

La lista de directorios de los objetos será tal y como se muestra en la figura 31

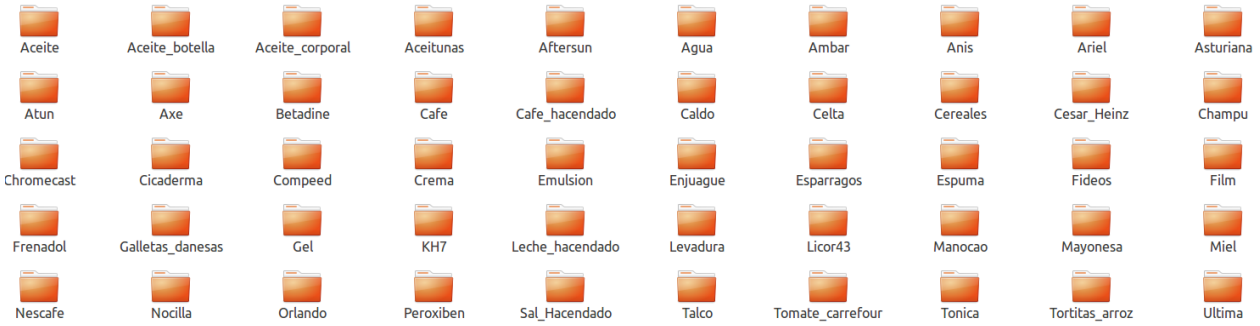


Figura 31: Objetos de la base de datos.

A su vez, en la figura 6 del capítulo 3.2 se muestran las imágenes que componen los objetos de la base de datos.

B.2. Objetos de test

Todas las imágenes que componen los objetos de test, de los cuales se ha hablado en el capítulo 5.1 estarán en un mismo directorio, junto con otro fichero *info.txt*. Este fichero, a diferencia del de los objetos de la base de datos, contendrá una línea por imagen, con la tupla *nombre_imagen nombre_objeto*. Por lo tanto la estructura de este fichero será:

(nombre_imagen nombre_objeto)+

A continuación, en la figura 32 se mostrarán todas las imágenes que componen los objetos de test en miniatura.

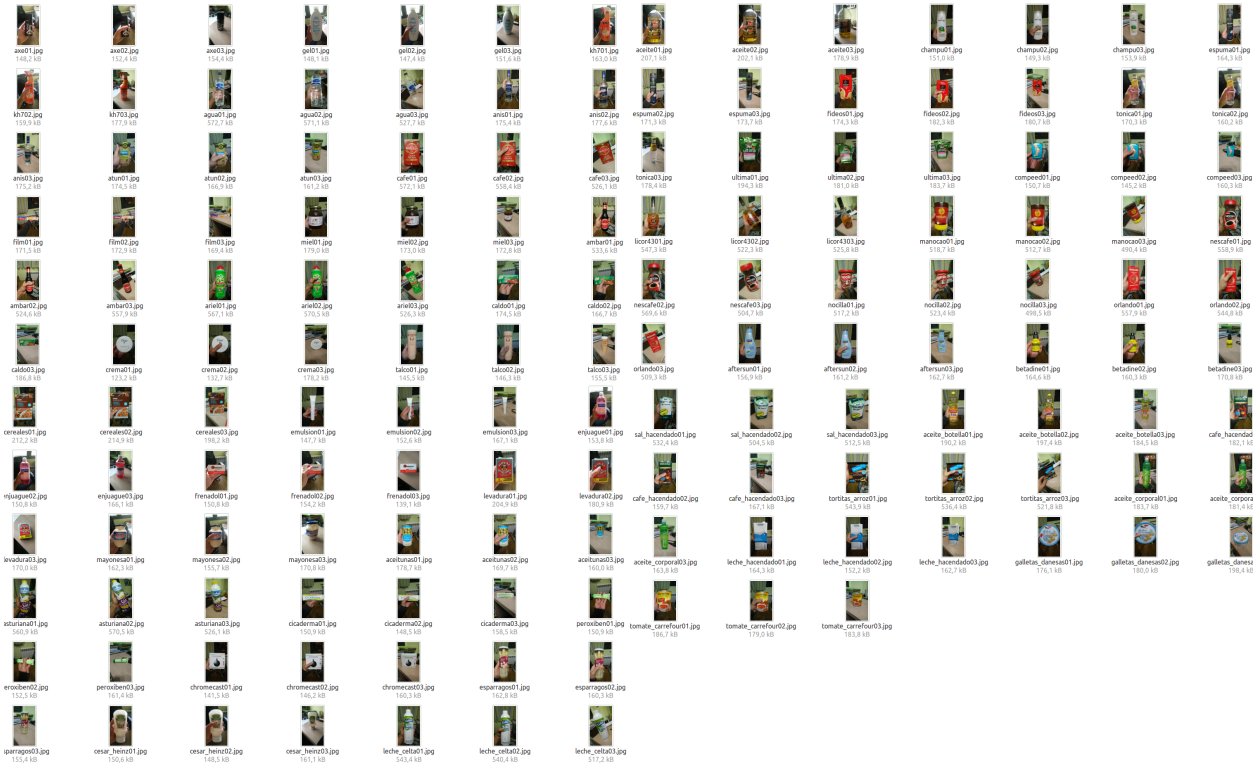


Figura 32: Objetos de test.

C. Diagrama de clases del sistema

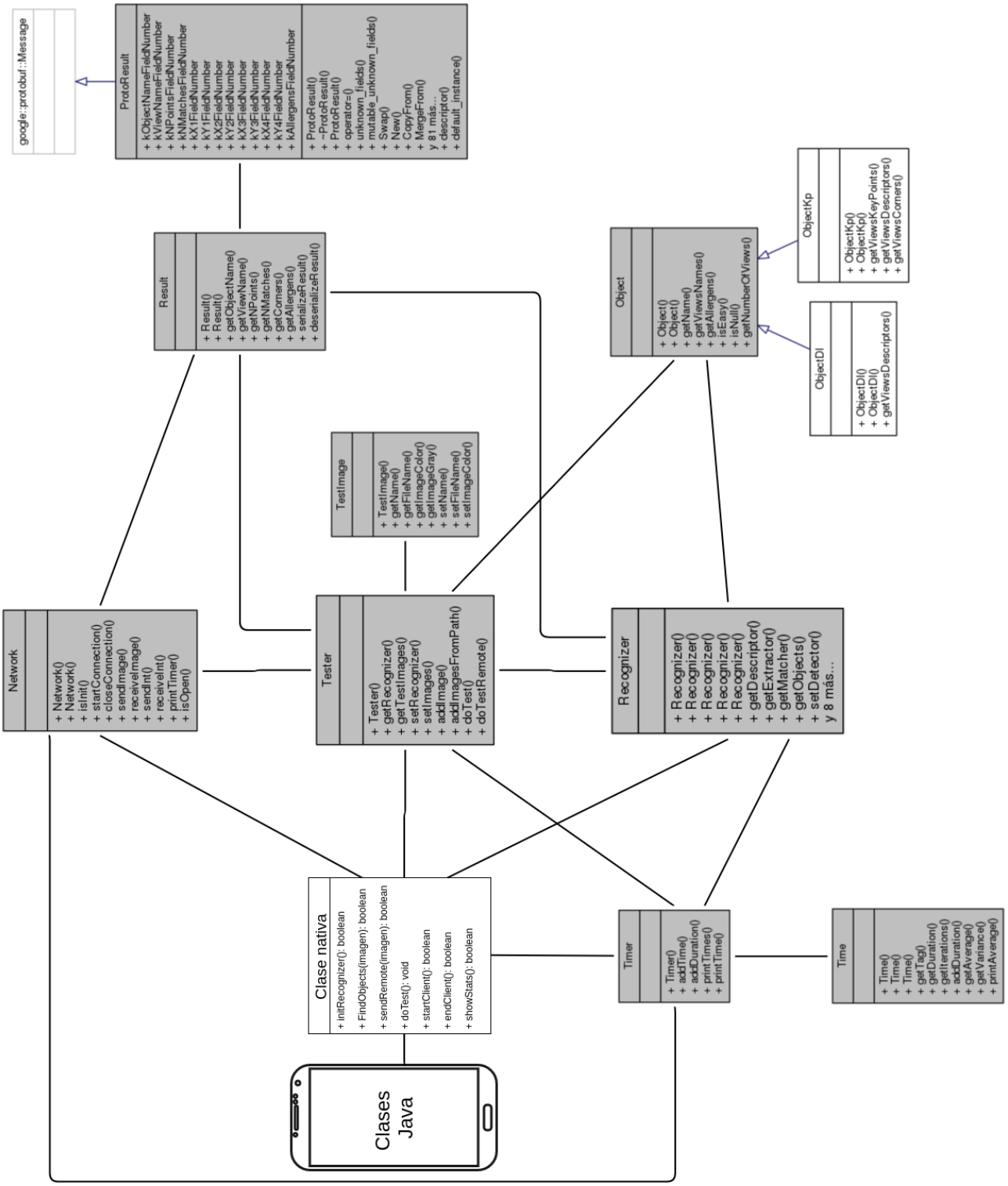


Figura 33: Diagrama de la relación entre clases de la aplicación.

D. Algoritmo ORB (Oriented FAST and Rotated BRIEF)

En este anexo se explica un poco más en detalle el detector/descriptor ORB. Todo lo comentado aquí estará más detalladamente contado en el artículo original [11].

ORB nace como alternativa eficiente a los detectores de características SIFT [8] y SURF [9]. El objetivo de sus autores era proporcionar un detector/descriptor de *local features* que pudiese ser empleado en tareas de tiempo real, y ejecutado en dispositivos de menor potencia que un ordenador, como por ejemplo *smartphones*. El resultado que afirman que obtuvieron es un reemplazo de *SIFT* que tiene un desempeño similar, pero está menos afectado por el ruido de la imagen y puede ser utilizado en aplicaciones en tiempo real. Según sus resultados obtenidos, ORB es hasta dos órdenes de magnitud más rápido que SIFT, y uno más que SURF.

ORB está basado en el detector FAST [10] y el descriptor BRIEF [23]. Estos métodos fueron modificados para resolver ciertas limitaciones que tenían.

FAST es un método eficiente en tiempo para encontrar puntos de interés, utilizado en sistemas de tiempo real. Sin embargo, a diferencia de otros detectores de *features*, como los ya nombrados, no incluye la descripción de la rotación de un punto de interés. Por lo tanto, una de las modificaciones realizadas sobre FAST fue proporcionar esa descripción para cada punto de interés. La técnica utilizada para ello es la denominada "Orientación por intensidad de centroide" que, en base a la intensidad de las esquinas (obtenida mediante la medida de Harris [24]) que rodean un punto, determina su orientación.

BRIEF es un descriptor binario cuyos resultados en cuanto a robustez frente a cambios de luz, difuminación y distorsión de la imagen es similar a SIFT. Sin embargo, es muy sensible a las rotaciones. Para solucionarlo de manera eficiente, primero orientaron el descriptor BRIEF de acuerdo a la orientación de los puntos de interés, y después aumentaron la varianza y disminuyeron la correlación del descriptor, que son dos factores que hacen a una *feature* más discriminativa, y por tanto, mejor a la hora de describirla.

A estas dos modificaciones se las llamó oFAST (Oriented FAST) y rBRIEF (Rotated BRIEF) respectivamente. De ahí el origen del nombre del detector/descriptor, Oriented FAST and Rotated BRIEF (ORB).

En el artículo original se detallan los experimentos de validación realizados, entre ellos, las diferencias de tiempos de ejecución entre algoritmos, que, como se ve en la tabla 14 son muy notorios, sobre todo la diferencia de ORB con SIFT.

	ORB	**SURF**	**SIFT**
Tiempo (ms)	15.3	217.3	5228.7

Tabla 14: Comparativa de tiempos de procesar una imagen usaando ORB, SURF y SIFT

E. Procesamiento en Redes neuronales convolucionales (CNN)

Se pueden distinguir dos fases en el procesado de una imagen en una red CNN: fase de procesado de datos, y fase de clasificación final. El tipo de procesado de los datos a través de una redes CNN se realiza principalmente mediante dos tipos de neuronas:

- **Neuronas convolucionales:** Cada una de estas neuronas realiza una operación matricial sobre una región de la imagen. Alguna de estas operaciones tiene sentido visual (como por ejemplo, remarcar los bordes), pero otras no son triviales y carecen de él. El objetivo de estas operaciones es filtrar la imagen, acentuando las características correspondientes siguiendo el modelo con el que ha sido entrenada la red.
- **Neuronas de reducción de muestreo:** Estas neuronas tienen la función de reducir la región que les corresponde, con el fin de reducir el coste computacional y el sobreajuste. La forma más común de efectuar esta reducción es la que se ve en la figura 34, la cual reduce una ventana formada por cuatro sub-ventanas 2x2 a una sola ventana 2x2, descartando todos los valores excepto los mayores de cada sub-ventana. [17]

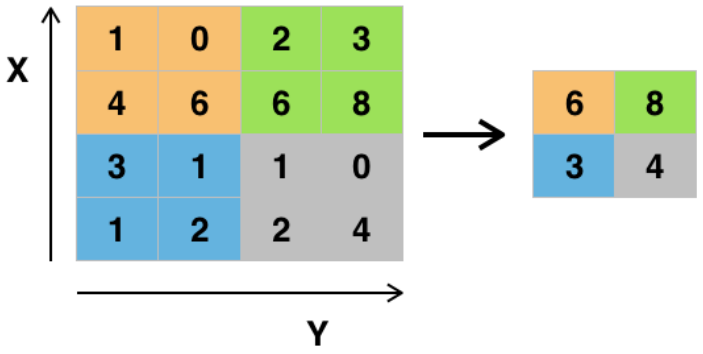


Figura 34: A la izquierda: antes de aplicar la función de muestreo. A la derecha, resultado de aplicar la función.

La siguiente fase, la fase de clasificación final, se compone de un único tipo de neurona:

- **Neuronas de clasificación:** Estas neuronas reciben como entrada el resultado de las operaciones de las neuronas de la primera fase sobre la imagen, obteniendo un conjunto de valores numéricos que representan características de la imagen. La función de estas neuronas, dependiendo de esas características, clasificar la imagen.

El proceso completo se puede observar en la figura 35. Al principio trabajan las neuronas convolucionales aplicando operaciones matriciales sobre regiones de la imagen, alternadas con

[17] Fuente imagen: `https://en.wikipedia.org/wiki/Convolutional_neural_network`

neuronas que reducen el muestreo. Finalmente, las neuronas de clasificación producen la salida final. En resumen, se podría decir que las neuronas de la primera fase se encargan de extraer las características de una imagen, y las de la segunda fase de resumir esas características en un descriptor, obteniendo finalmente el resultado de la clasificación. [17]

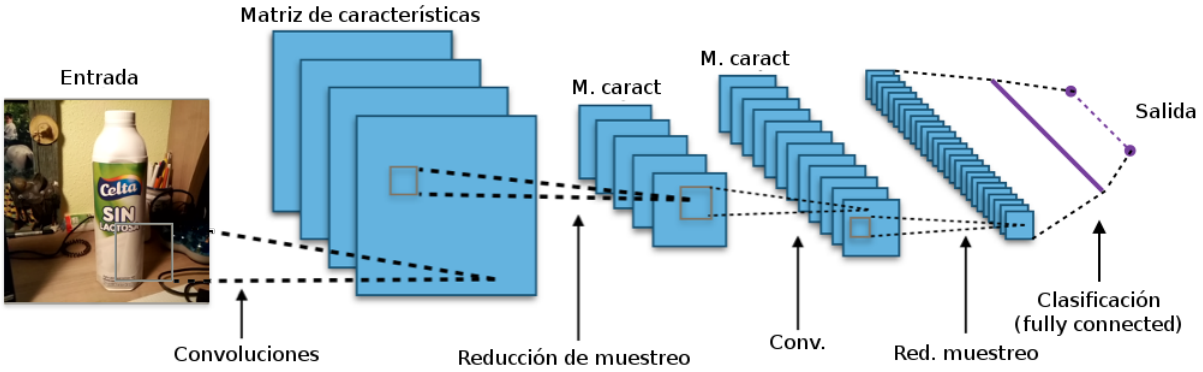


Figura 35: Diagrama explicativo de las distintas capas que forman la red neuronal convolucional y sus operaciones y resultados.

F. Resultados adicionales de los experimentos

En este anexo se mostrarán todas las pruebas y experimentos que se han realizado a lo largo del proyecto.

F.1. Experimentos algoritmos de reconocimiento basados en *local features*. Primera versión

Estos experimentos se realizaron sobre la primera versión de la base de datos, formada por 13 objetos. Solo se probaron reconocedores basados en *local features*.

- **Android. ORB/ORB.**

  Tiempo: 7.98 segs. Media puntos: 500. Media correspondencias buenas: 33

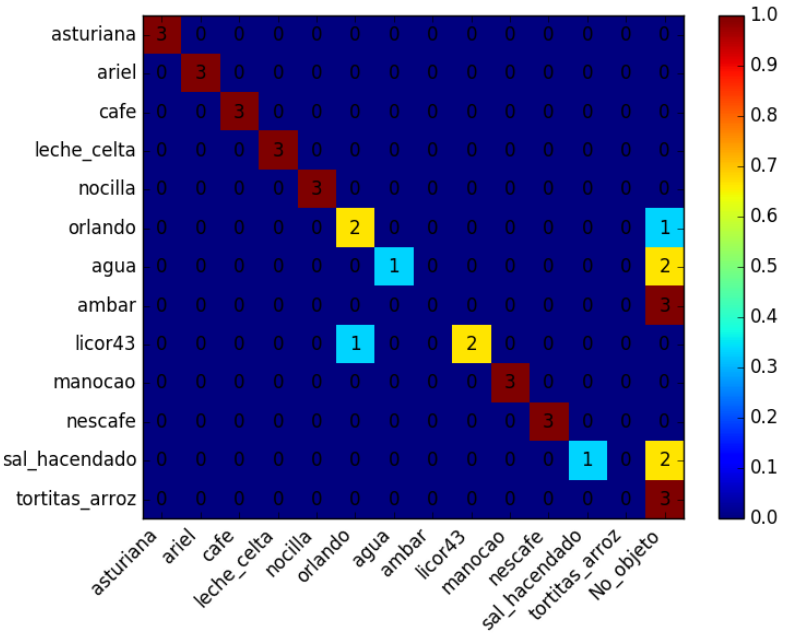


Figura 36: Matriz de confusión de los resultados obtenidos mediante ORB en Android.

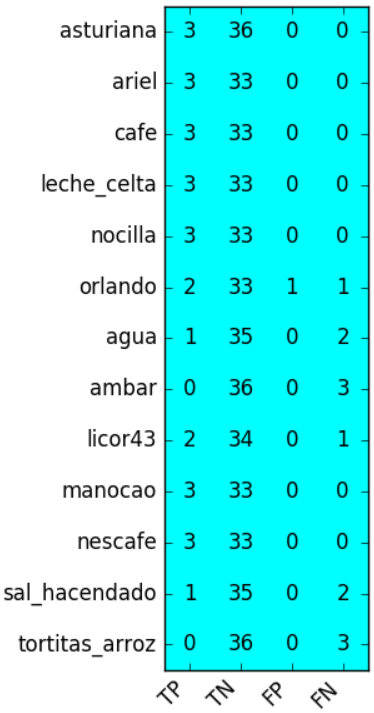


Figura 37: Tabla de “True positives” de los resultados obtenidos mediante ORB en Android

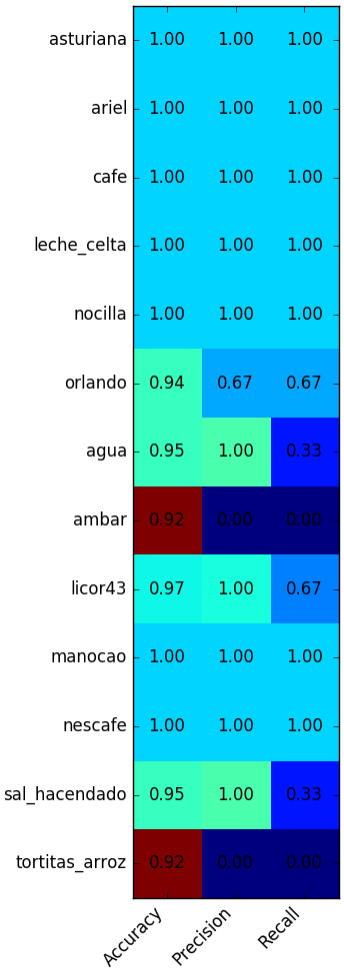


Figura 38: Tabla de precisión de los resultados obtenidos mediante ORB en Android.

- **PC. ORB/ORB.**

  Tiempo: 1.57 segs. Media puntos: 500. Media correspondencias buenas: 33

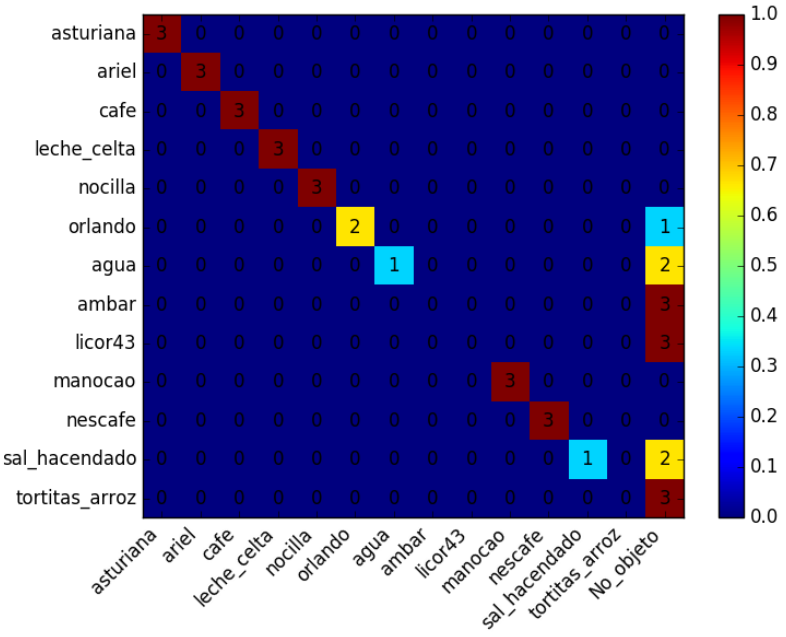


Figura 39: Matriz de confusión de los resultados obtenidos mediante ORB en PC.

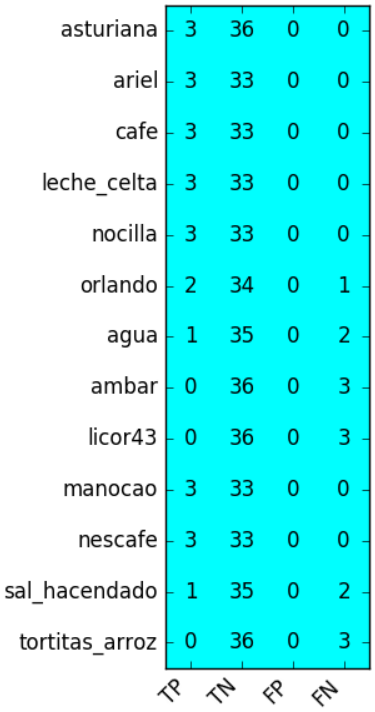


Figura 40: Tabla de "True positives" de los resultados obtenidos mediante ORB en PC

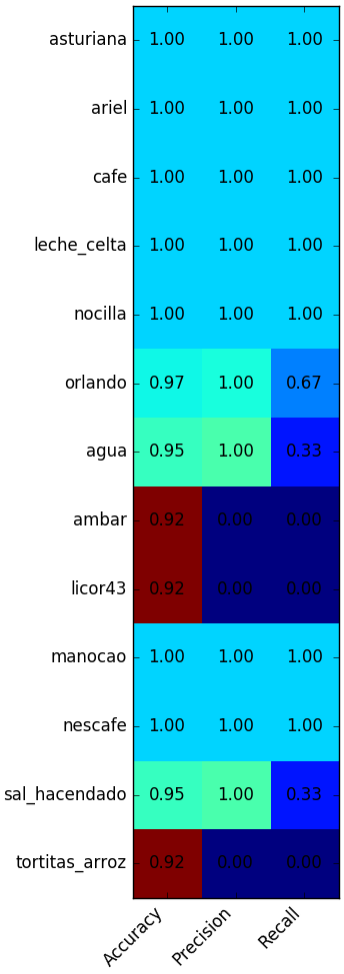


Figura 41: Tabla de precisión de los resultados obtenidos mediante ORB en PC.

- **Android. ORB(WTA_K = 4)/ORB(WTA_K = 4).**

  Tiempo: 9.6 segs. Media puntos: 500. Media correspondencias buenas: 37

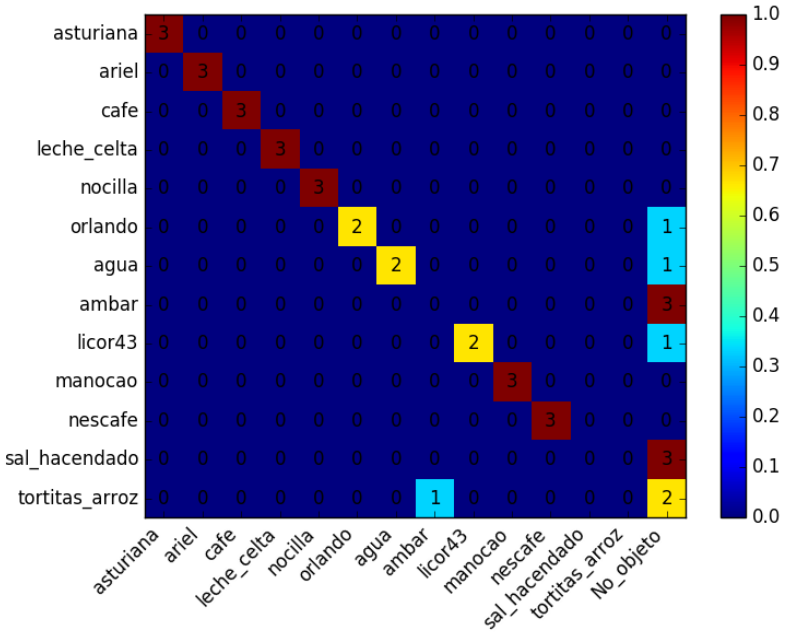


Figura 42: Matriz de confusión de los resultados obtenidos mediante ORB (WTA_K = 4) en Android.

	TP	TN	FP	FN
asturiana	3	36	0	0
ariel	3	33	0	0
cafe	3	33	0	0
leche_celta	3	33	0	0
nocilla	3	33	0	0
orlando	2	34	0	1
agua	2	34	0	1
ambar	0	35	1	3
licor43	2	34	0	1
manocao	3	33	0	0
nescafe	3	33	0	0
sal_hacendado	0	36	0	3
tortitas_arroz	0	36	0	3

Figura 43: Tabla de "True positives" de los resultados obtenidos mediante ORB (WTA_K = 4) en Android

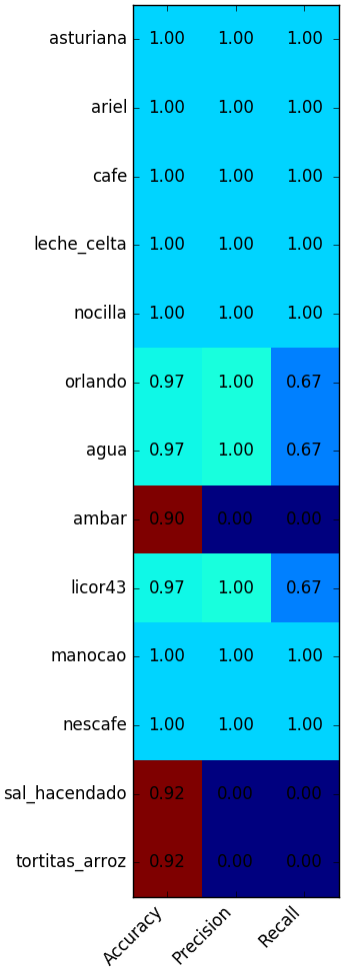


Figura 44: Tabla de precisión de los resultados obtenidos mediante ORB (WTA_K = 4) en Android.

- **PC. ORB(WTA_K = 4)/ORB(WTA_K = 4).**

  Tiempo: 1.55 segs. Media puntos: 500. Media correspondencias buenas: 36

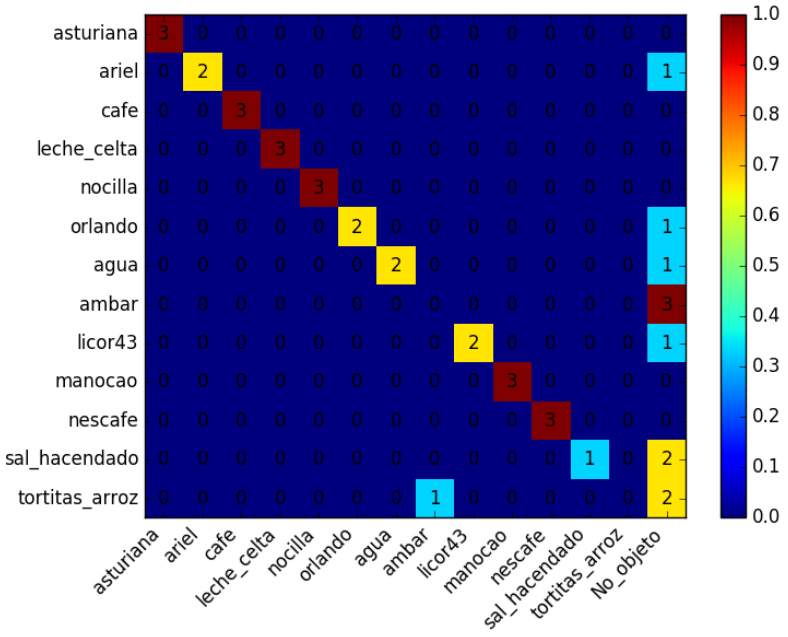


Figura 45: Matriz de confusión de los resultados obtenidos mediante ORB(WTA_K = 4) en PC.

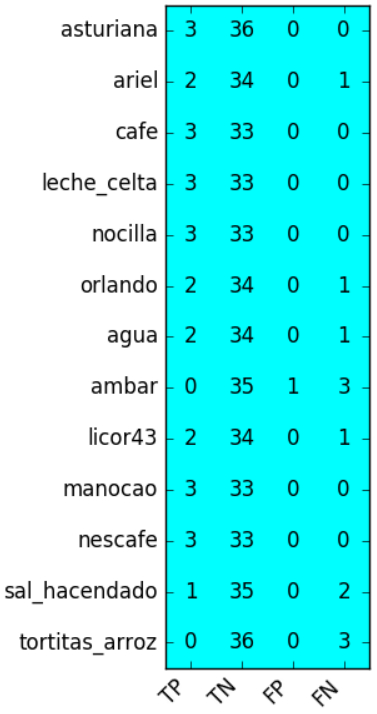


Figura 46: Tabla de "True positives" de los resultados obtenidos mediante ORB(WTA_K = 4) en PC

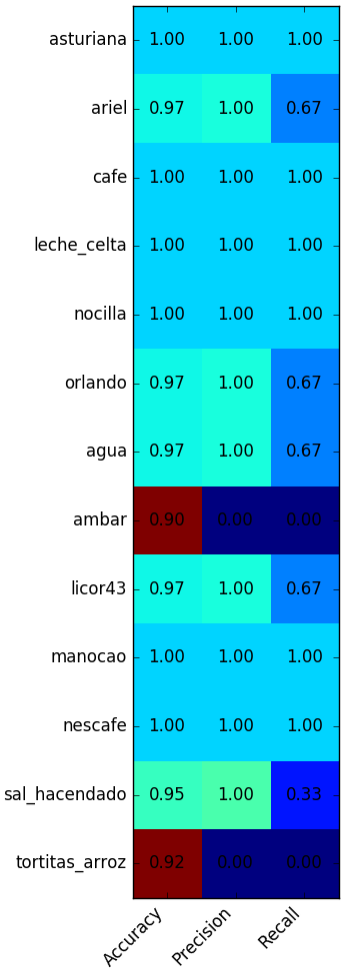


Figura 47: Tabla de precisión de los resultados obtenidos mediante ORB(WTA_K = 4) en PC.

- **Android. FAST/BRISK.**

  Tiempo: 383.53 segs. Media puntos: 5107. Media correspondencias buenas: 83

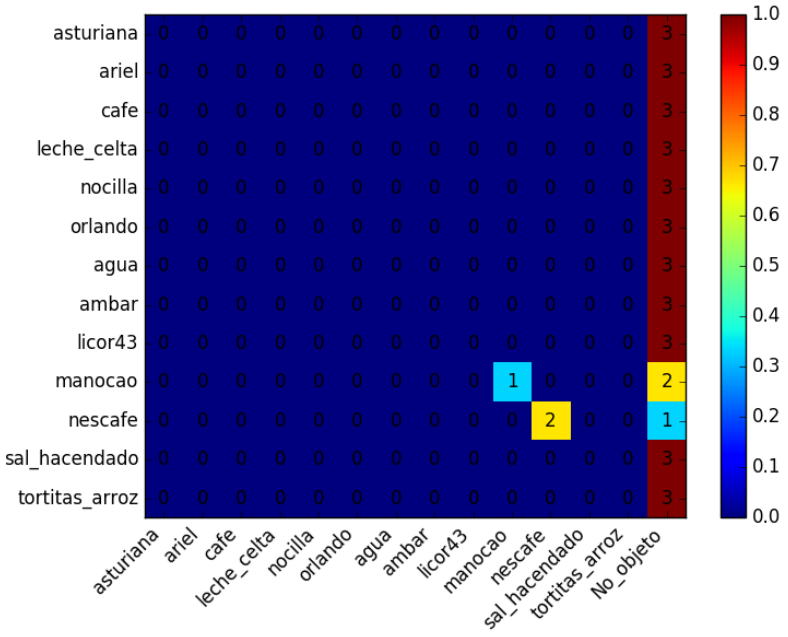


Figura 48: Matriz de confusión de los resultados obtenidos mediante FAST/BRISK en Android.

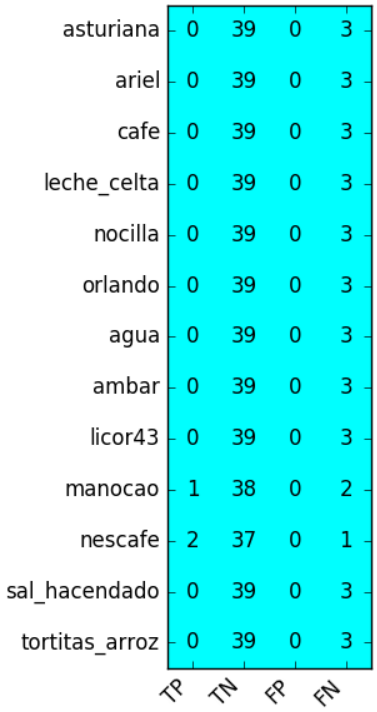


Figura 49: Tabla de "True positives" de los resultados obtenidos mediante FAST/BRISK en Android

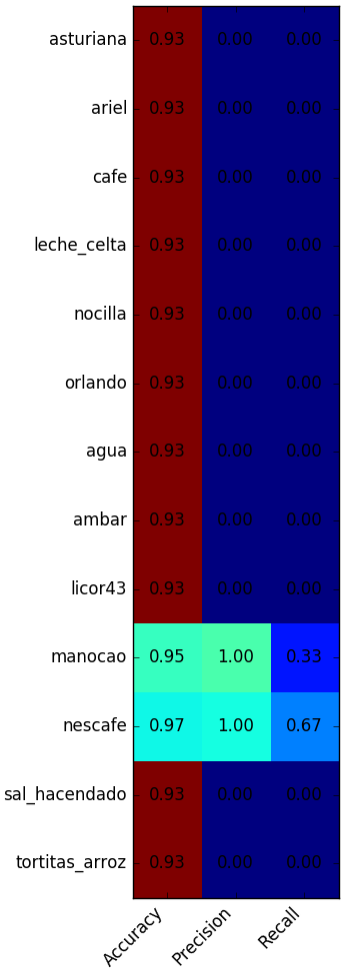


Figura 50: Tabla de precisión de los resultados obtenidos mediante FAST/BRISK en Android.

- **PC. FAST/BRISK.**

  Tiempo: 229.16 segs. Media puntos: 5107. Media correspondencias buenas: 82

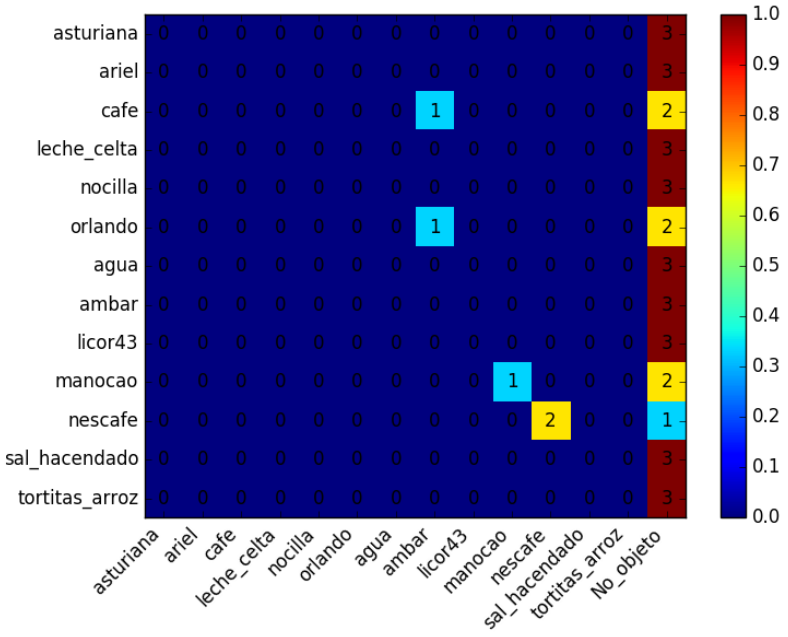


Figura 51: Matriz de confusión de los resultados obtenidos mediante FAST/BRISK en PC.

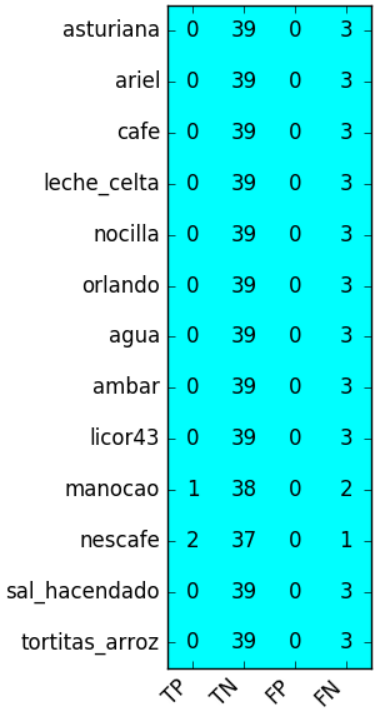


Figura 52: Tabla de "True positives" de los resultados obtenidos mediante FAST/BRISK en PC

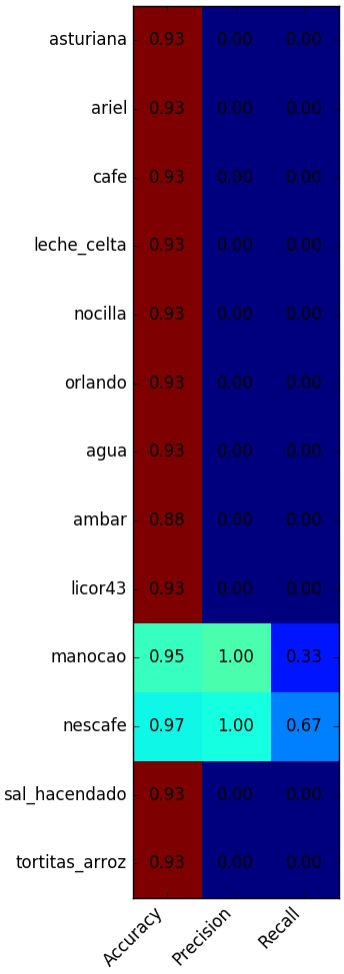


Figura 53: Tabla de precisión de los resultados obtenidos mediante FAST/BRISK en PC.

- **Android. BRISK/BRISK.**

  Tiempo: 57.53 segs. Media puntos: 2191. Media correspondencias buenas: 84

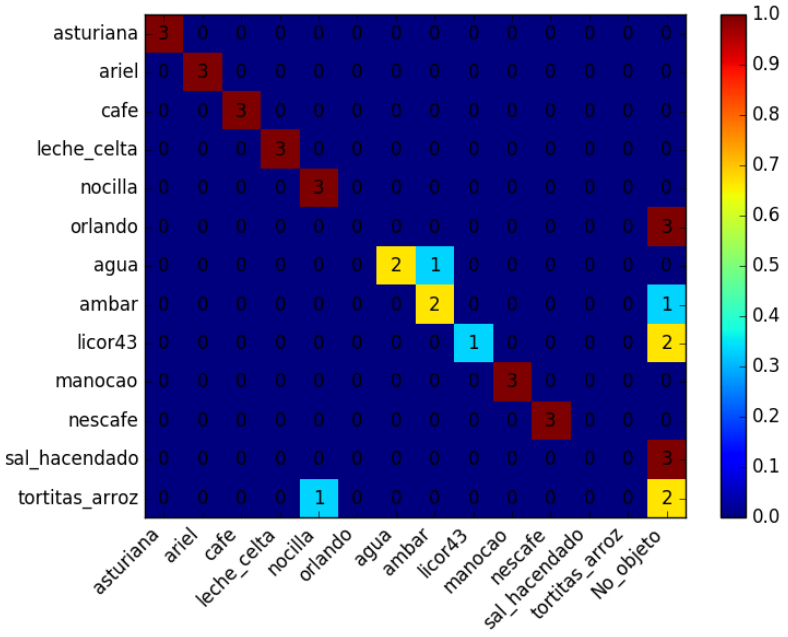


Figura 54: Matriz de confusión de los resultados obtenidos mediante BRISK en Android.

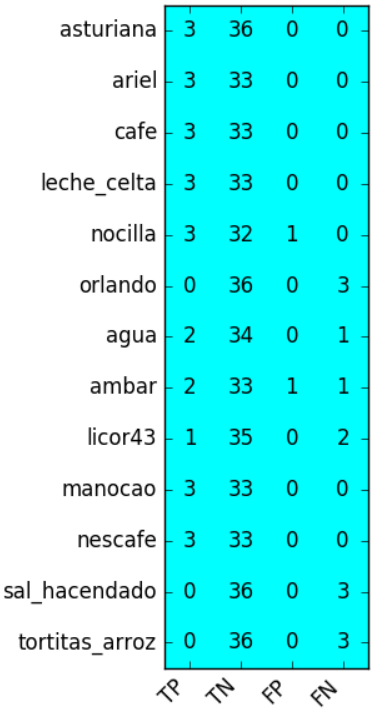


Figura 55: Tabla de “True positives” de los resultados obtenidos mediante BRISK en Android

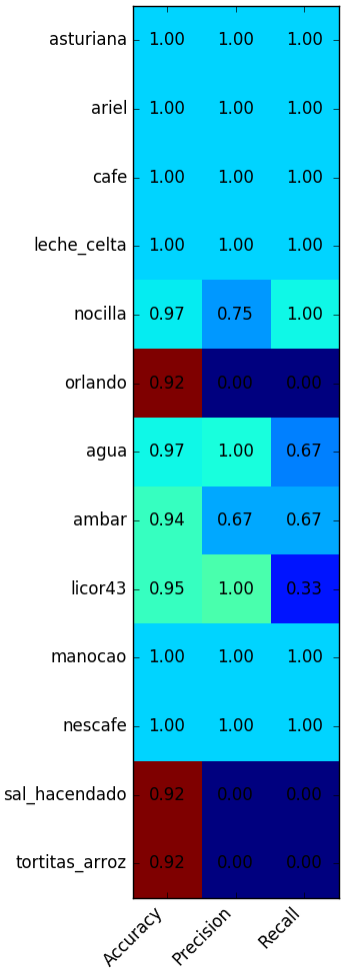


Figura 56: Tabla de precisión de los resultados obtenidos mediante BRISK en Android.

- **PC. BRISK/BRISK.**

  Tiempo: 27.54 segs. Media puntos: 2161. Media correspondencias buenas: 84

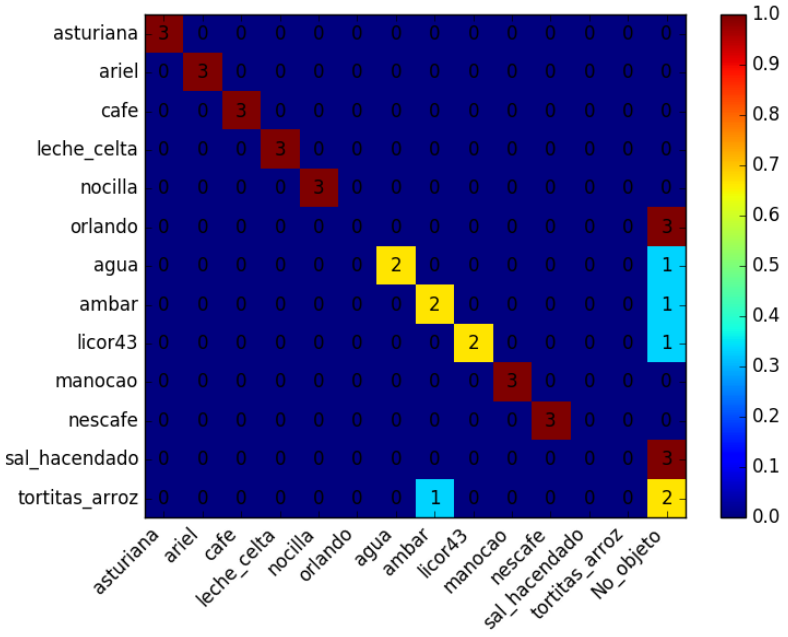


Figura 57: Matriz de confusión de los resultados obtenidos mediante BRISK en PC.

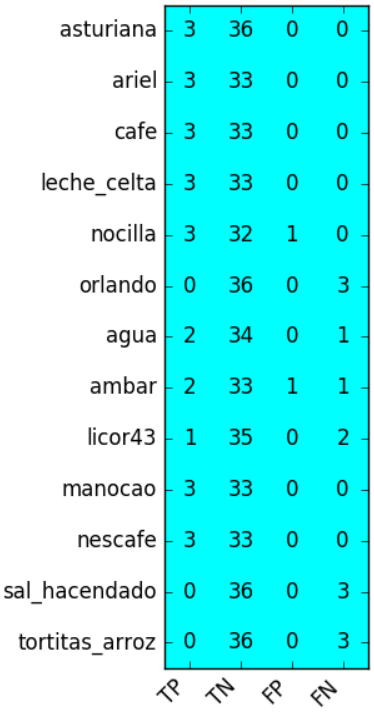


Figura 58: Tabla de “True positives” de los resultados obtenidos mediante BRISK en PC

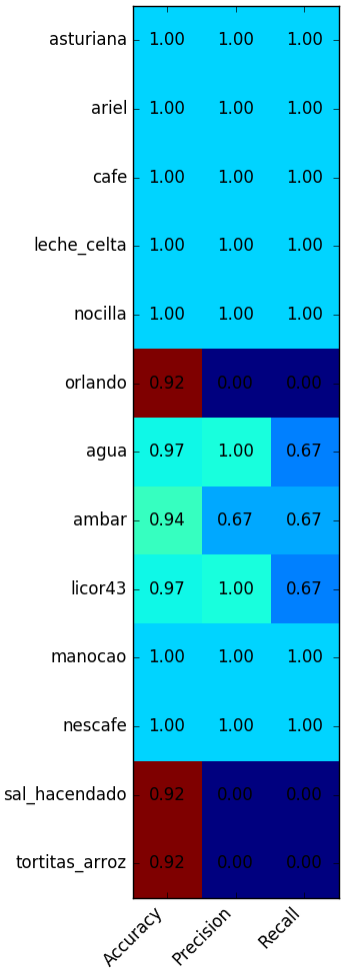


Figura 59: Tabla de precisión de los resultados obtenidos mediante BRISK en PC.

- **Android. BRISK(thres = 85)/BRISK(thres = 85).**

  Tiempo:9.59 segs. Media puntos: 467. Media correspondencias buenas: 38

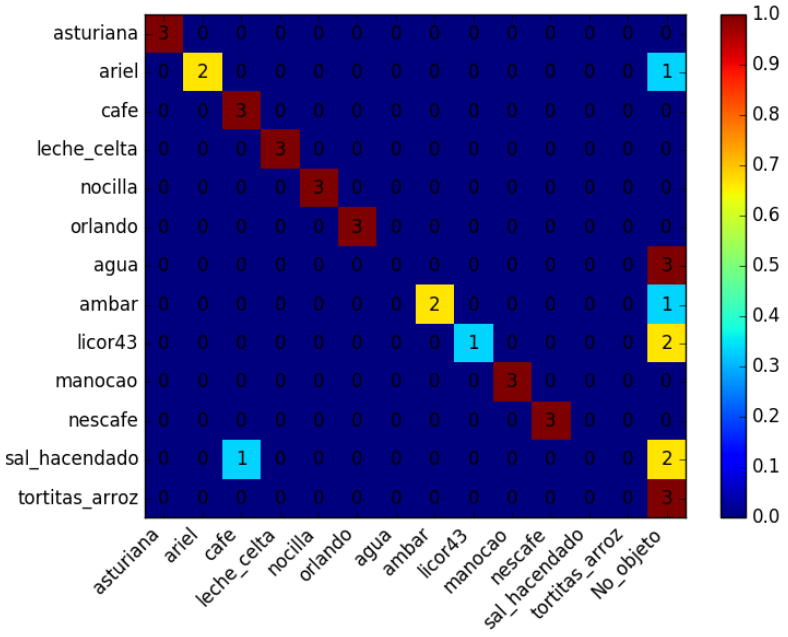


Figura 60: Matriz de confusión de los resultados obtenidos mediante BRISK(thres = 85) en Android.

	TP	TN	FP	FN
asturiana	3	36	0	0
ariel	2	34	0	1
cafe	3	32	1	0
leche_celta	3	33	0	0
nocilla	3	33	0	0
orlando	3	33	0	0
agua	0	36	0	3
ambar	2	34	0	1
licor43	1	35	0	2
manocao	3	33	0	0
nescafe	3	33	0	0
sal_hacendado	0	36	0	3
tortitas_arroz	0	36	0	3

Figura 61: Tabla de “True positives” de los resultados obtenidos mediante BRISK(thres = 85) en Android

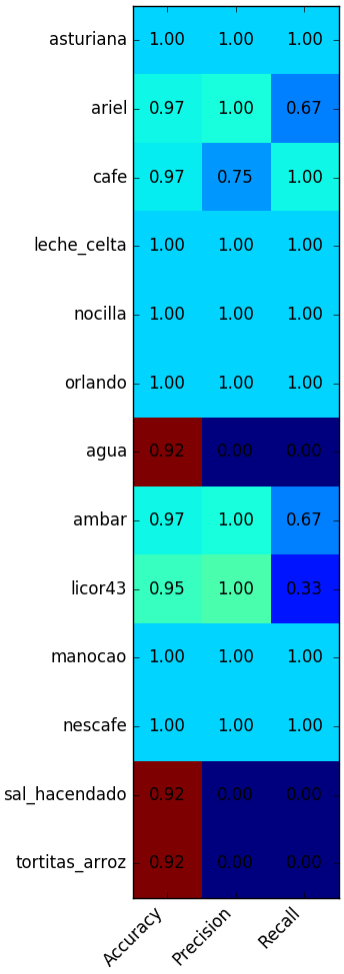


Figura 62: Tabla de precisión de los resultados obtenidos mediante BRISK(thres = 85) en Android.

- **PC. BRISK(thres = 85)/BRISK(thres = 85).**

  Tiempo: 2.34 segs. Media puntos: 465. Media correspondencias buenas: 37

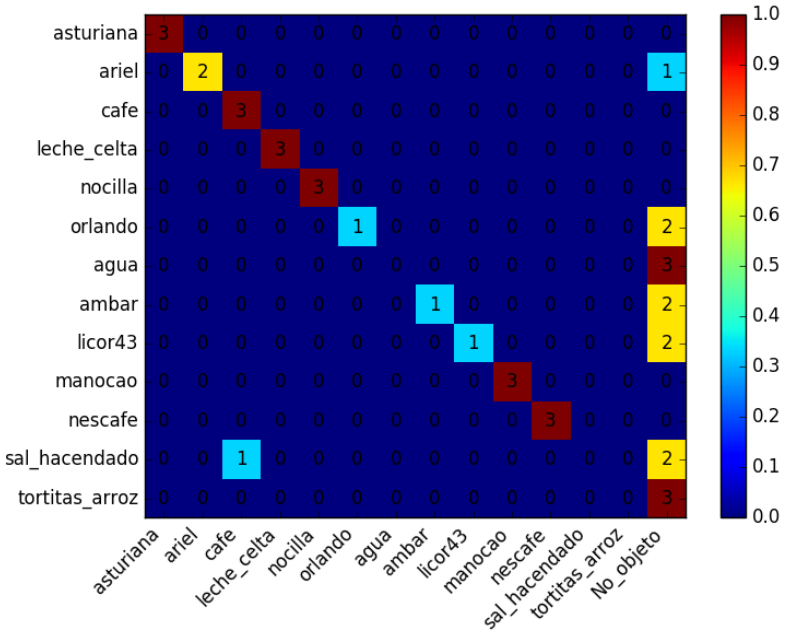


Figura 63: Matriz de confusión de los resultados obtenidos mediante BRISK(thres = 85) en PC.

	TP	TN	FP	FN
asturiana	3	36	0	0
ariel	2	34	0	1
cafe	3	32	1	0
leche_celta	3	33	0	0
nocilla	3	33	0	0
orlando	3	33	0	0
agua	0	36	0	3
ambar	2	34	0	1
licor43	1	35	0	2
manocao	3	33	0	0
nescafe	3	33	0	0
sal_hacendado	0	36	0	3
tortitas_arroz	0	36	0	3

Figura 64: Tabla de “True positives” de los resultados obtenidos mediante BRISK(thres = 85) en PC

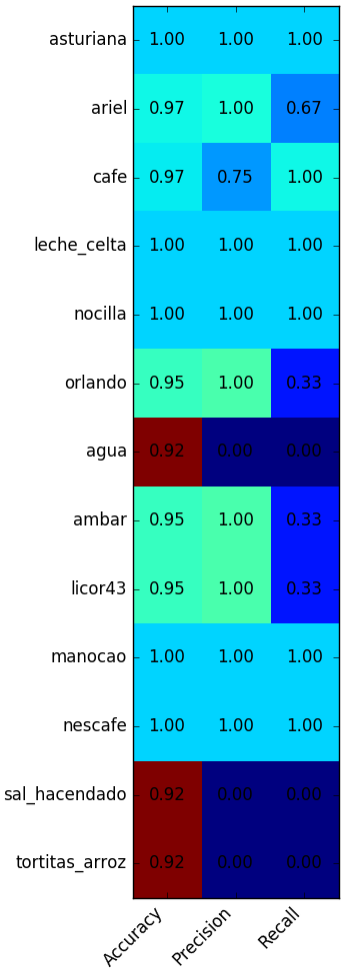


Figura 65: Tabla de precisión de los resultados obtenidos mediante BRISK(thres = 85) en PC.

- **Android. FAST/ORB.**

  Tiempo: 338.12 segs. Media puntos: 4731. Media correspondencias buenas: 490

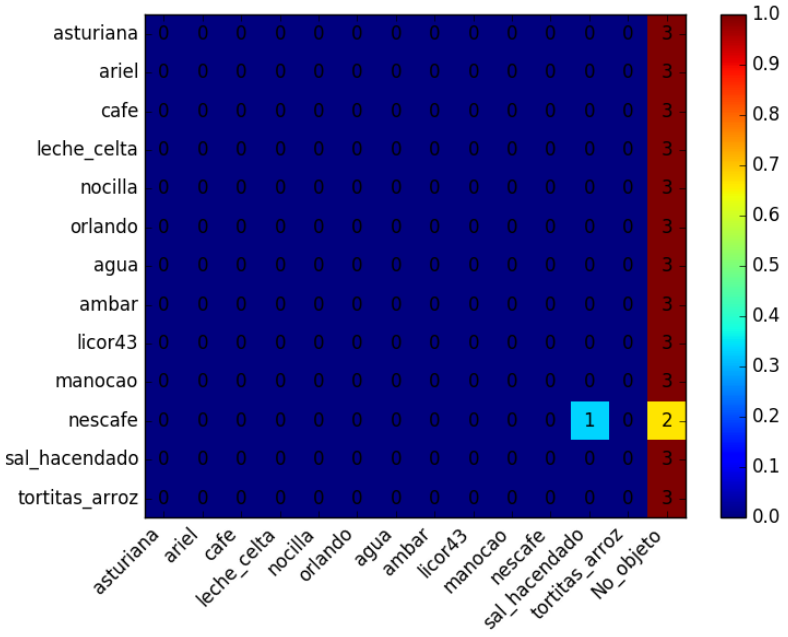


Figura 66: Matriz de confusión de los resultados obtenidos mediante FAST/ORB en Android.

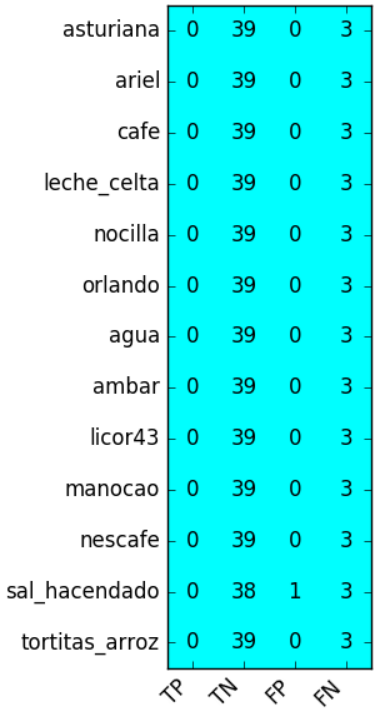


Figura 67: Tabla de "True positives" de los resultados obtenidos mediante FAST/ORB en Android

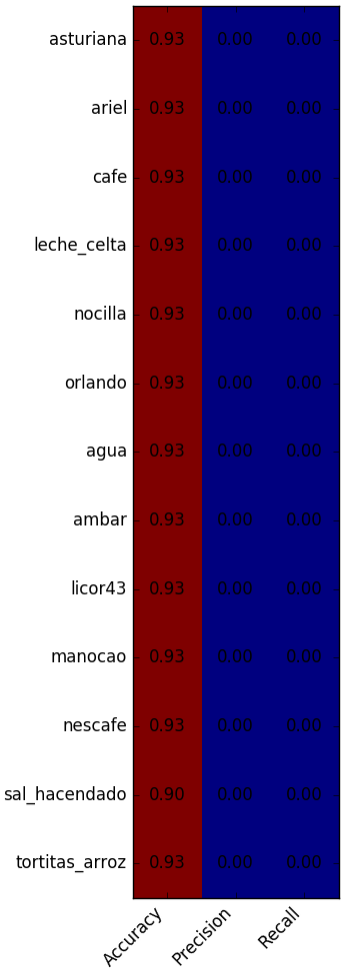


Figura 68: Tabla de precisión de los resultados obtenidos mediante FAST/ORB en Android.

- **PC. FAST/ORB.**

  Tiempo: 165.73 segs. Media puntos: 4728. Media correspondencias buenas: 463

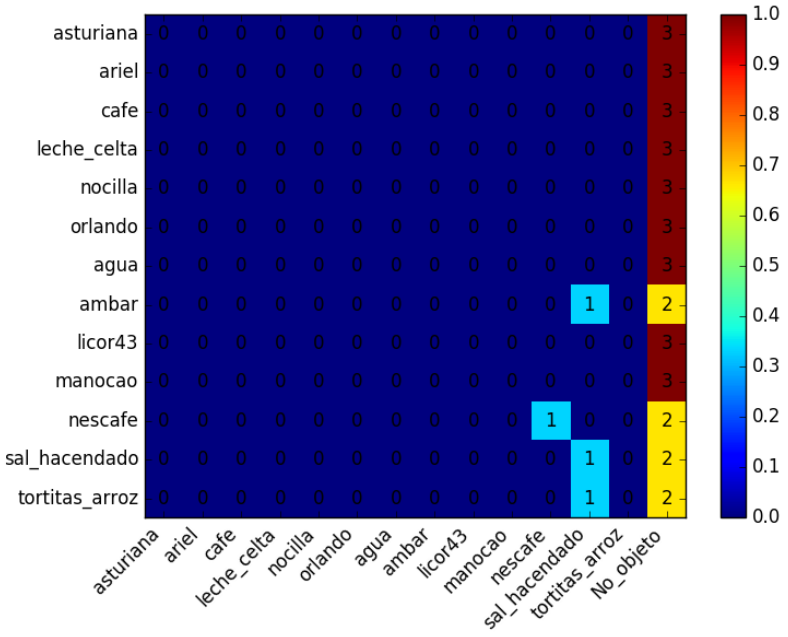


Figura 69: Matriz de confusión de los resultados obtenidos mediante FAST/ORB en PC.

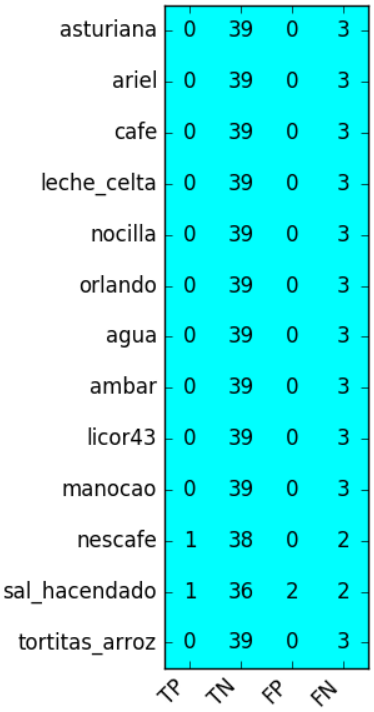


Figura 70: Tabla de “True positives” de los resultados obtenidos mediante FAST/ORB en PC

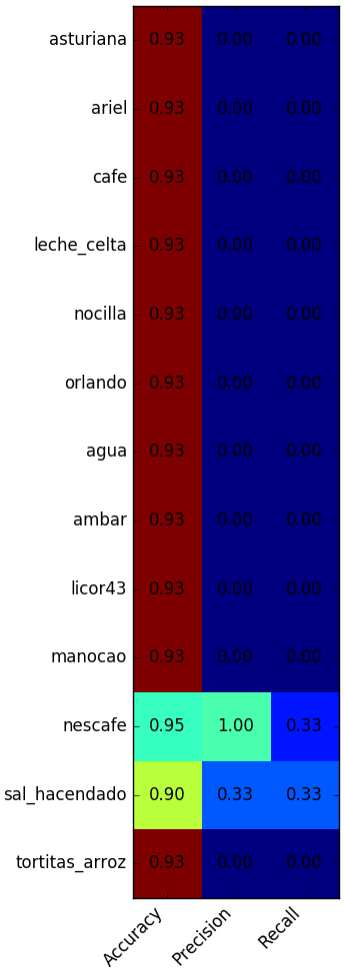


Figura 71: Tabla de precisión de los resultados obtenidos mediante FAST/ORB en PC.

F.2. Experimentos algoritmos de reconocimiento basados en *local features*. Segunda versión

Estas pruebas se realizaron ya con la base de datos final, compuesta por 50 objetos (54 imágenes en total). Aquí se probaron las arquitecturas de local y cliente-servidor (siendo el servidor el portátil) para todos los reconocedores candidatos, y la arquitectura cliente-servidor (siendo el servidor la máquina remota) para el reconocedor elegido finalmente (ORB).

- **Android. ORB/ORB.**

  Tiempo: 48.23 segs. Media puntos: 500. Media correspondencias buenas: 27

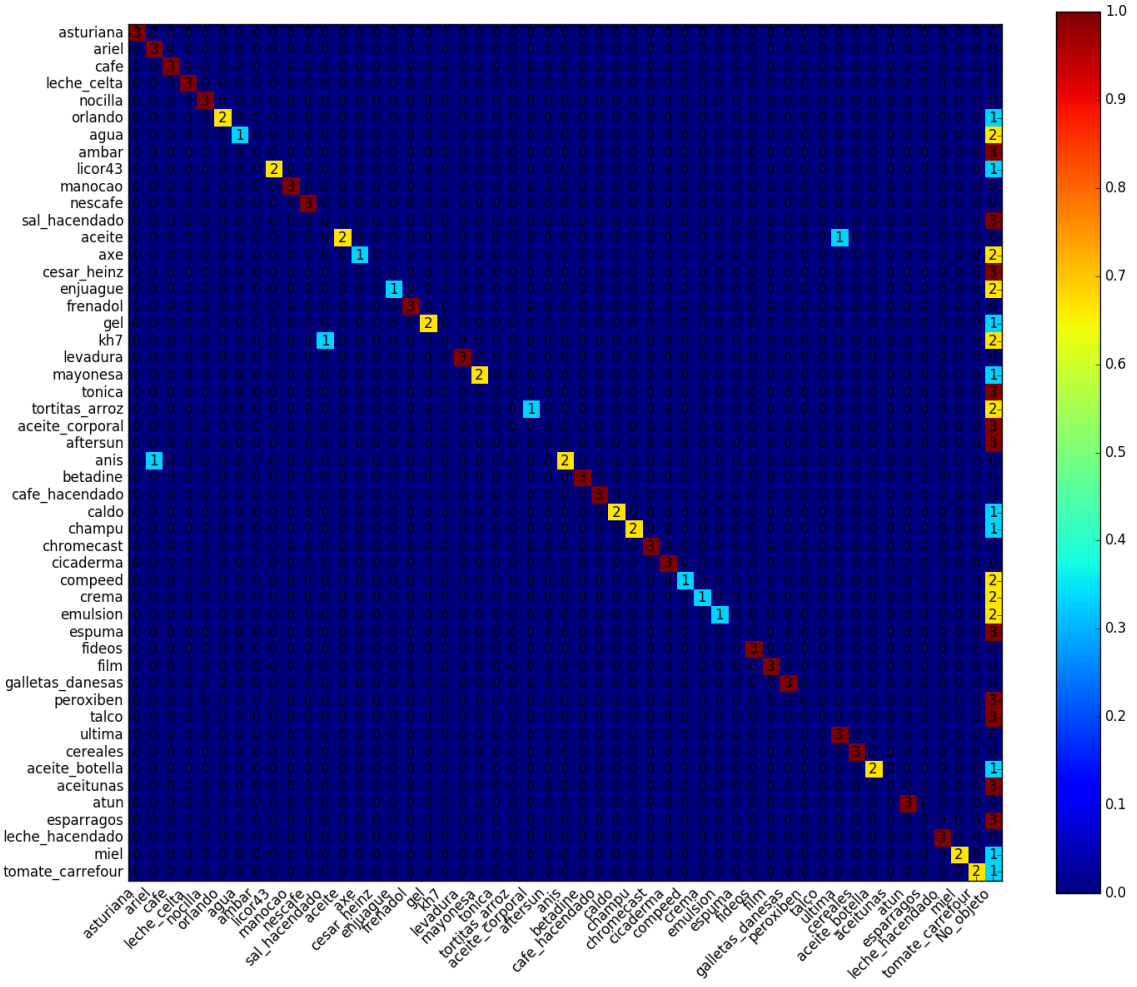


Figura 72: Matriz de confusión de los resultados obtenidos mediante ORB en Android.

Figura 73: Tabla de “True positives” de los resultados obtenidos mediante ORB en Android

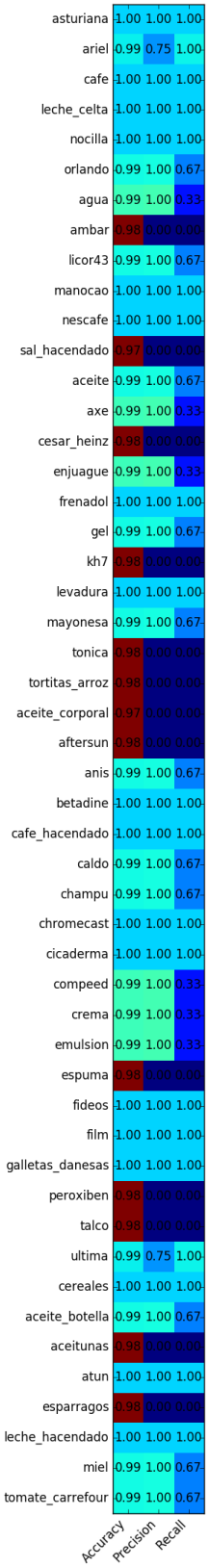


Figura 74: Tabla de precisión de los resultados obtenidos mediante ORB en Android.

- **PC. ORB/ORB.**

  Tiempo: 16.06 segs. Media puntos: 500. Media correspondencias buenas: 28

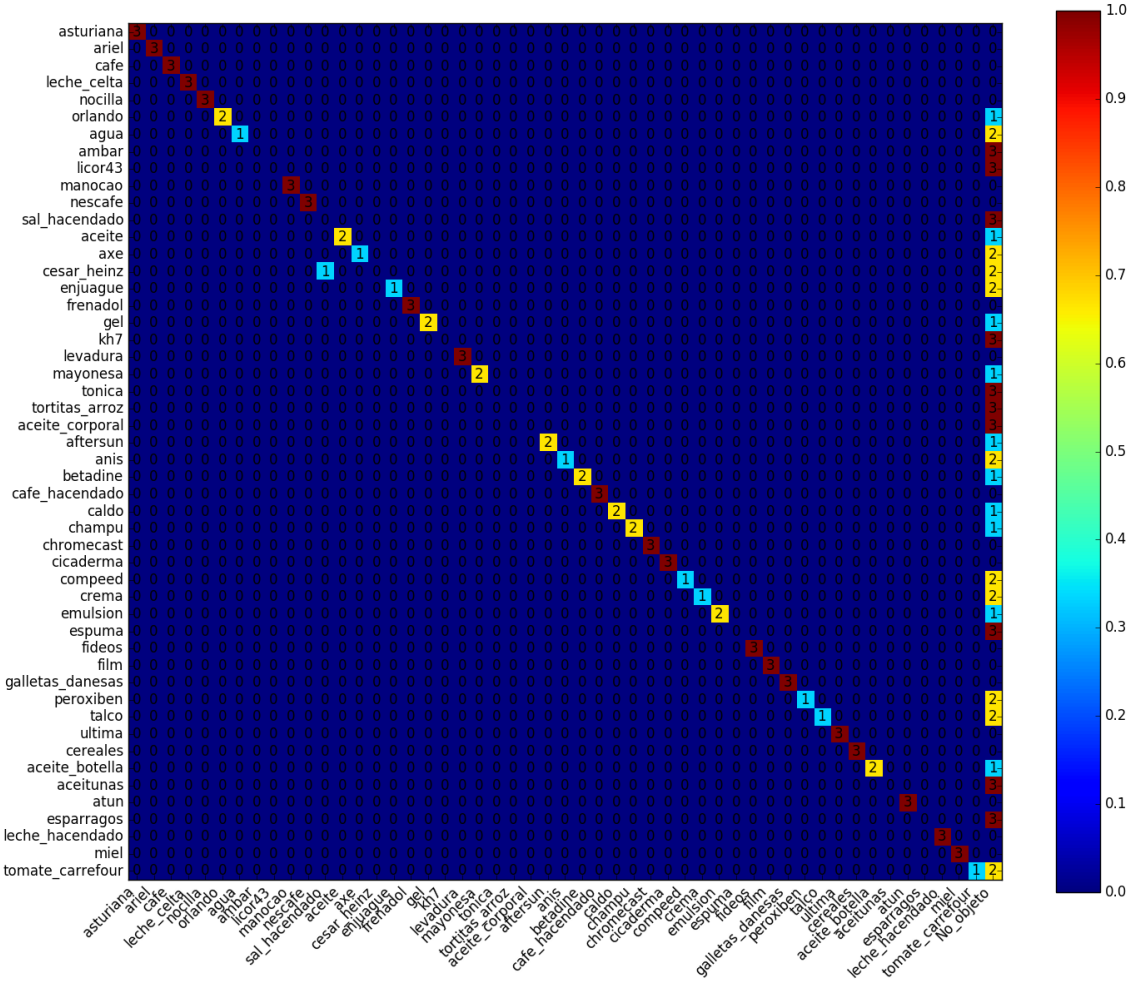


Figura 75: Matriz de confusión de los resultados obtenidos mediante ORB en PC.

Figura 76: Tabla de “True positives” de los resultados obtenidos mediante ORB en PC

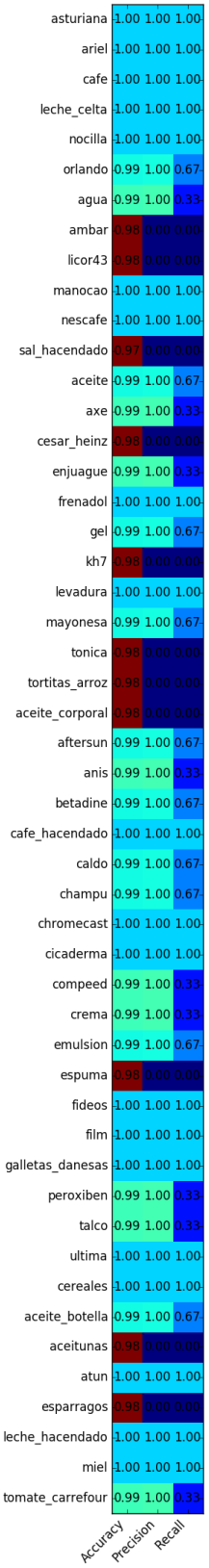


Figura 77: Tabla de precisión de los resultados obtenidos mediante ORB en PC.

- **Server. ORB/ORB. (Calidad imagen 100 %)**

  Media puntos: 500. Media correspondencias buenas: 28

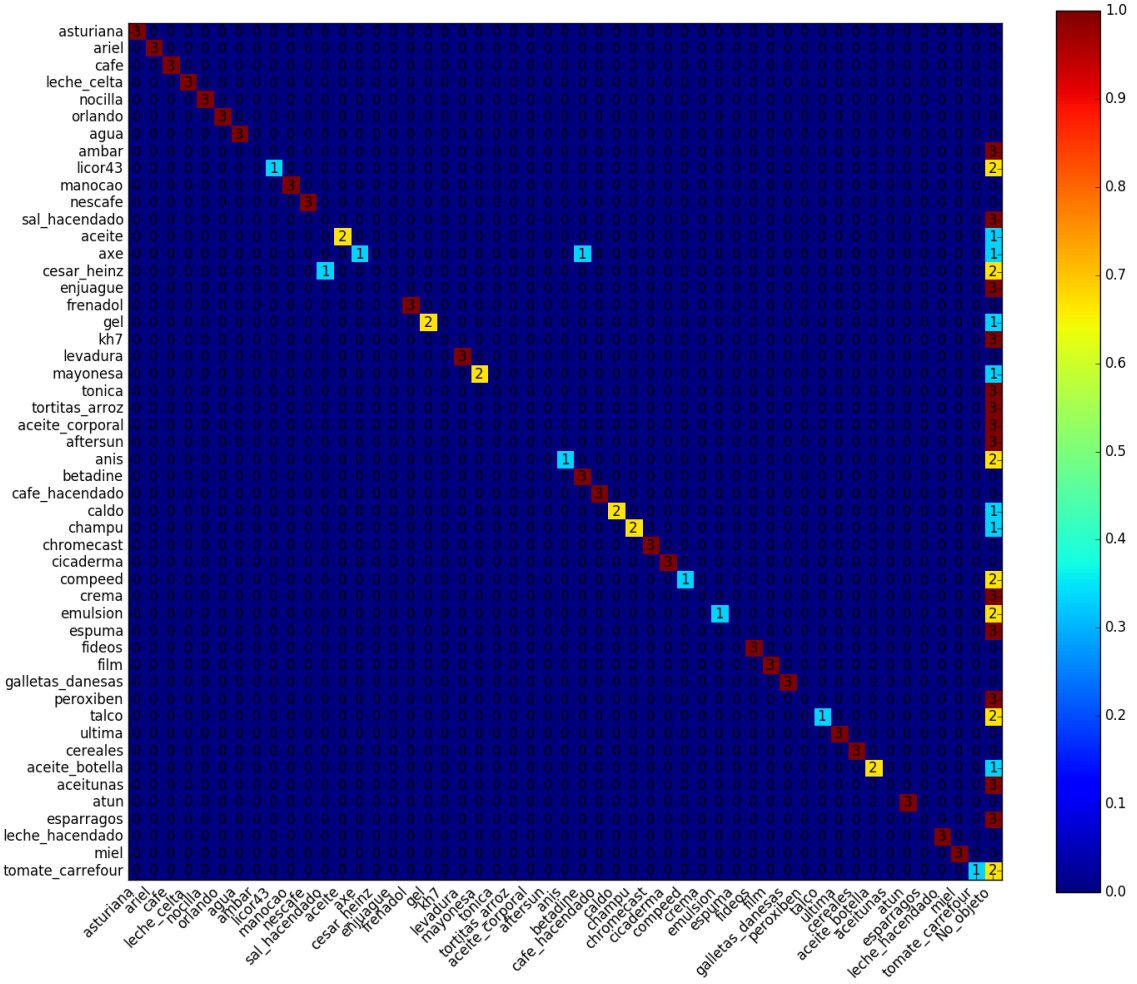


Figura 78: Matriz de confusión de los resultados obtenidos mediante ORB en el servidor.

Figura 79: Tabla de “True positives” de los resultados obtenidos mediante ORB en el servidor.

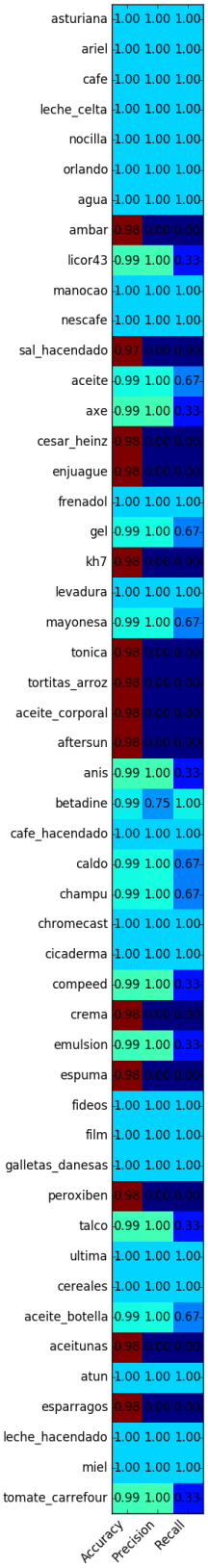


Figura 80: Tabla de precisión de los resultados obtenidos mediante ORB en el servidor.

- **Server. ORB/ORB. (Calidad imagen 90 %**

  Media puntos: 500. Media correspondencias buenas: 27

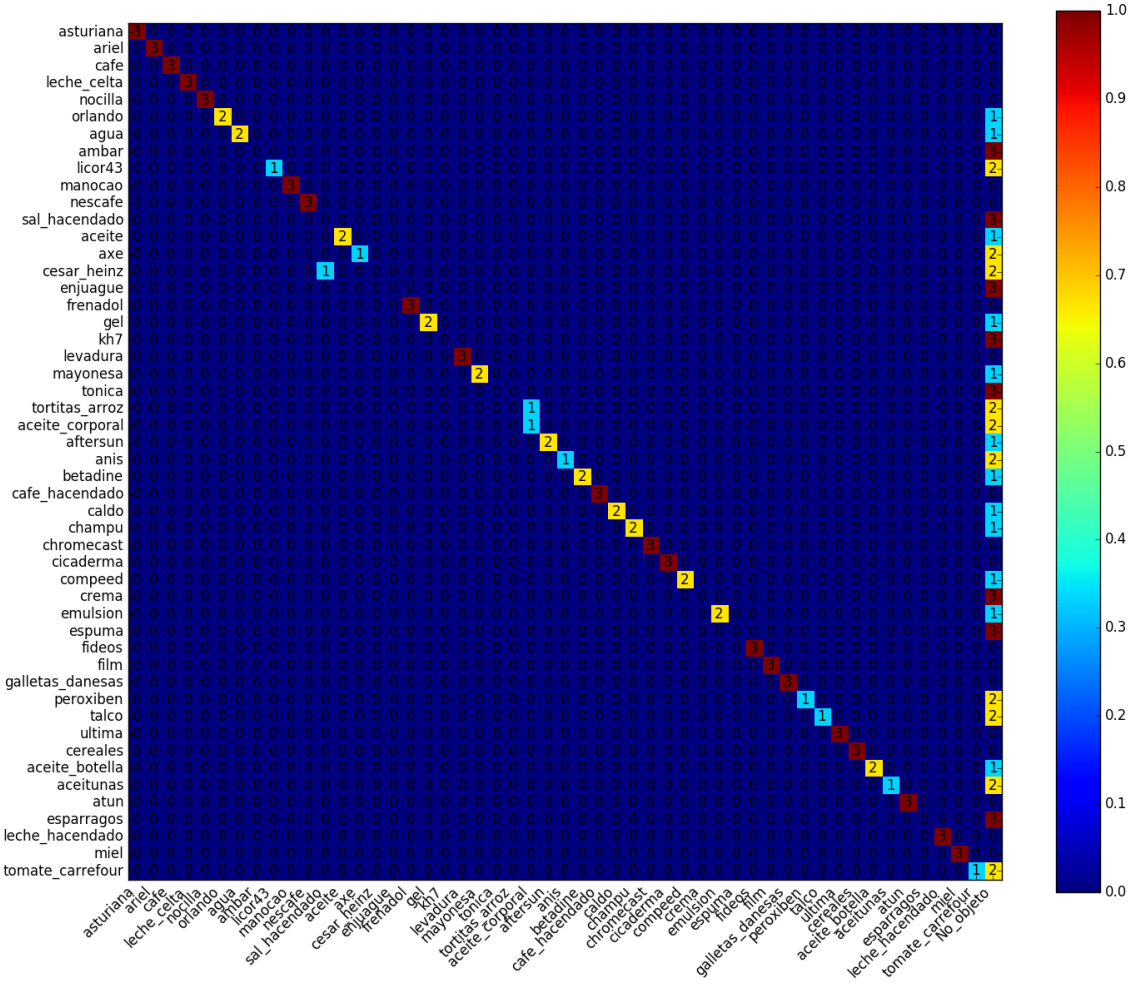


Figura 81: Matriz de confusión de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 90 %.

Figura 82: Tabla de "True positives" de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 90 %.

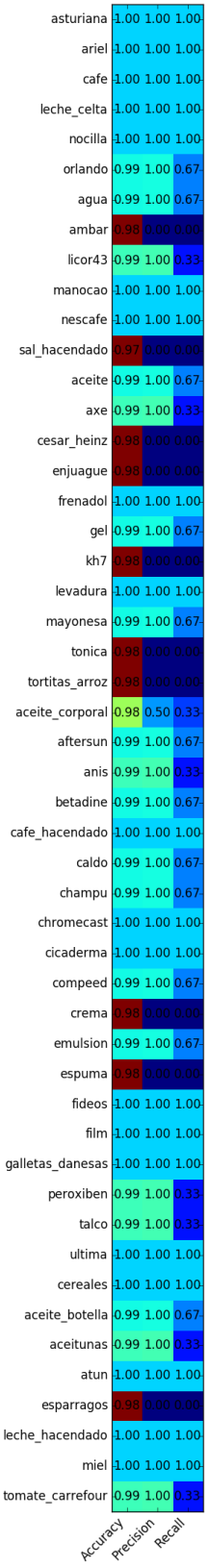


Figura 83: Tabla de precisión de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 90 %.

- **Server. ORB/ORB. (Calidad imagen 80 %**

  Media puntos: 500. Media correspondencias buenas: 27

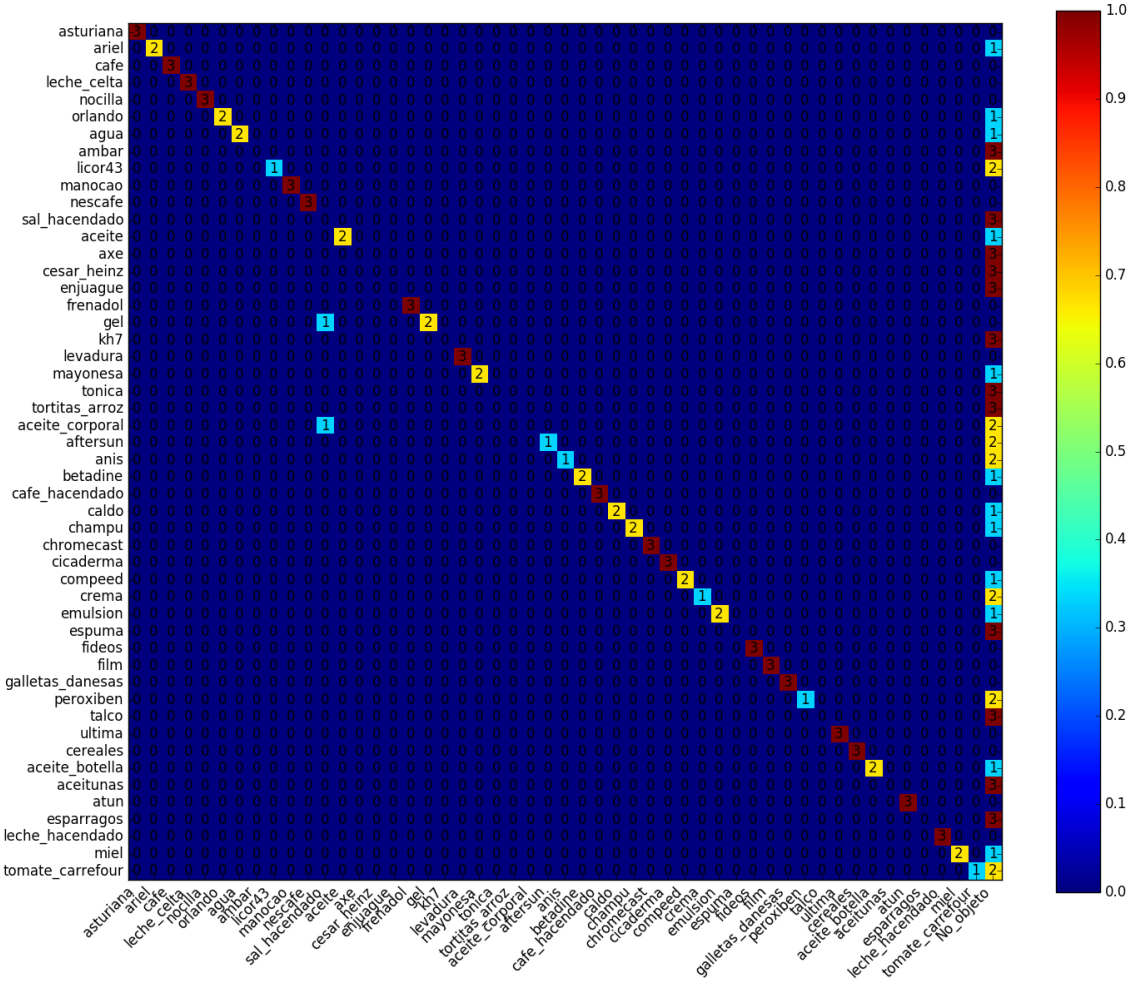


Figura 84: Matriz de confusión de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 80 %.

Figura 85: Tabla de "True positives" de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 80 %.

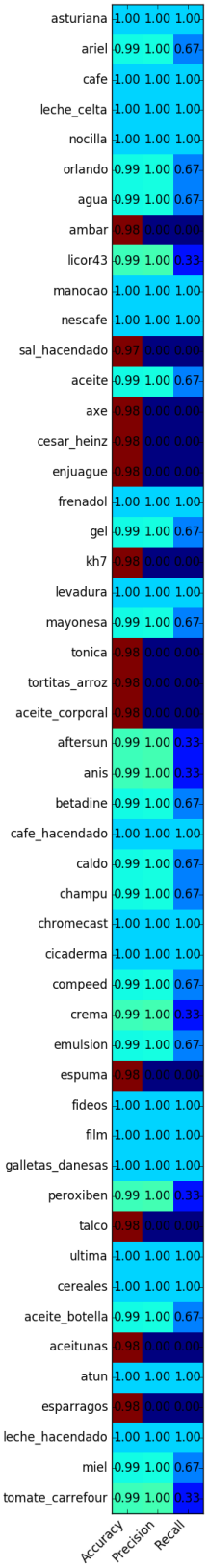


Figura 86: Tabla de precisión de los resultados obtenidos mediante ORB en el servidor, enviadas con calidad al 80 %.

- **Android. ORB(WTA_K = 4)/ORB(WTA_K = 4).**

  Tiempo: 71.33 segs. Media puntos: 500. Media correspondencias buenas: 30

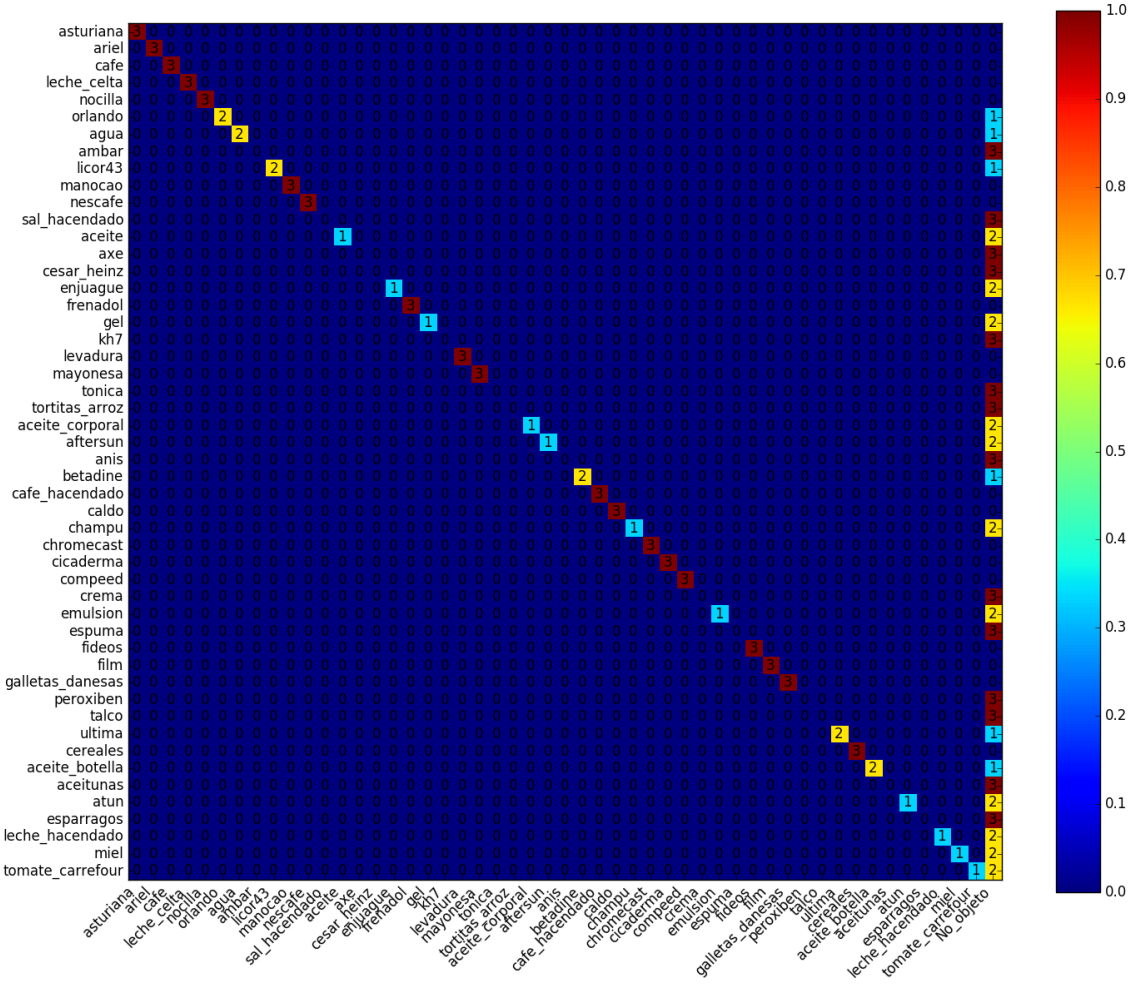


Figura 87: Matriz de confusión de los resultados obtenidos mediante ORB (WTA_K = 4) en Android.

Figura 88: Tabla de "True positives" de los resultados obtenidos mediante ORB (WTA_K = 4) en Android

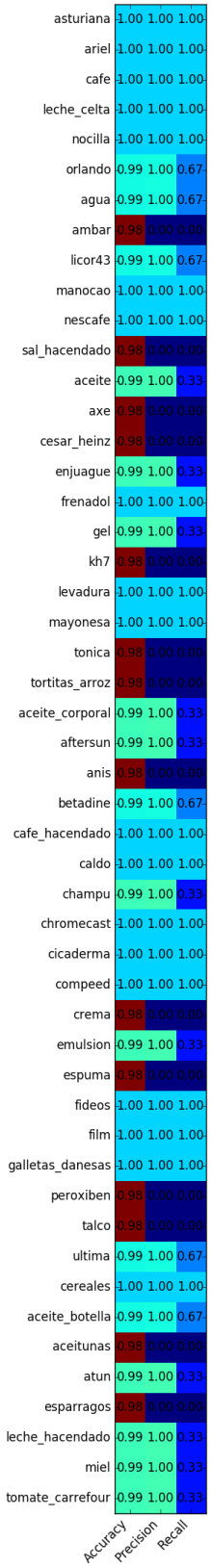


Figura 89: Tabla de precisión de los resultados obtenidos mediante ORB (WTA_K = 4) en Android.

- **PC. ORB(WTA_K = 4)/ORB(WTA_K = 4).**

  Tiempo: 18.08 segs. Media puntos: 500. Media correspondencias buenas: 30

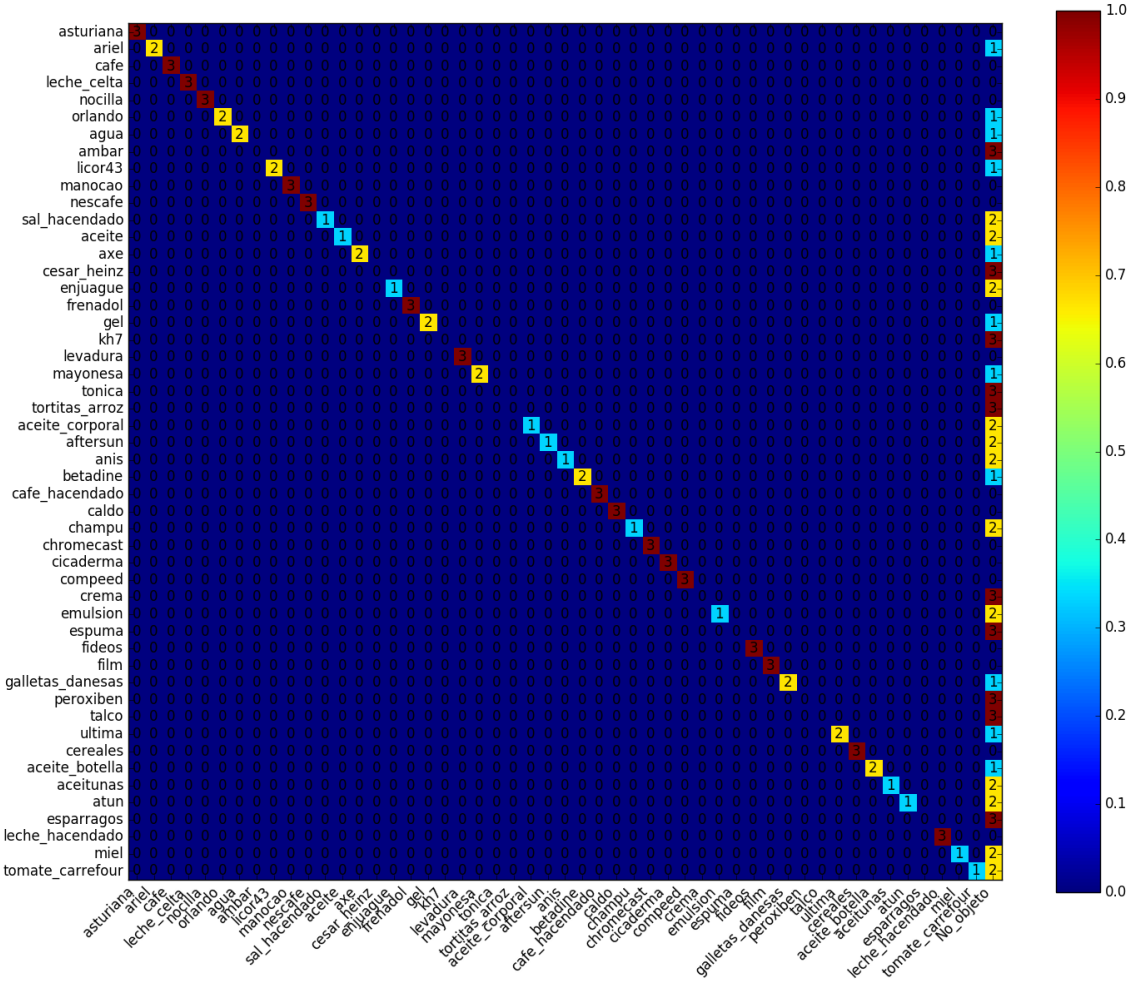


Figura 90: Matriz de confusión de los resultados obtenidos mediante ORB(WTA_K = 4) en PC.

Figura 91: Tabla de "True positives" de los resultados obtenidos mediante ORB(WTA_K = 4) en PC

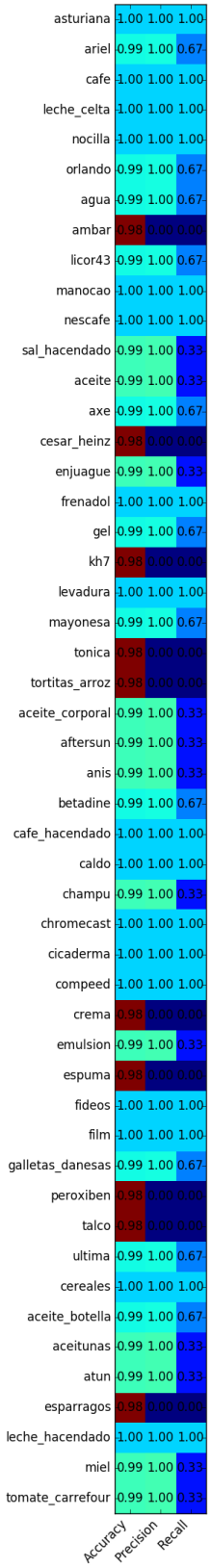


Figura 92: Tabla de precisión de los resultados obtenidos mediante ORB(WTA_K = 4) en PC.

- **Server. ORB(WTA_K = 4)/ORB(WTA_K = 4).**

  Media puntos: 500. Media correspondencias buenas: 27

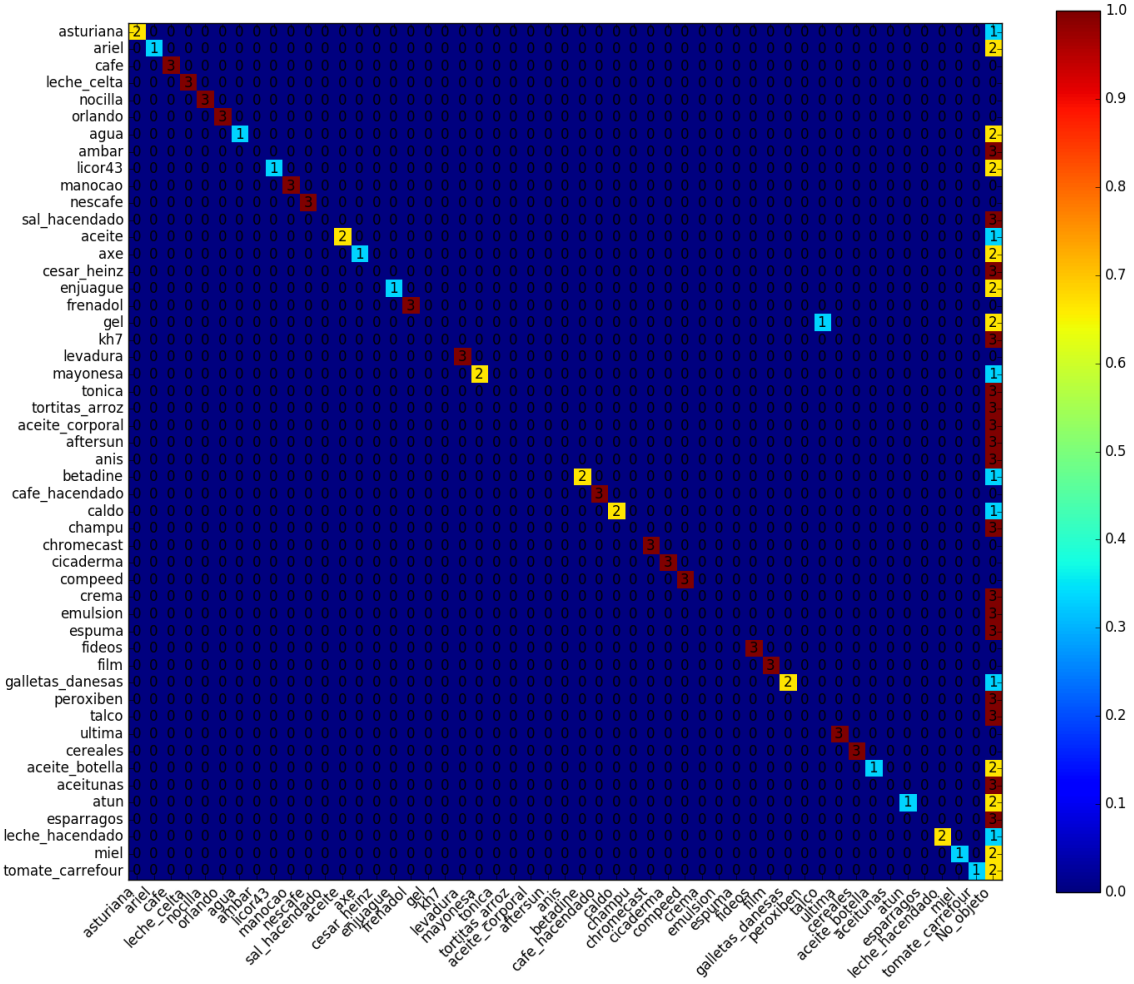


Figura 93: Matriz de confusión de los resultados obtenidos mediante ORB(WTA_K = 4) en el servidor.

Figura 94: Tabla de “True positives” de los resultados obtenidos mediante ORB(WTA_K = 4) el servidor

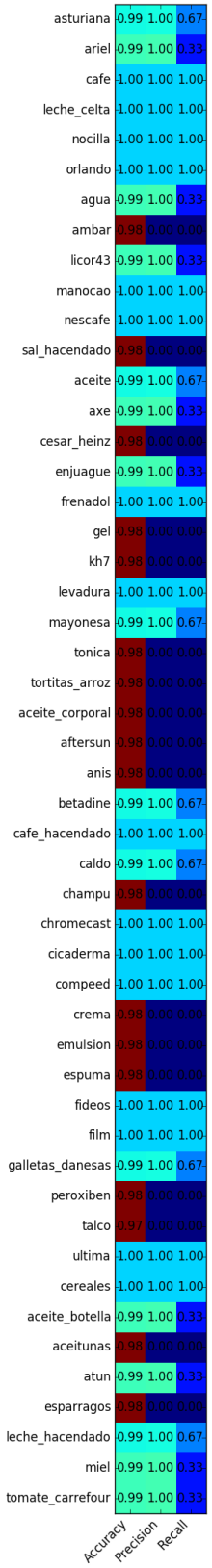


Figura 95: Tabla de precisión de los resultados obtenidos mediante ORB(WTA_K = 4) en el servidor.

- **Android. BRISK/BRISK.**

  Tiempo: 502.7 segs. Media puntos: 1855. Media correspondencias buenas: 63

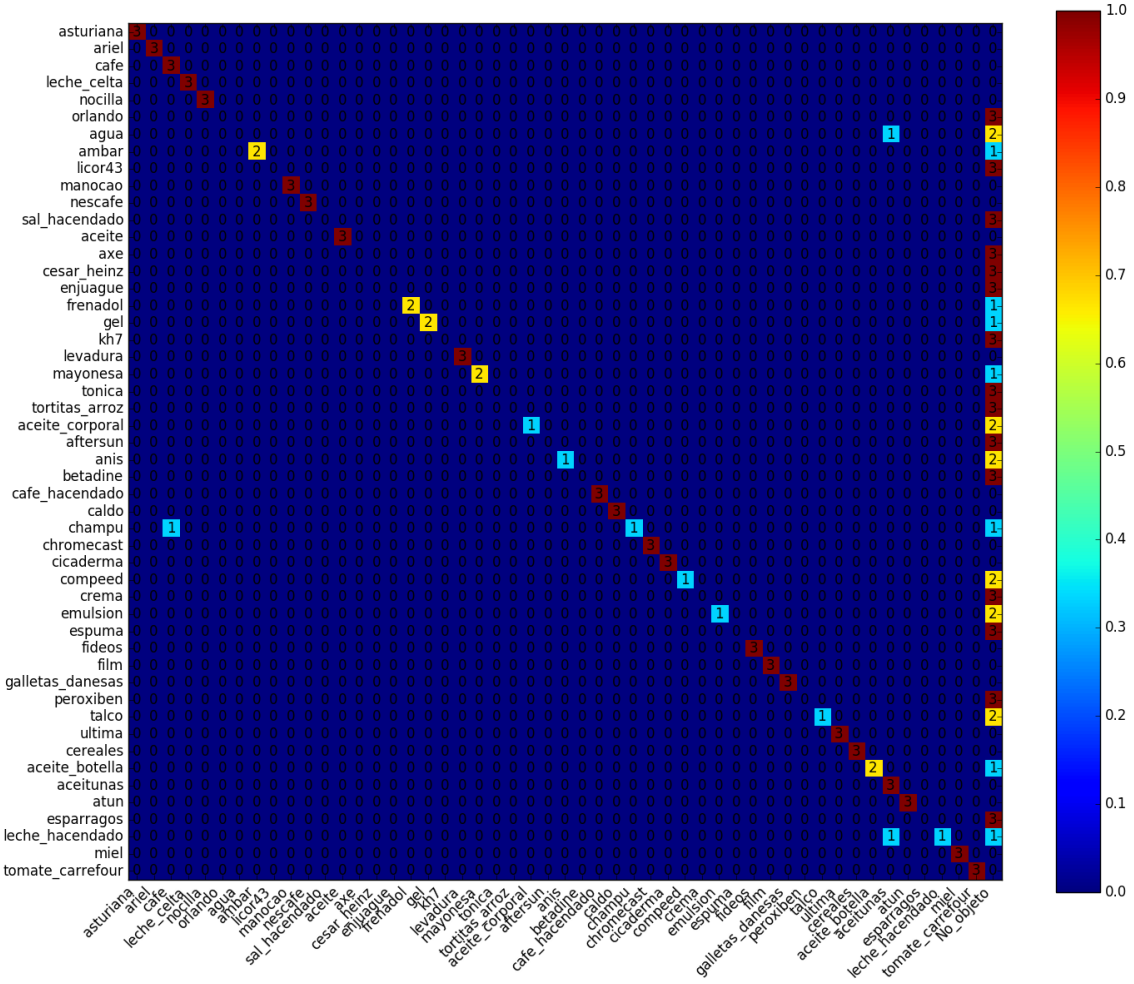


Figura 96: Matriz de confusión de los resultados obtenidos mediante BRISK en Android.

Figura 97: Tabla de “True positives” de los resultados obtenidos mediante BRISK en Android

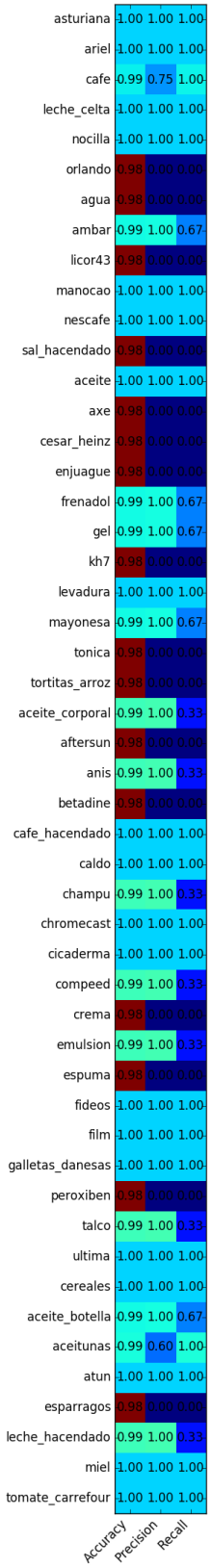


Figura 98: Tabla de precisión de los resultados obtenidos mediante BRISK en Android.

- **PC. BRISK/BRISK.**

  Tiempo: 353.50 segs. Media puntos: 1855. Media correspondencias buenas: 63

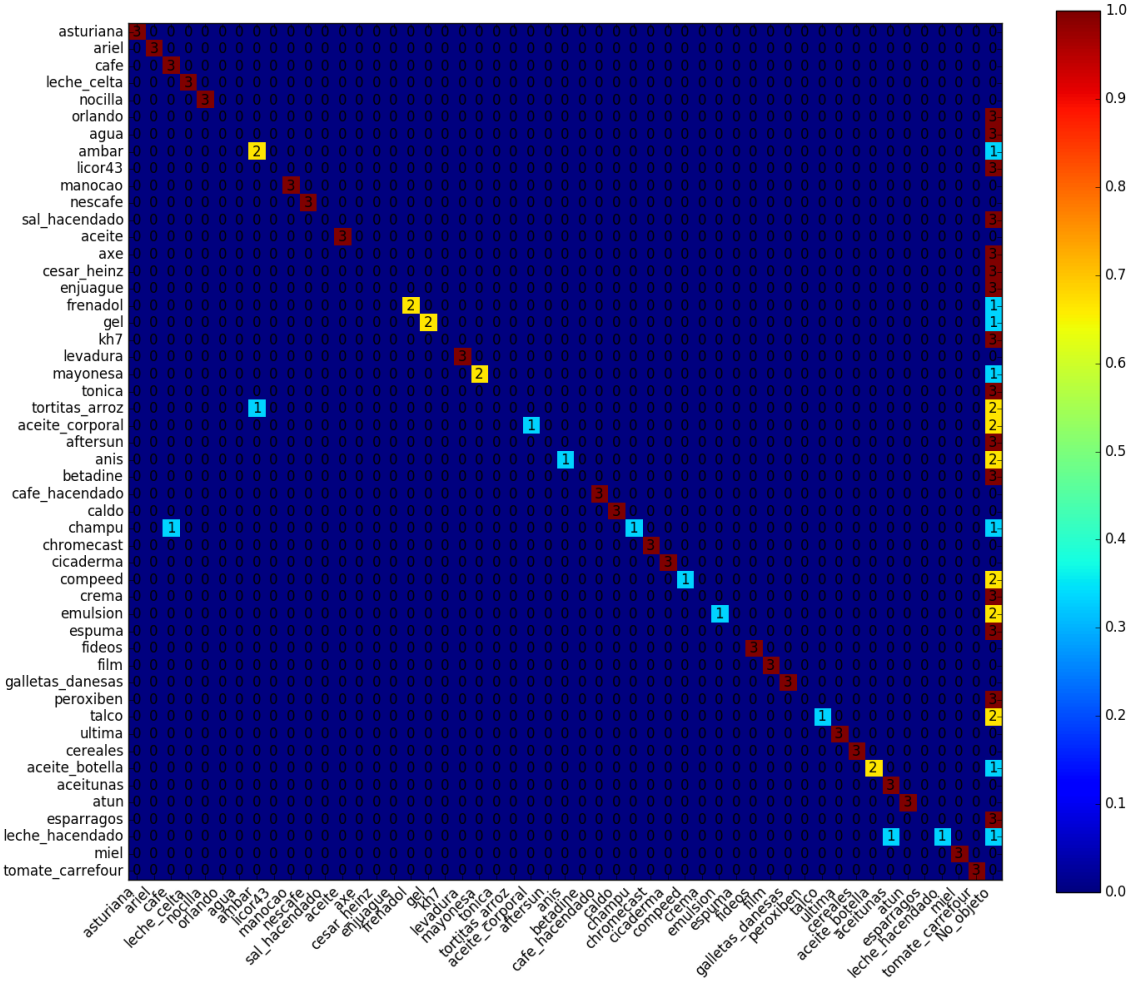


Figura 99: Matriz de confusión de los resultados obtenidos mediante BRISK en PC.

Figura 100: Tabla de “True positives” de los resultados obtenidos mediante BRISK en PC

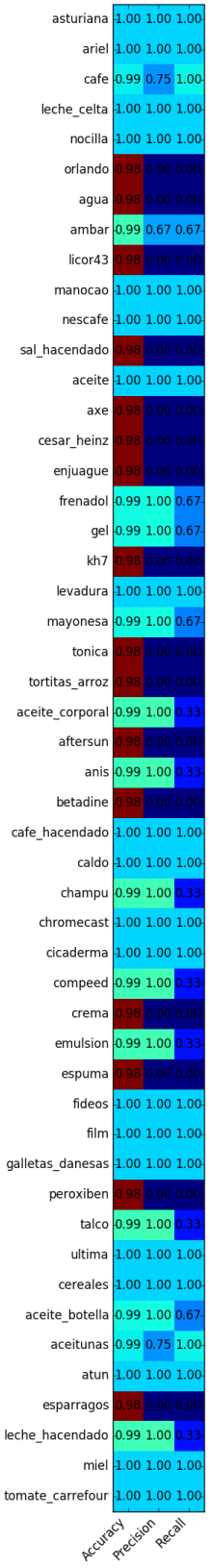


Figura 101: Tabla de precisión de los resultados obtenidos mediante BRISK en PC.

- **Android. BRISK(thres = 85)/BRISK(thres = 85).**

  Tiempo:44.4 segs. Media puntos: 331. Media correspondencias buenas: 23

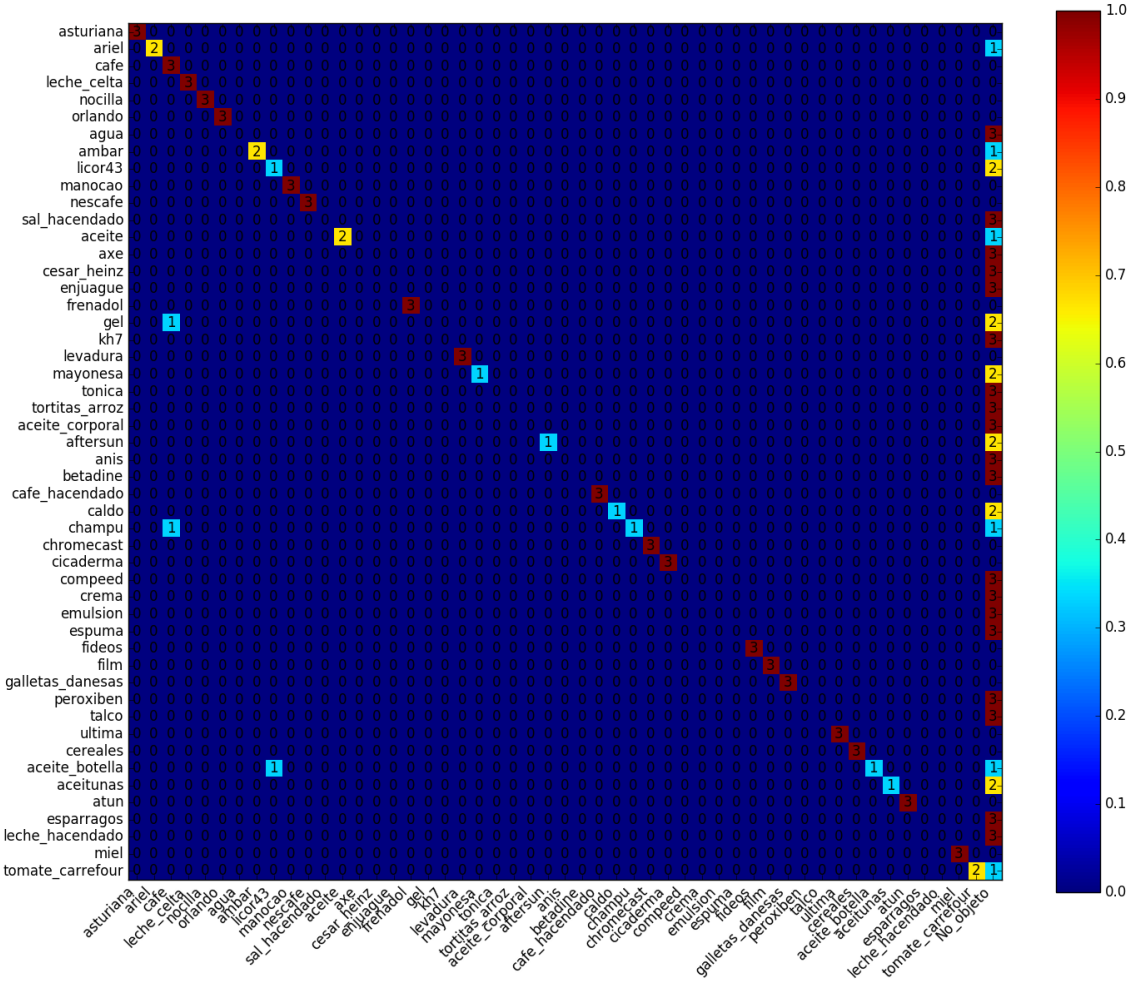


Figura 102: Matriz de confusión de los resultados obtenidos mediante BRISK(thres = 85) en Android.

Figura 103: Tabla de "True positives" de los resultados obtenidos mediante BRISK(thres = 85) en Android

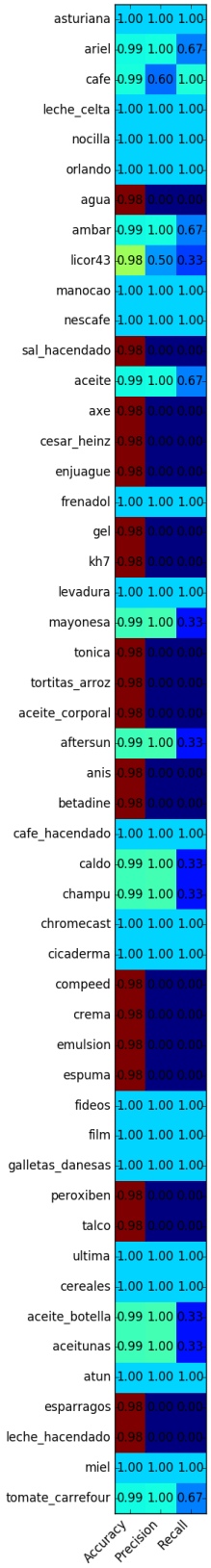


Figura 104: Tabla de precisión de los resultados obtenidos mediante BRISK(thres = 85) en Android.

- **PC. BRISK(thres = 85)/BRISK(thres = 85).**

  Tiempo: 19.75 segs. Media puntos: 331. Media correspondencias buenas: 23

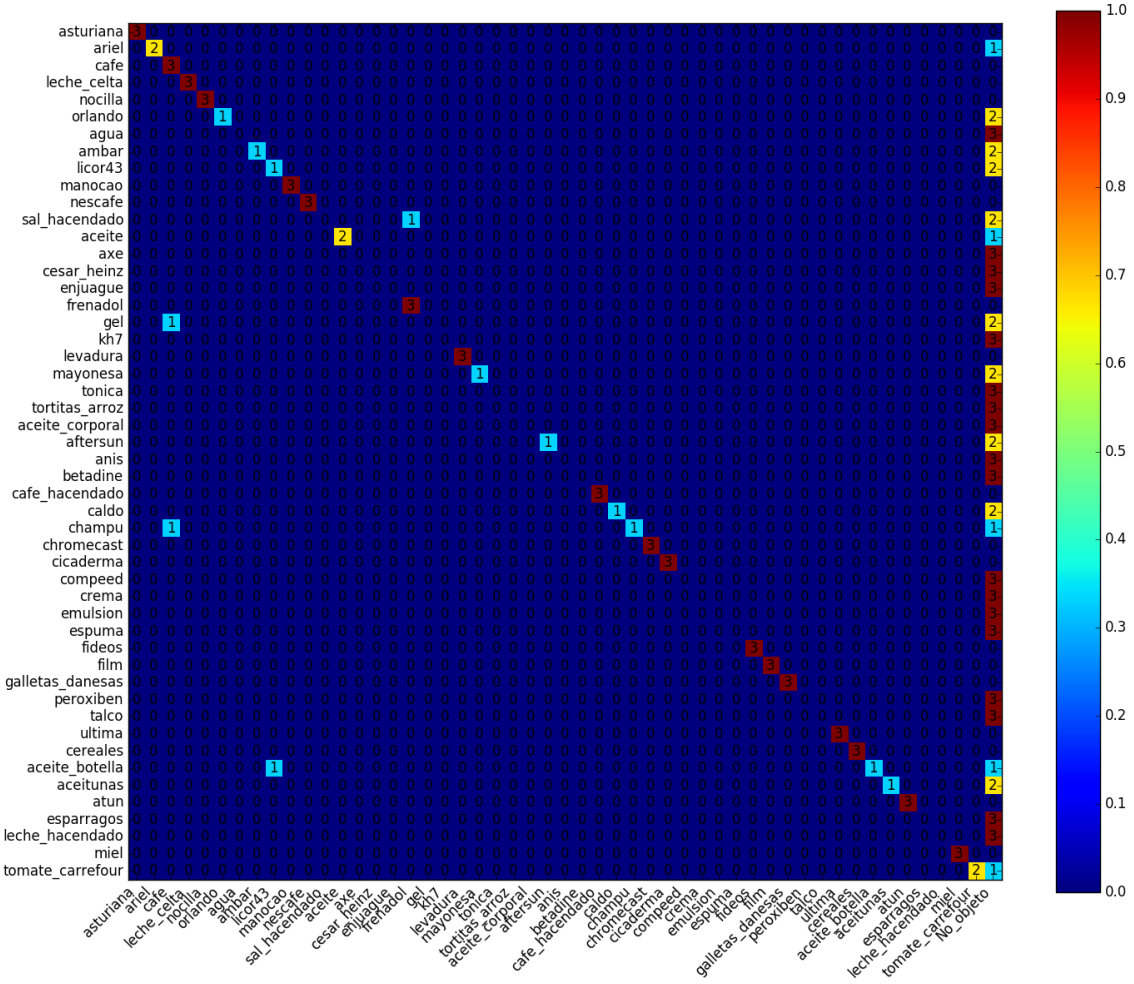


Figura 105: Matriz de confusión de los resultados obtenidos mediante BRISK(thres = 85) en PC.

Figura 106: Tabla de "True positives" de los resultados obtenidos mediante BRISK(thres = 85) en PC

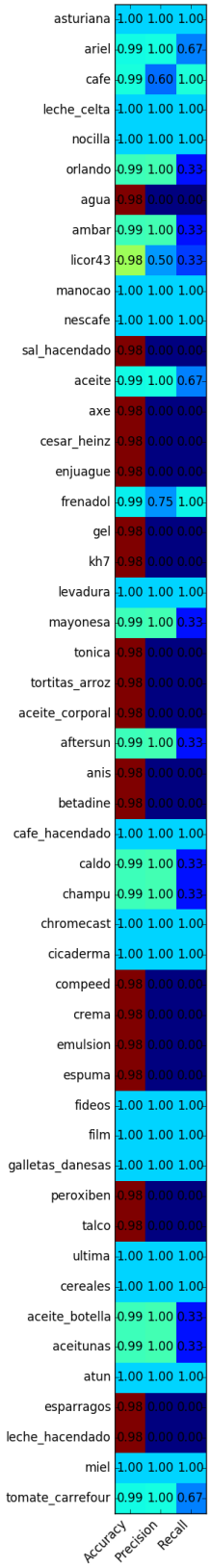


Figura 107: Tabla de precisión de los resultados obtenidos mediante BRISK(thres = 85) en PC.

F.3. Experimentos algoritmos de reconocimiento basados en *Deep Learning*.

Estos experimentos se realizaron con la base de datos final (50 objetos) y en modo cliente-servidor.

- **Capa fc7. Imágenes sin retocar.**

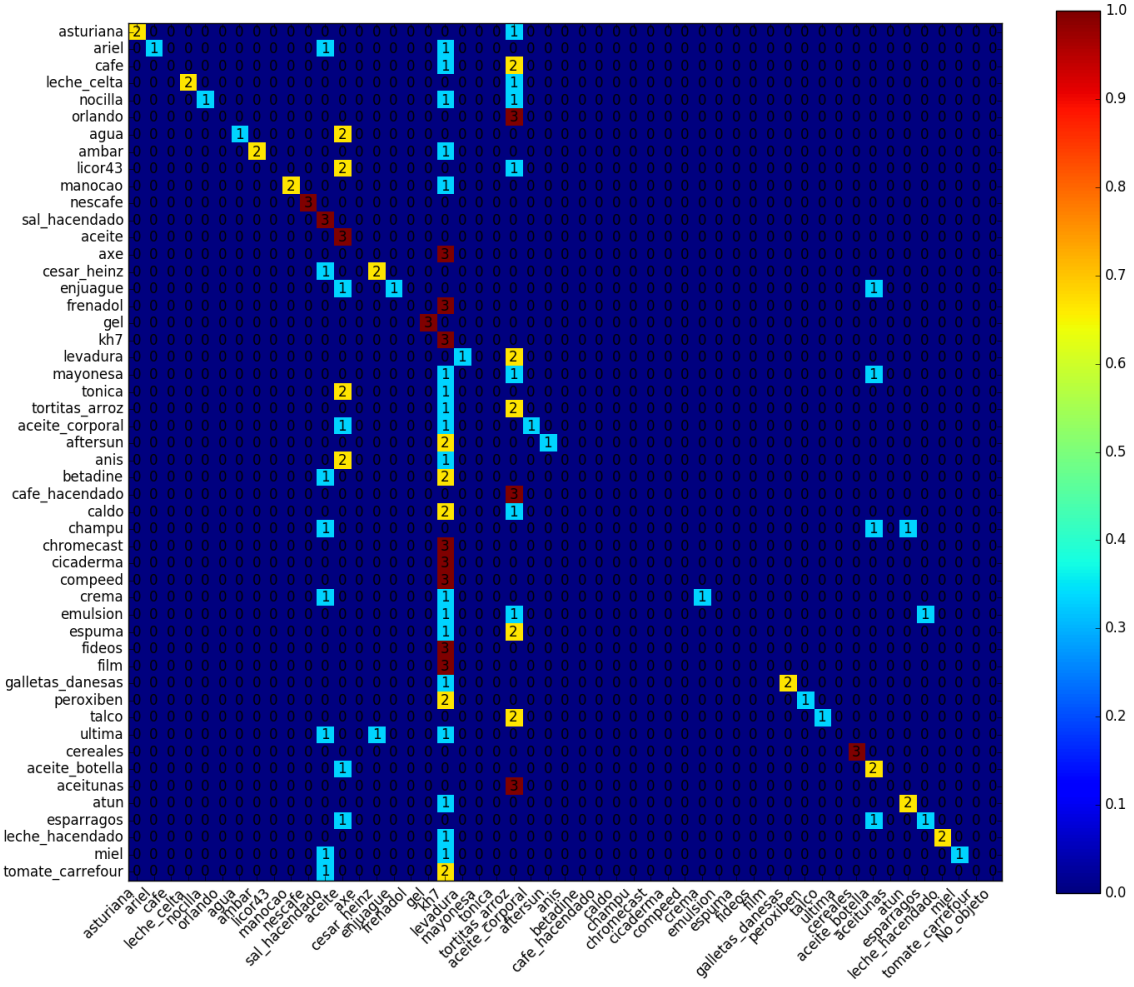


Figura 108: Matriz de confusión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal.

Figura 109: Tabla de “True positives” de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal.

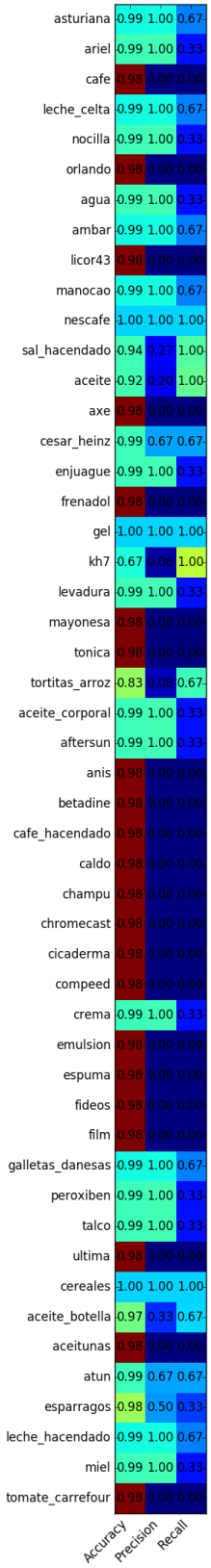


Figura 110: Tabla de precisión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal.

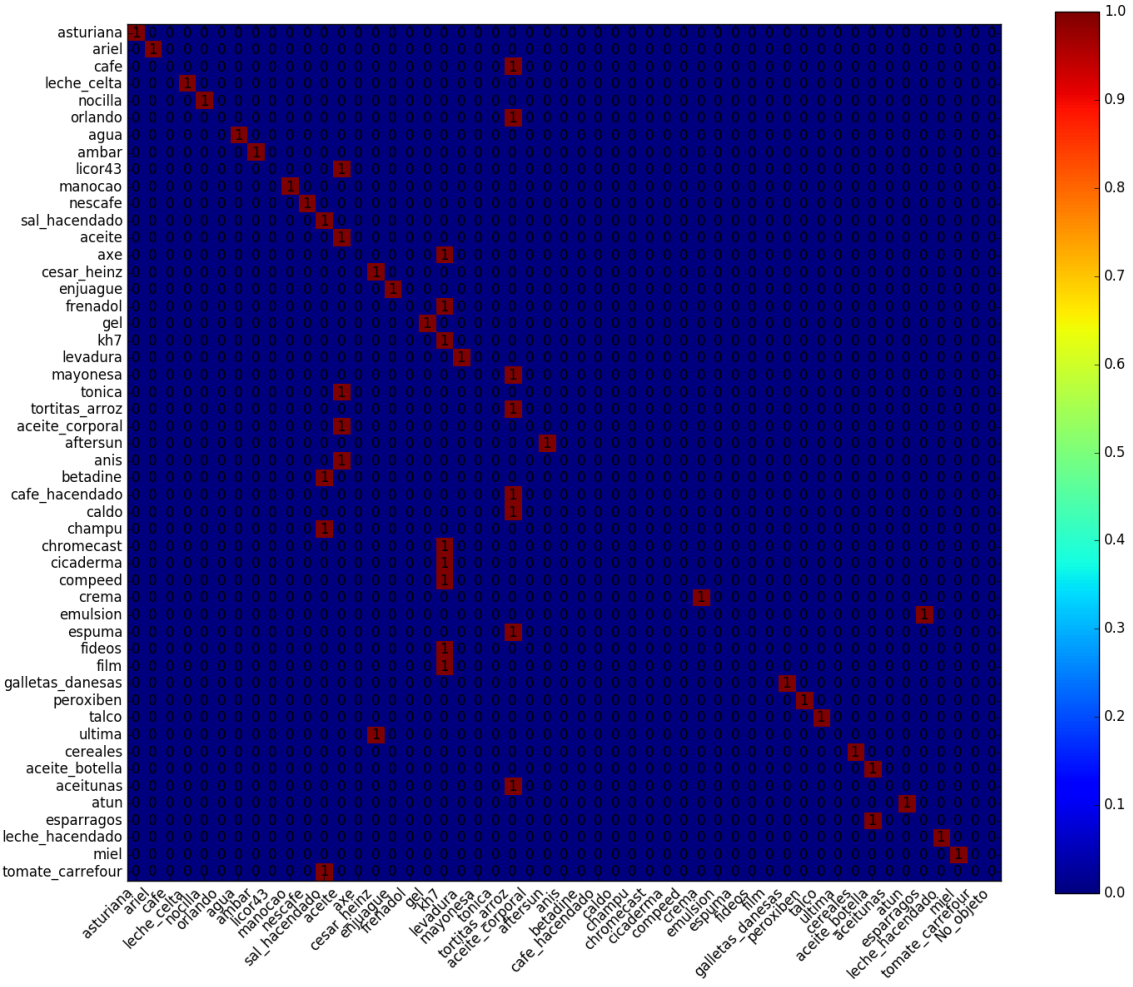


Figura 111: Matriz de confusión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal.

Figura 112: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal.

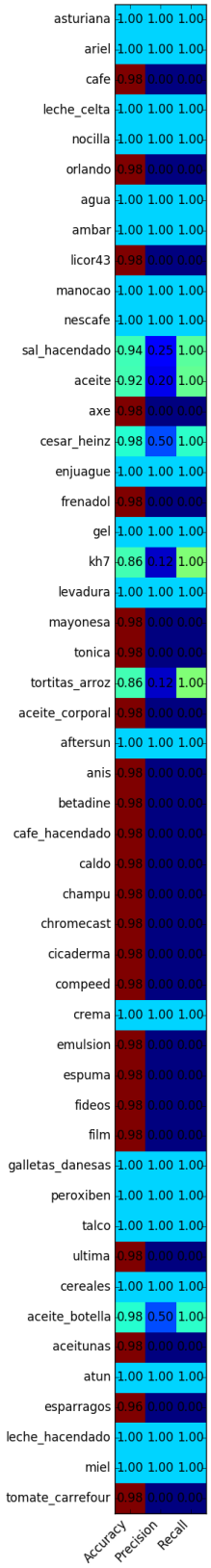


Figura 113: Tabla de precisión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal.

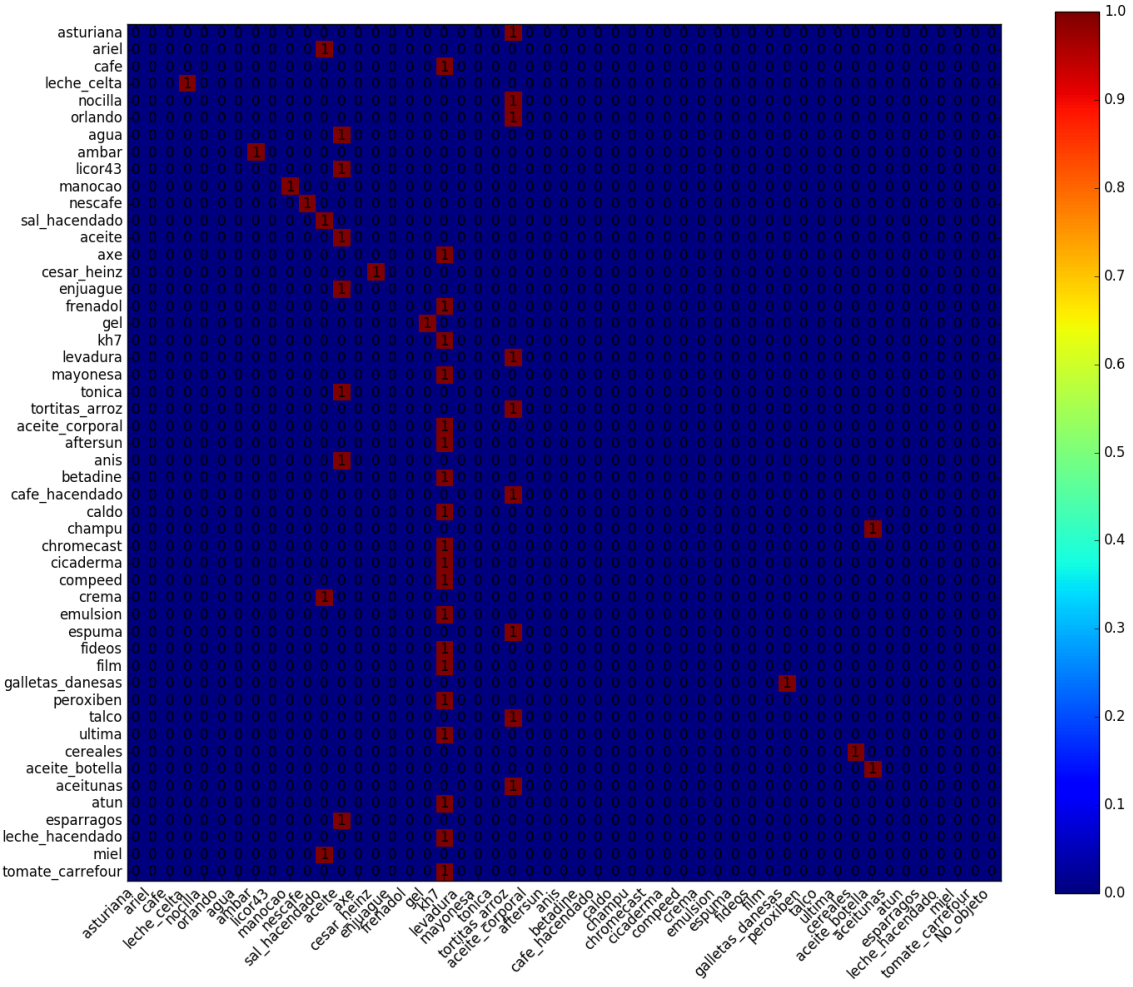


Figura 114: Matriz de confusión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal.

Figura 115: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal.

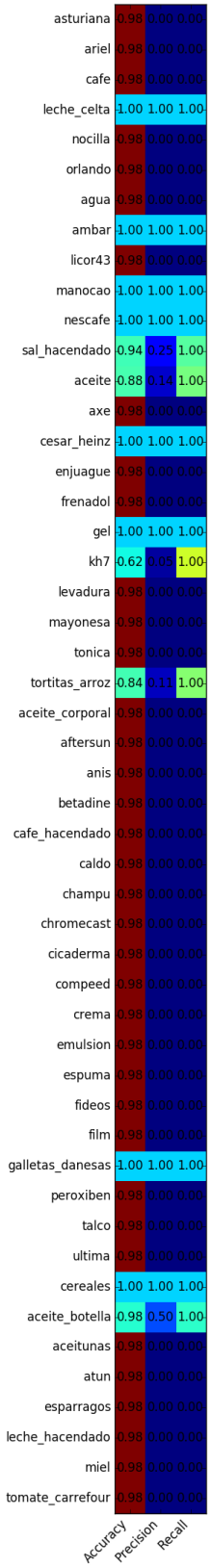


Figura 116: Tabla de precisión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal.

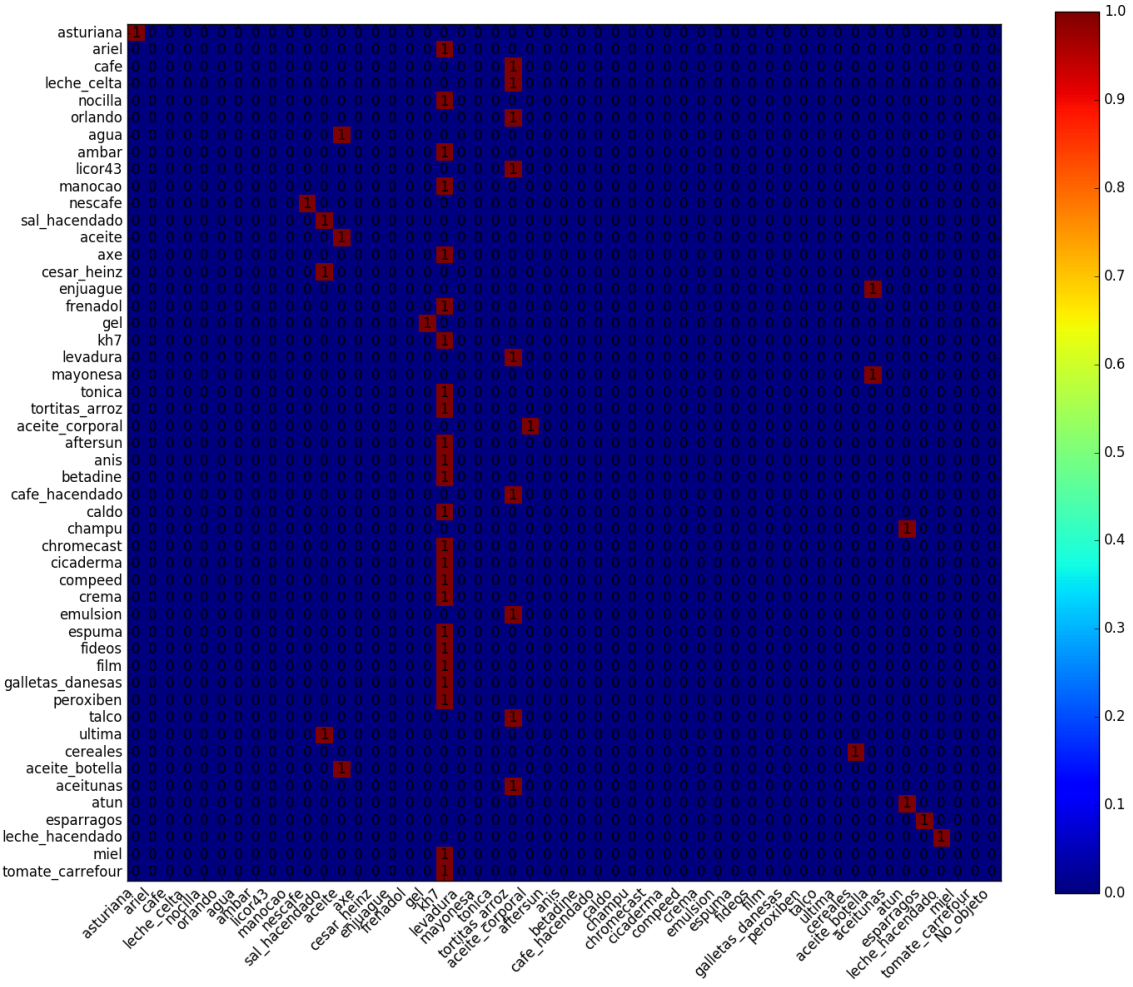


Figura 117: Matriz de confusión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal.

Figura 118: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal.

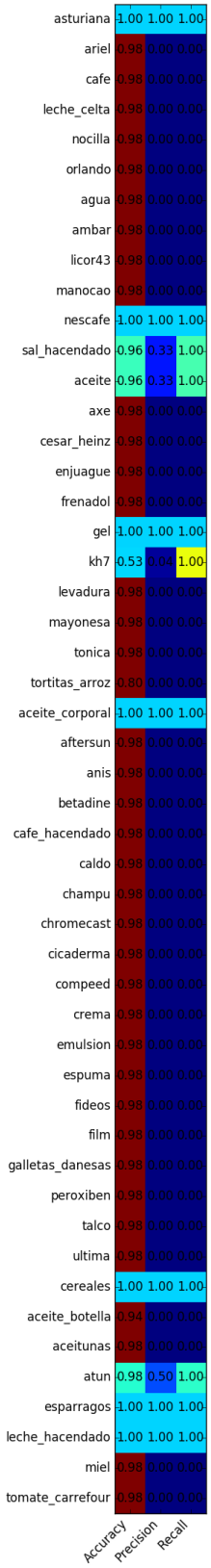


Figura 119: Tabla de precisión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal.

- **Capa fc8. Imágenes sin retocar.**

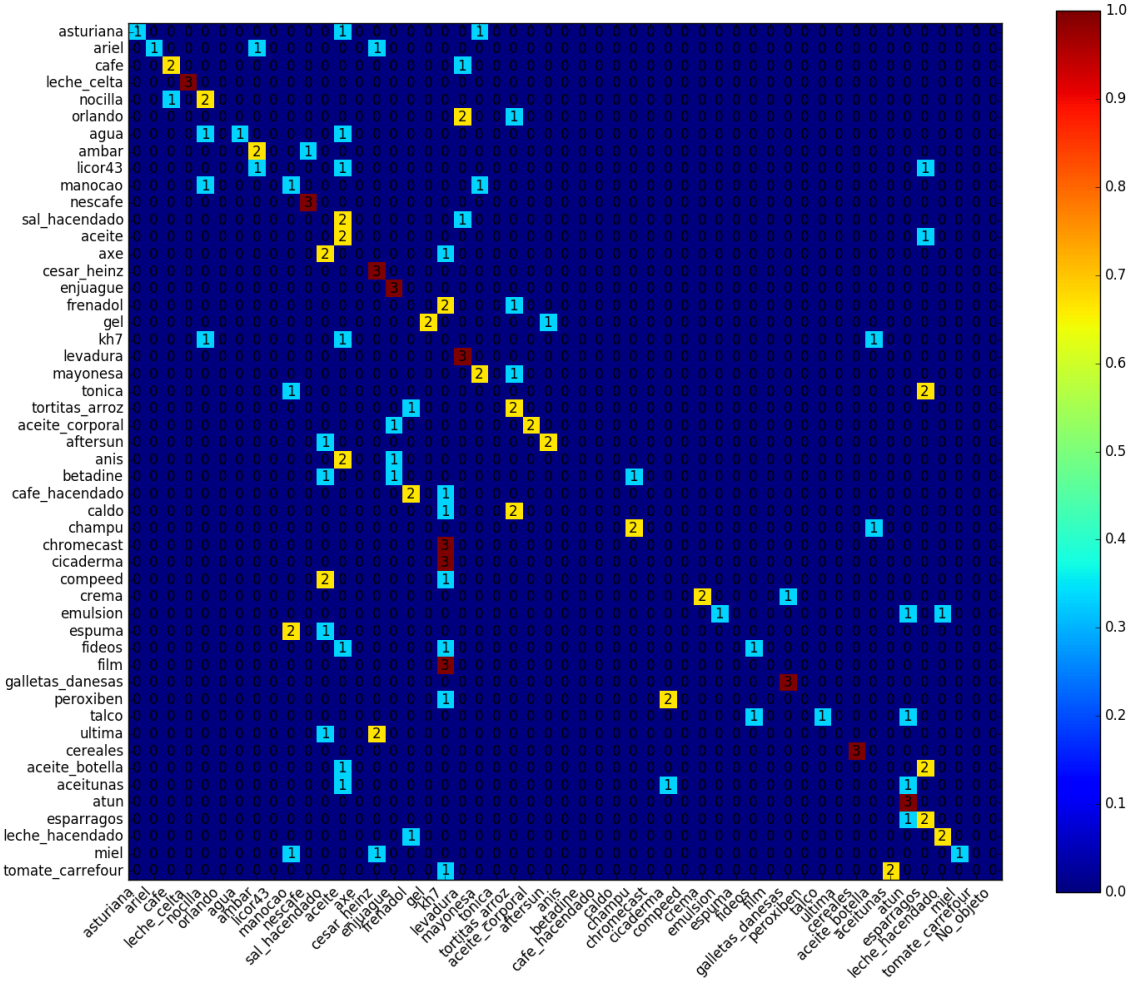


Figura 120: Matriz de confusión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal.

Figura 121: Tabla de “True positives” de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal.

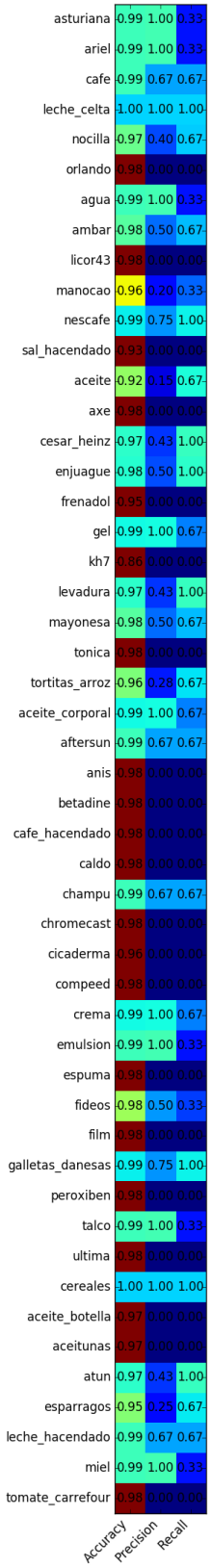


Figura 122: Tabla de precisión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal.

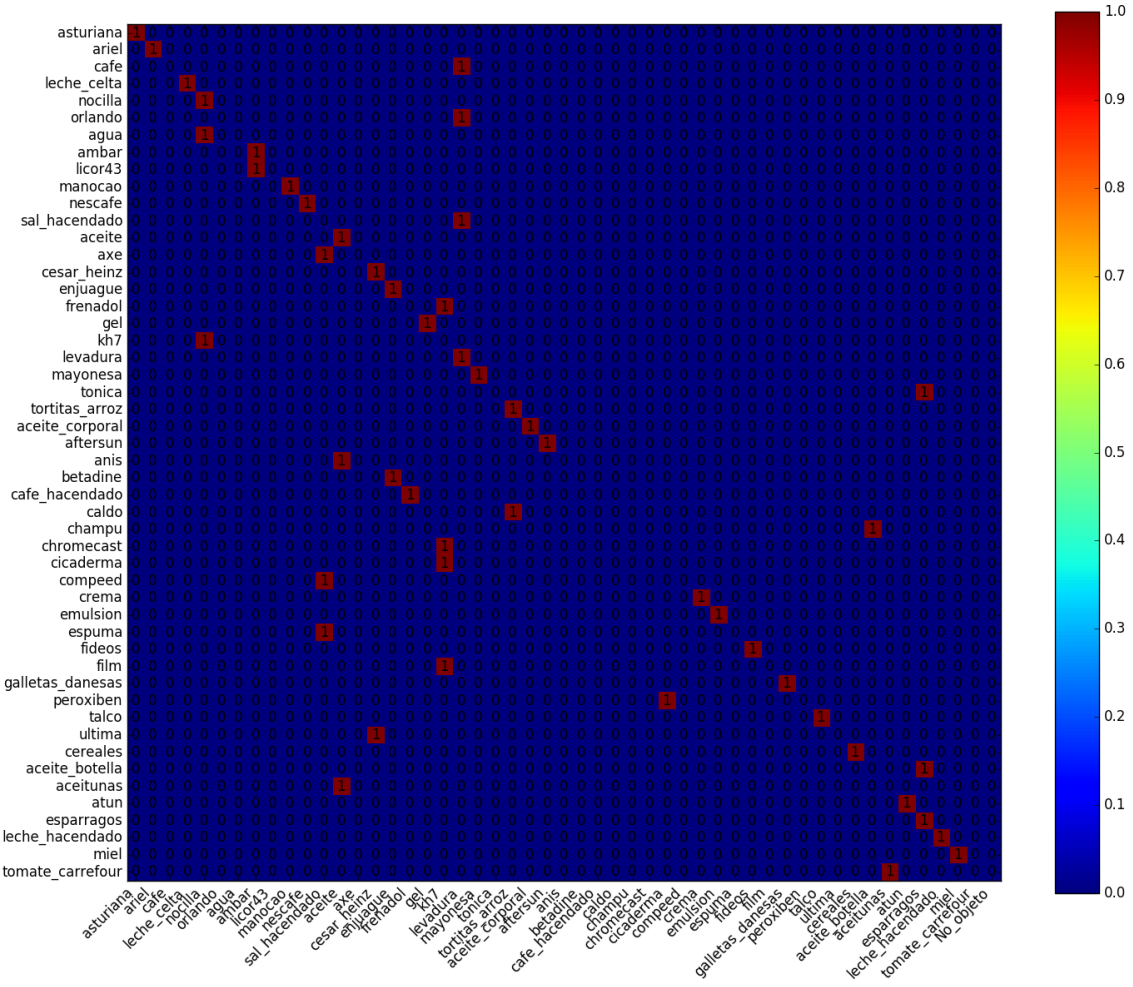


Figura 123: Matriz de confusión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal.

Figura 124: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal.

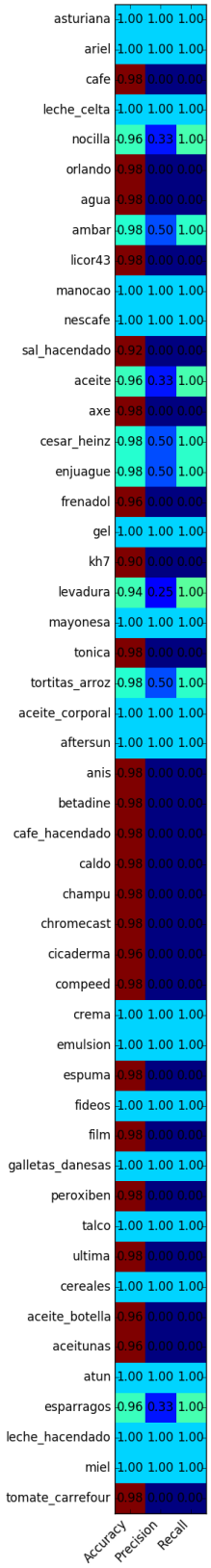


Figura 125: Tabla de precisión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal.

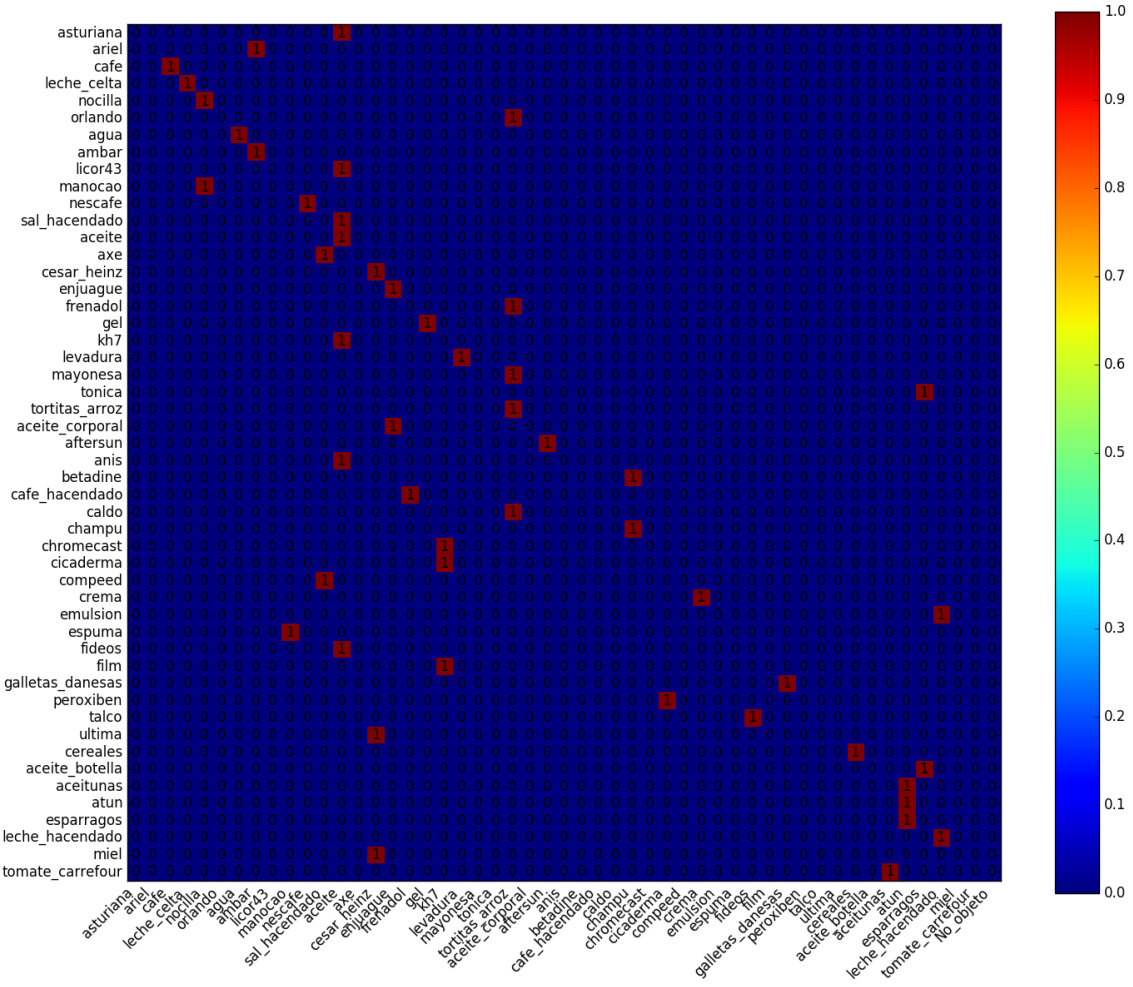


Figura 126: Matriz de confusión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal.

Figura 127: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal.

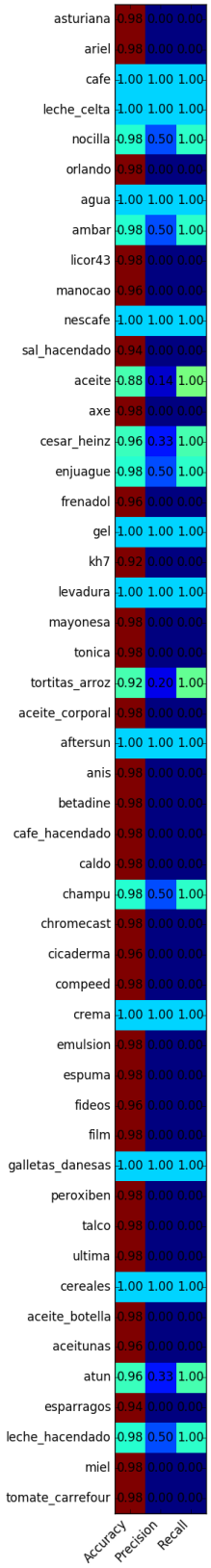


Figura 128: Tabla de precisión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal.

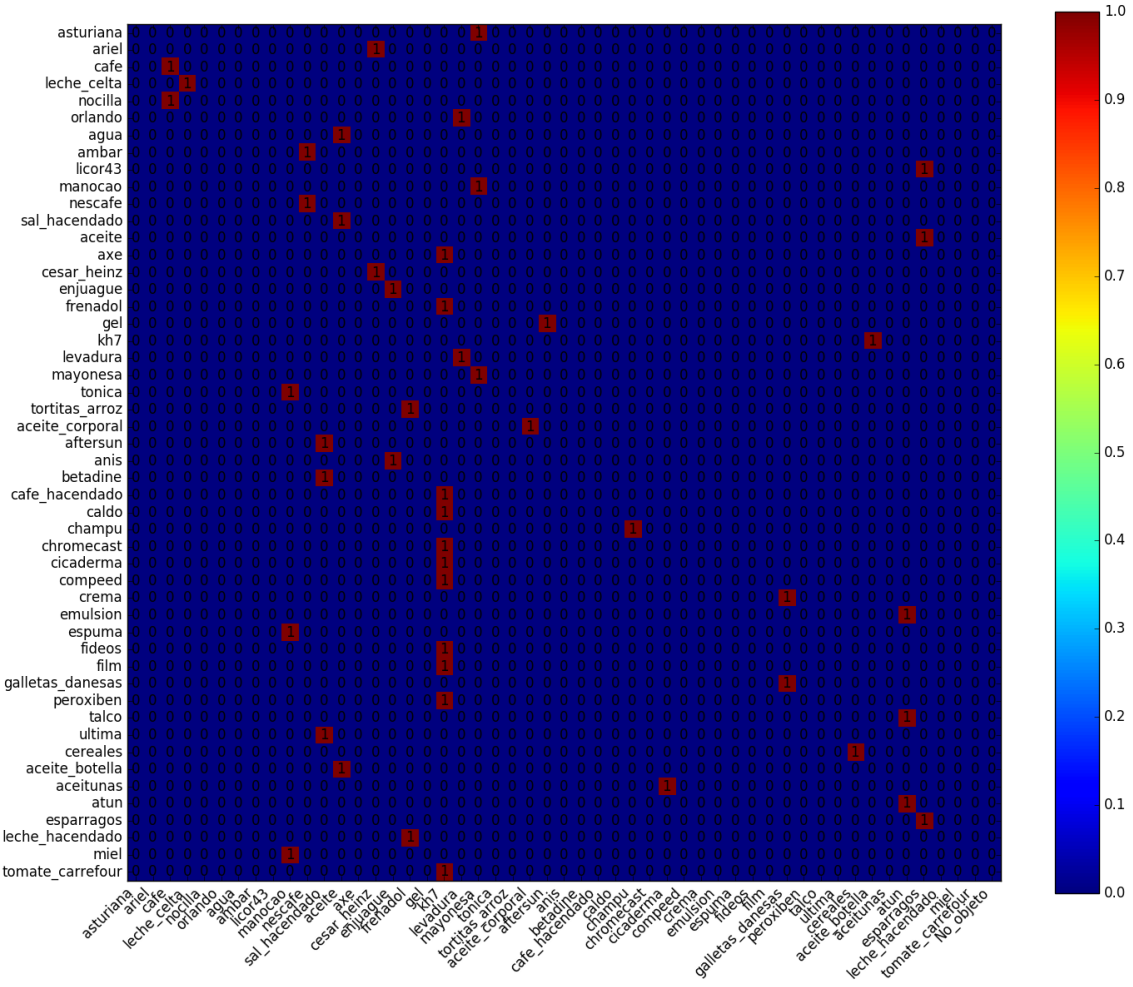


Figura 129: Matriz de confusión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal.

Figura 130: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal.

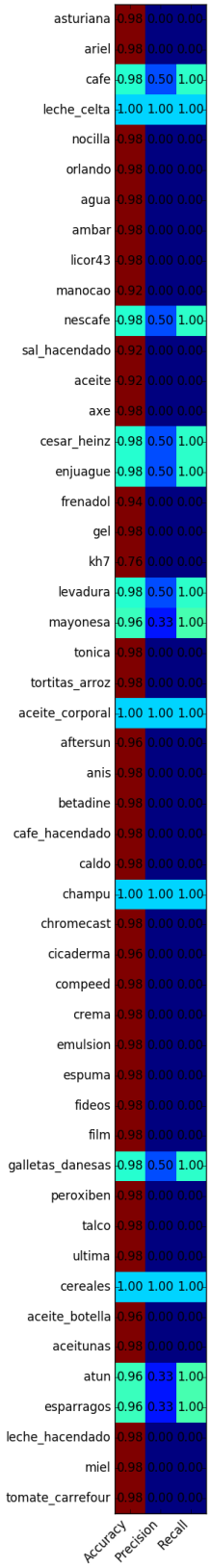


Figura 131: Tabla de precisión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal.

- **Capa fc7. Imágenes originales reescaladas.**

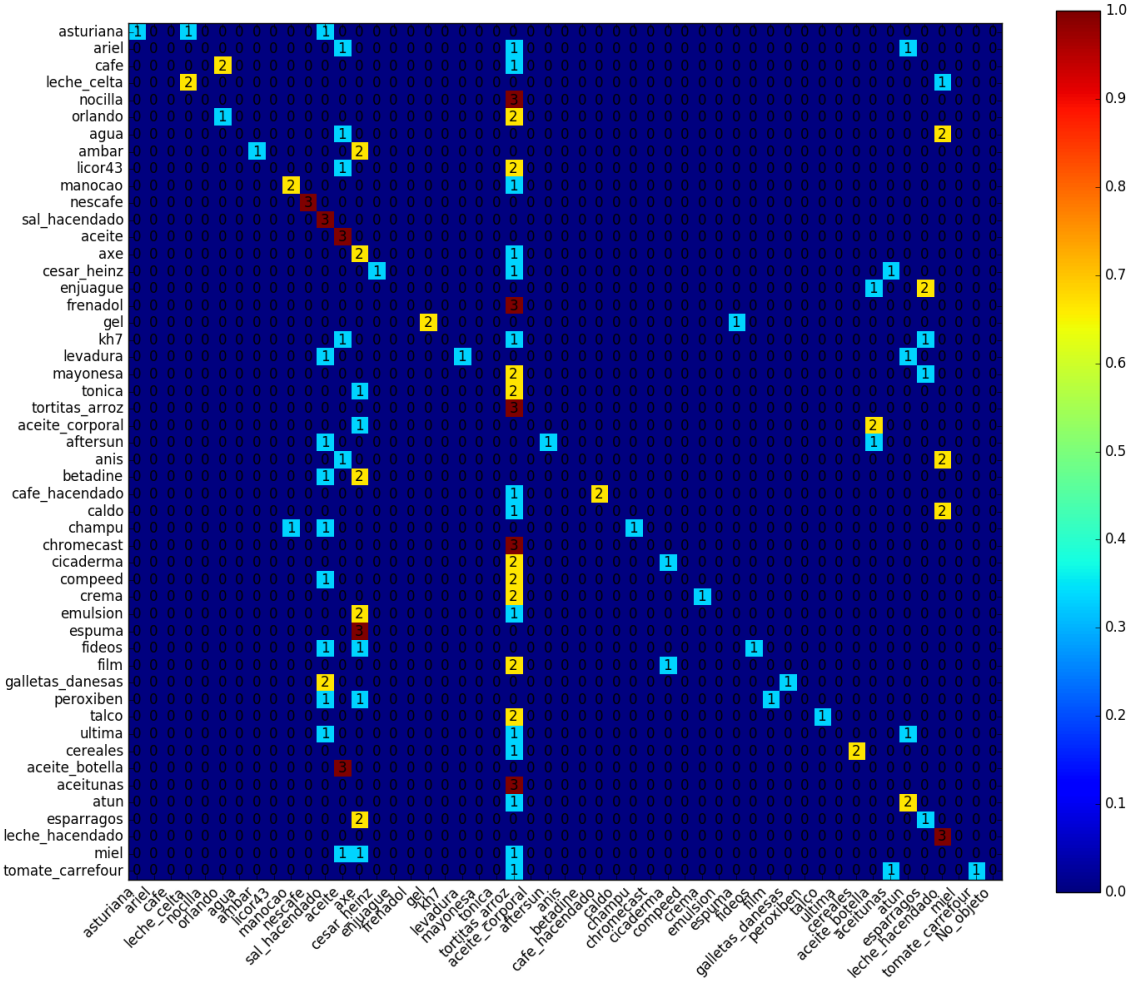


Figura 132: Matriz de confusión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 133: Tabla de "True positives" de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

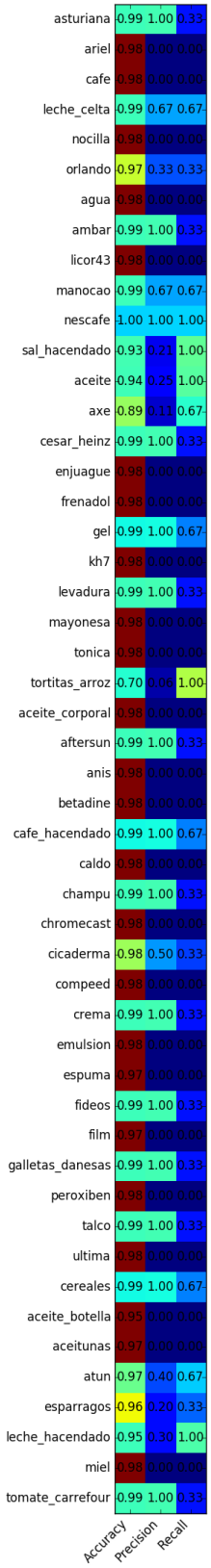


Figura 134: Tabla de precisión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

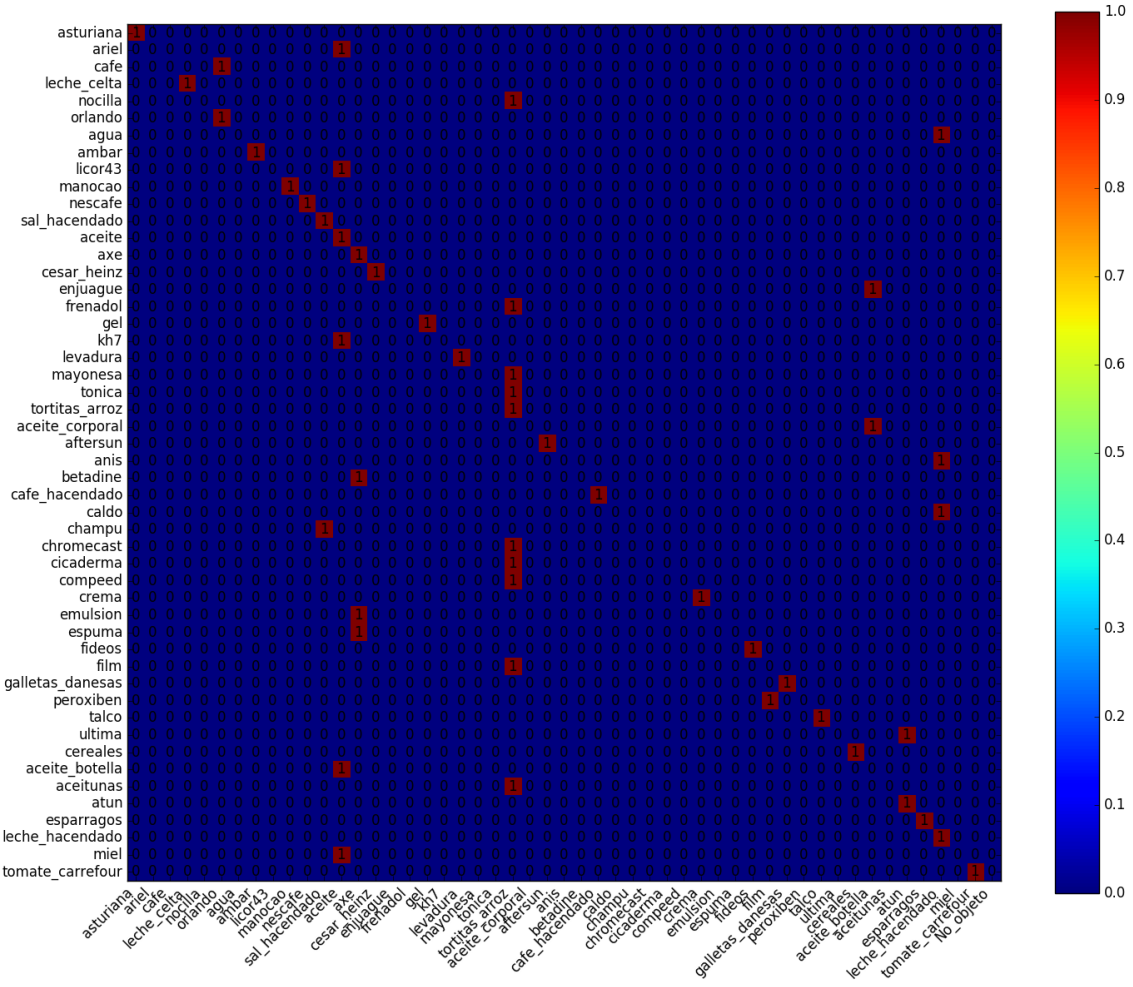


Figura 135: Matriz de confusión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 136: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

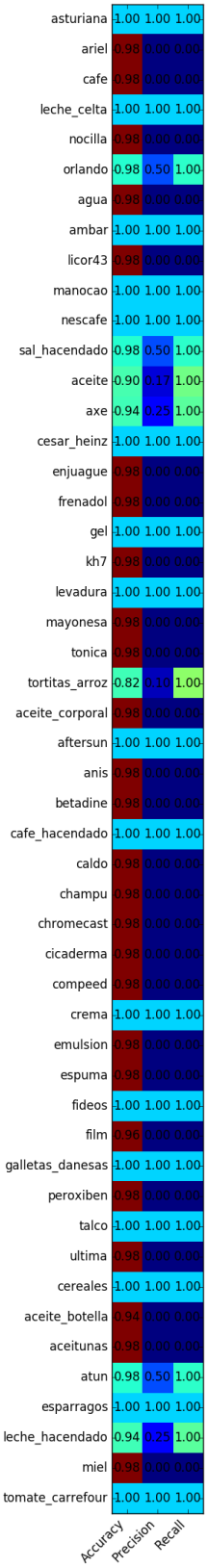


Figura 137: Tabla de precisión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

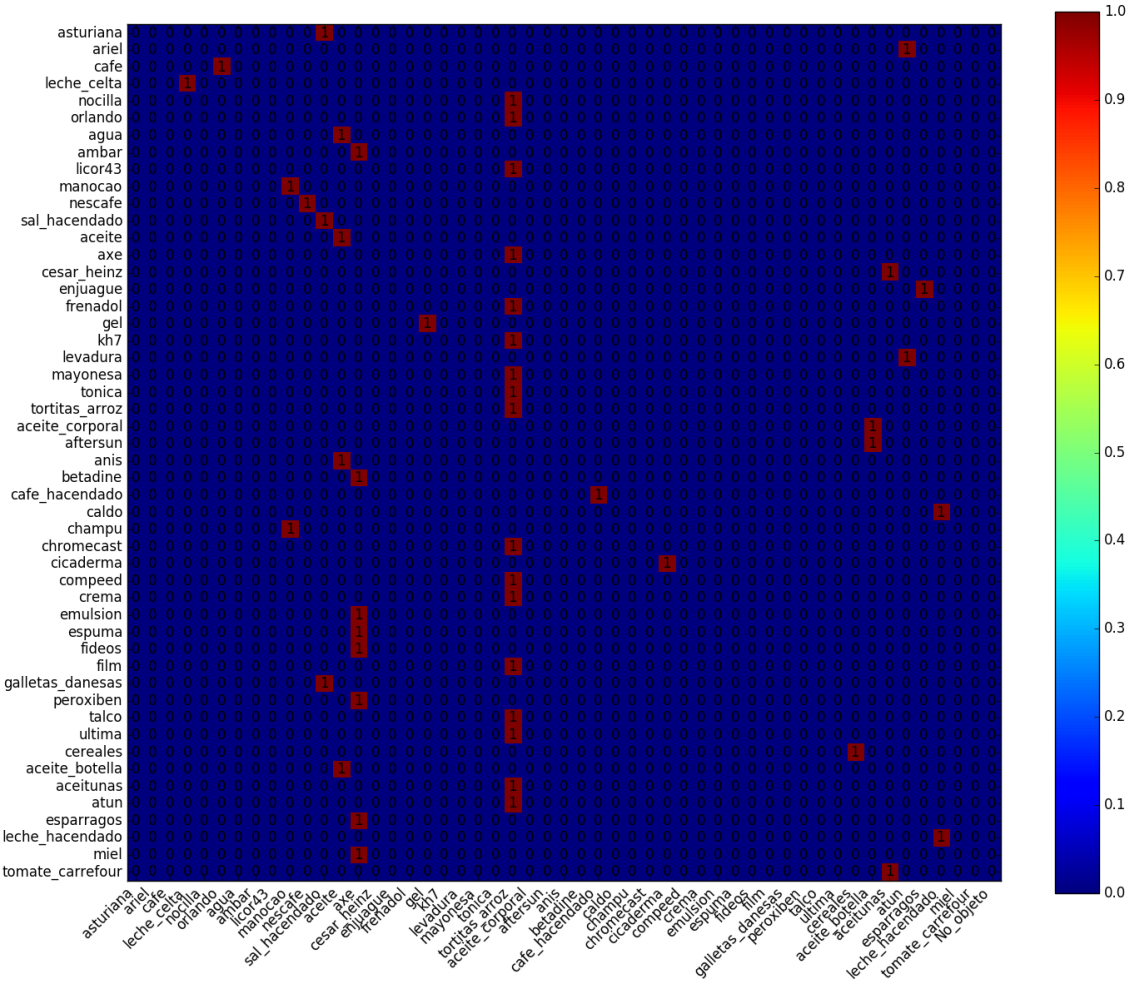


Figura 138: Matriz de confusión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 139: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

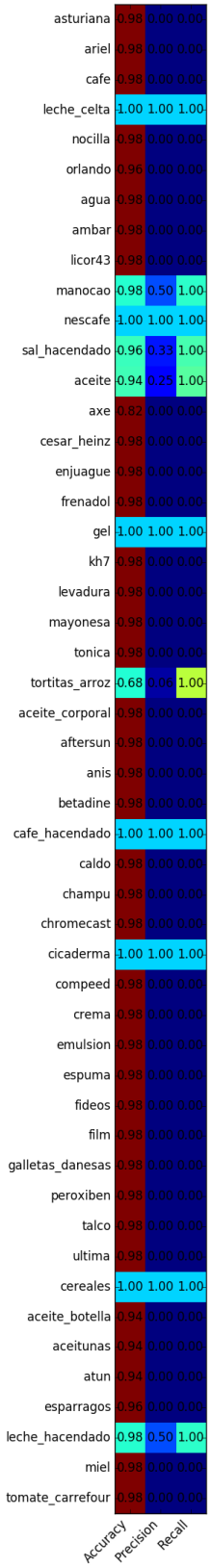


Figura 140: Tabla de precisión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

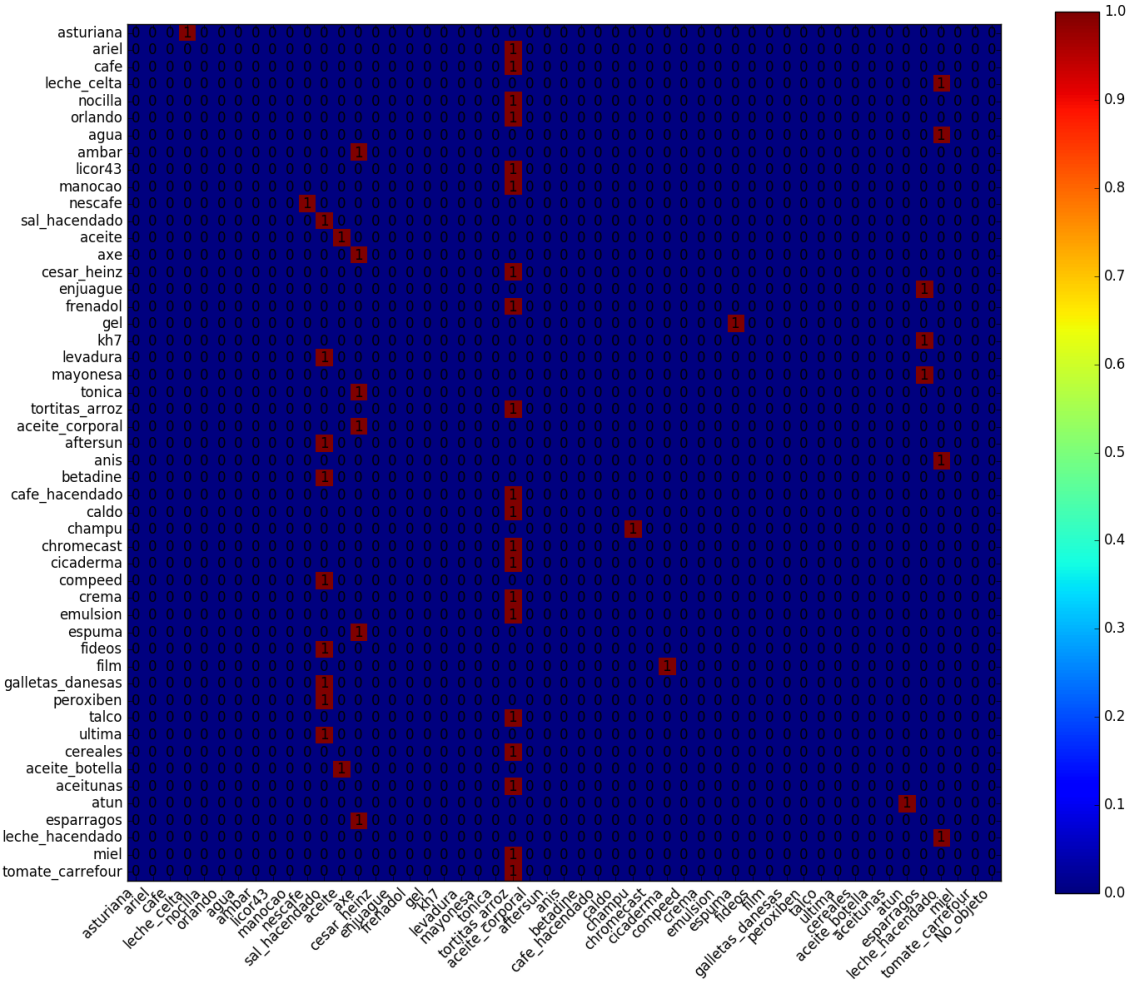


Figura 141: Matriz de confusión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 142: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

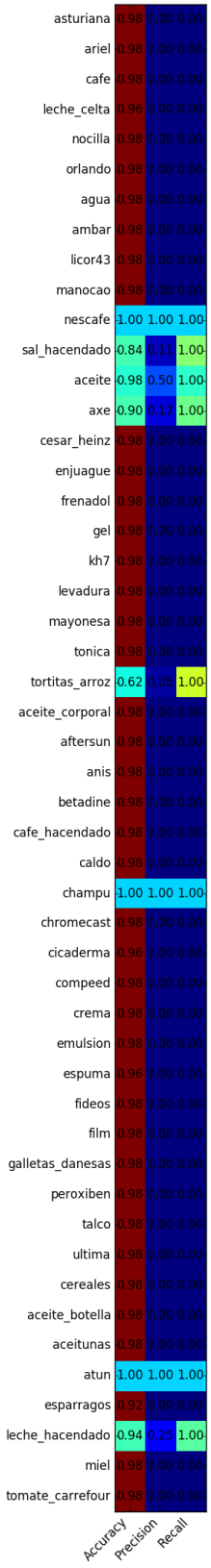


Figura 143: Tabla de precisión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc7 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

- **Capa fc8. Imágenes originales reescaladas.**

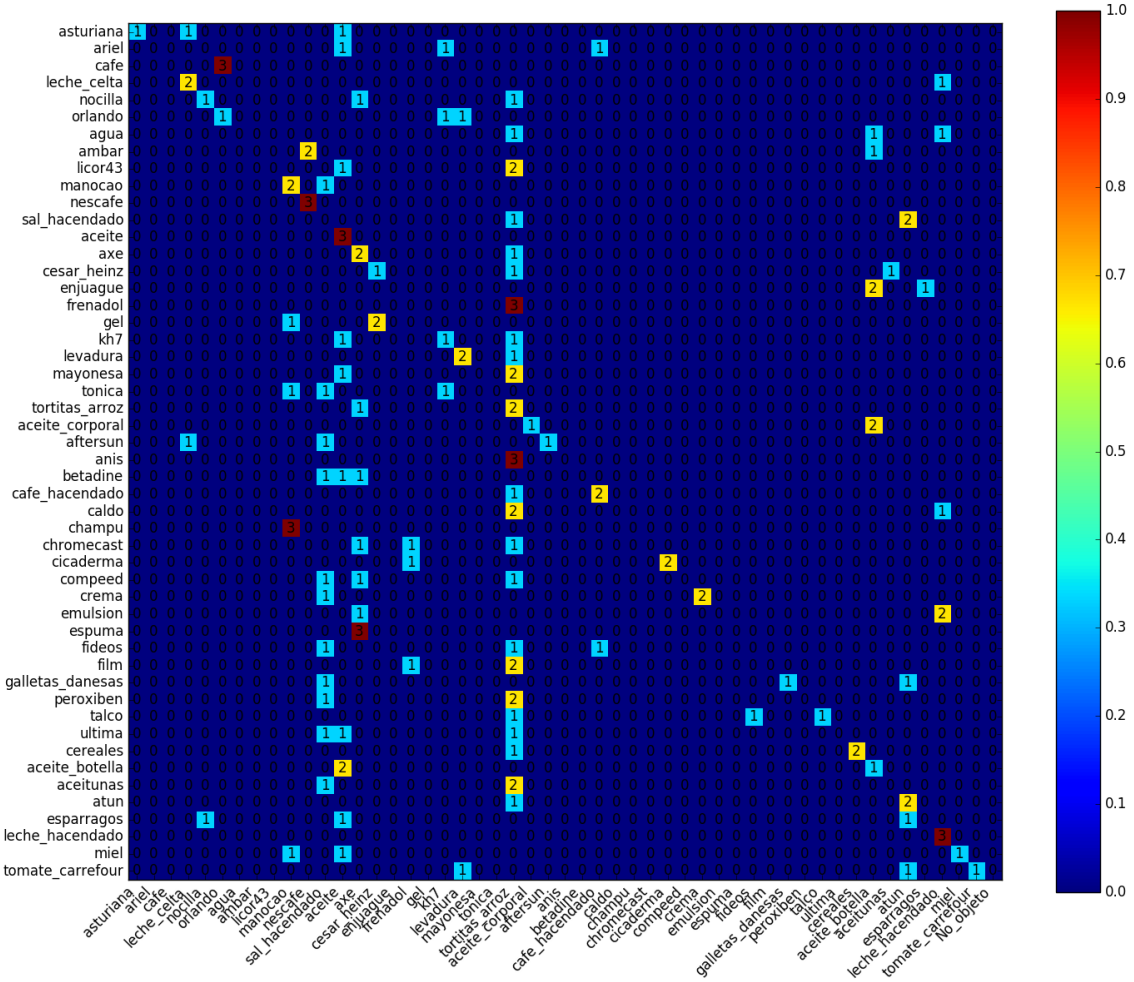


Figura 144: Matriz de confusión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 145: Tabla de "True positives" de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

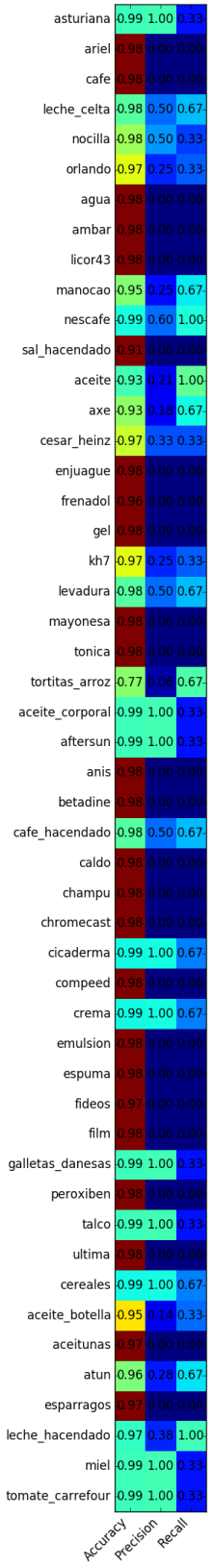


Figura 146: Tabla de precisión de los resultados obtenidos a partir de todos los objetos de test usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

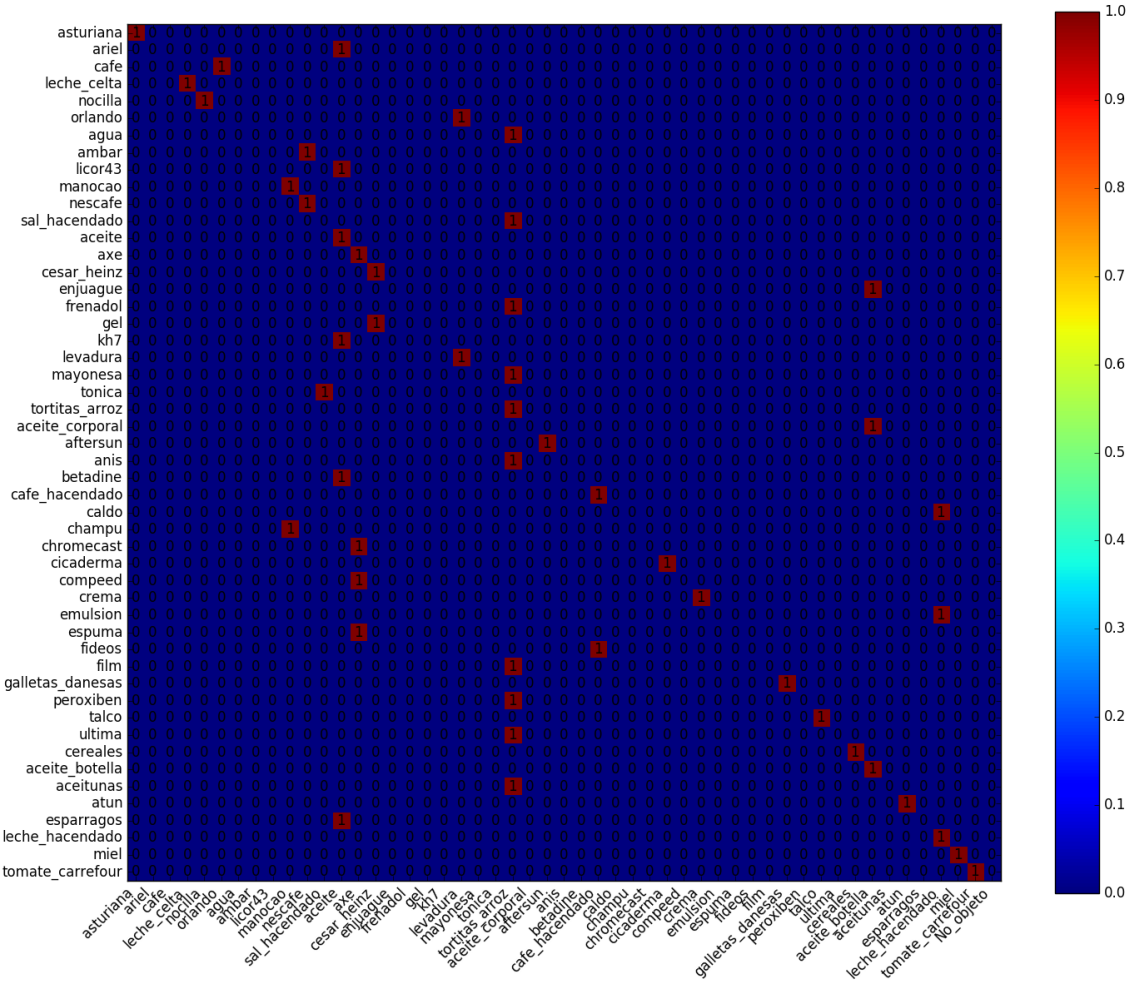


Figura 147: Matriz de confusión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

Figura 148: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

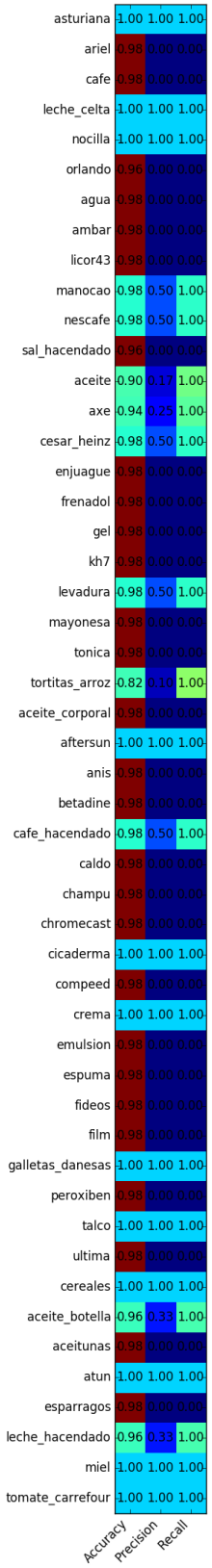


Figura 149: Tabla de precisión de los resultados obtenidos a partir de los objetos de test agarrados con la mano usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

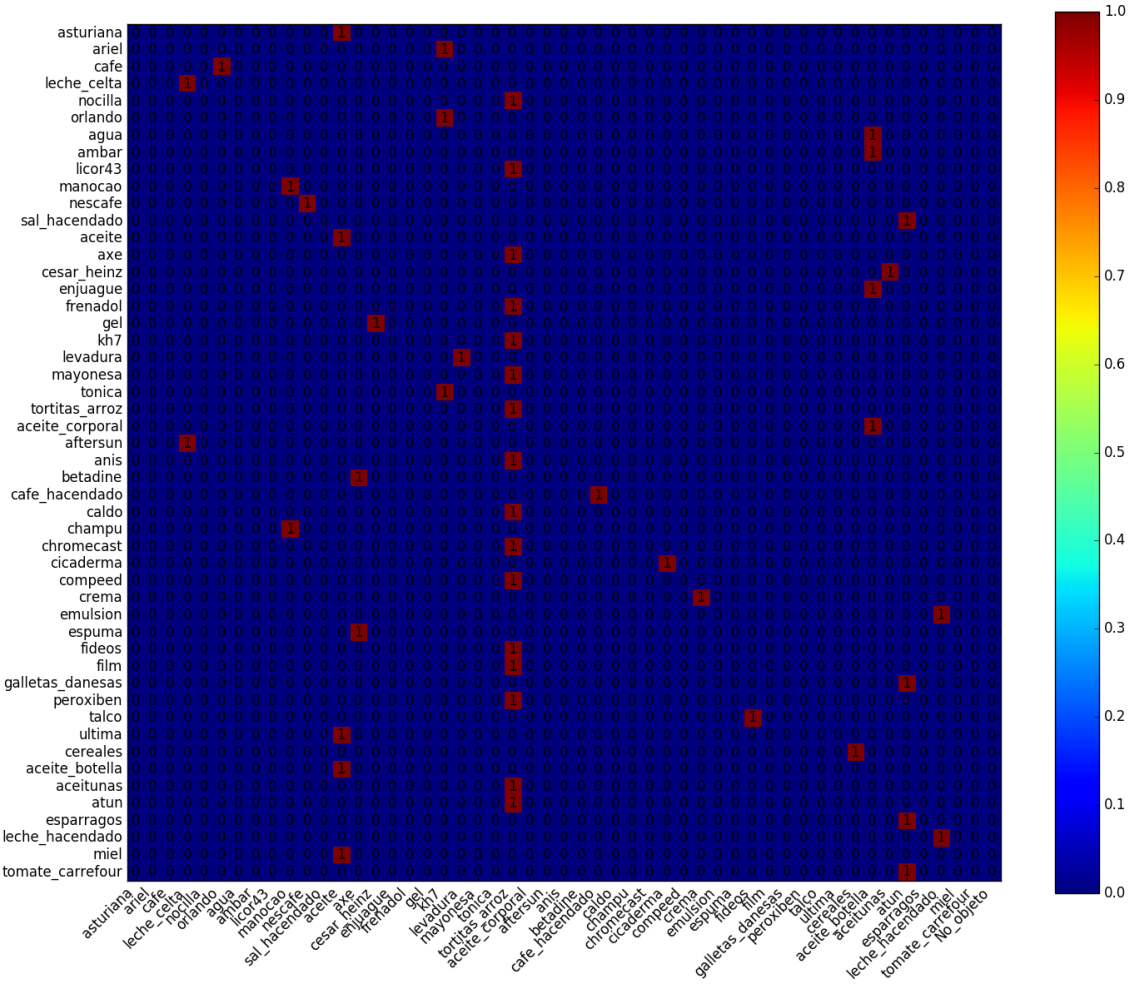


Figura 150: Matriz de confusión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal.

Figura 151: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

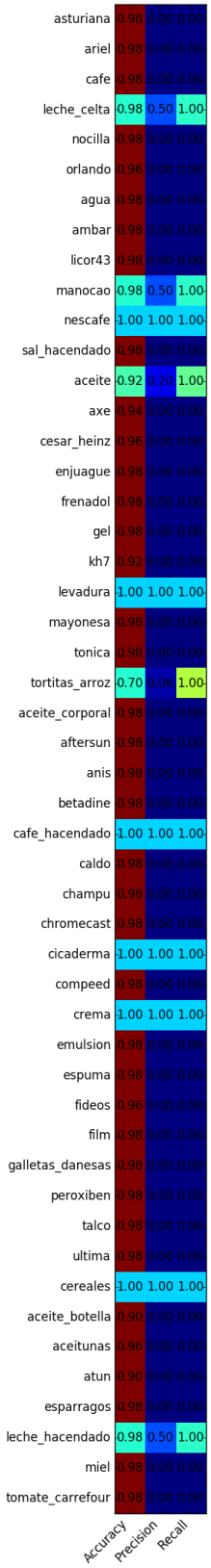


Figura 152: Tabla de precisión de los resultados obtenidos a partir de los objetos de test con oclusión usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

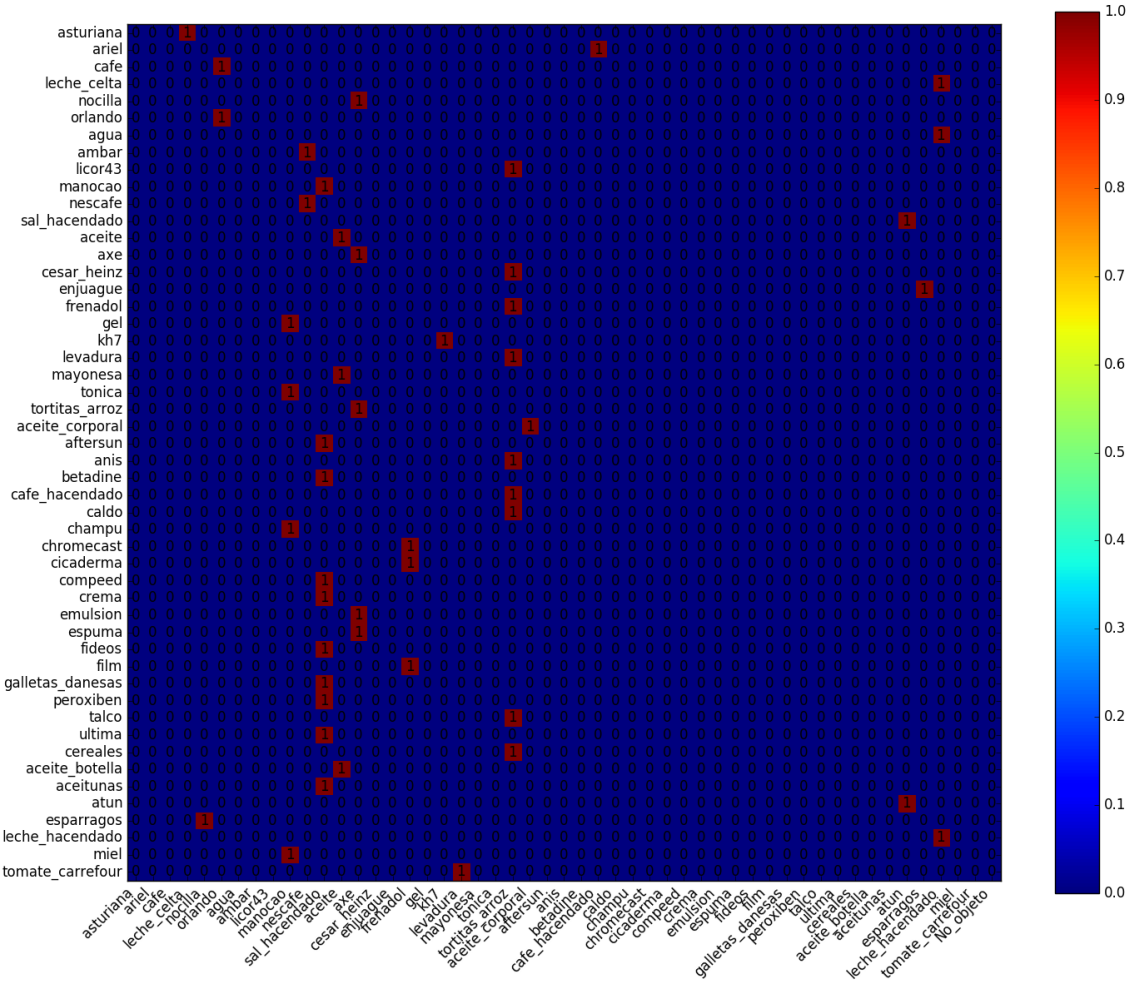


Figura 153: Matriz de confusión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal.

Figura 154: Tabla de "True positives" de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

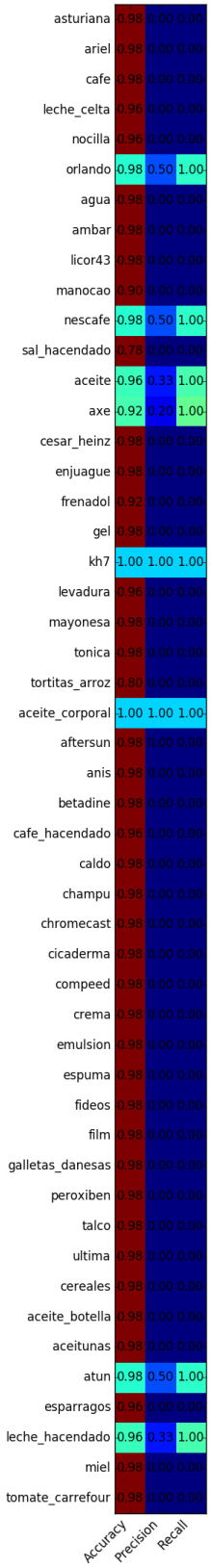


Figura 155: Tabla de precisión de los resultados obtenidos a partir de los objetos de test en la mesa usando como descriptor la capa fc8 de la red neuronal. Tanto las imágenes de test como las de la base de datos han sido reescaladas.

- **Capa fc7. 5 Imágenes reescaladas y sin ruido**

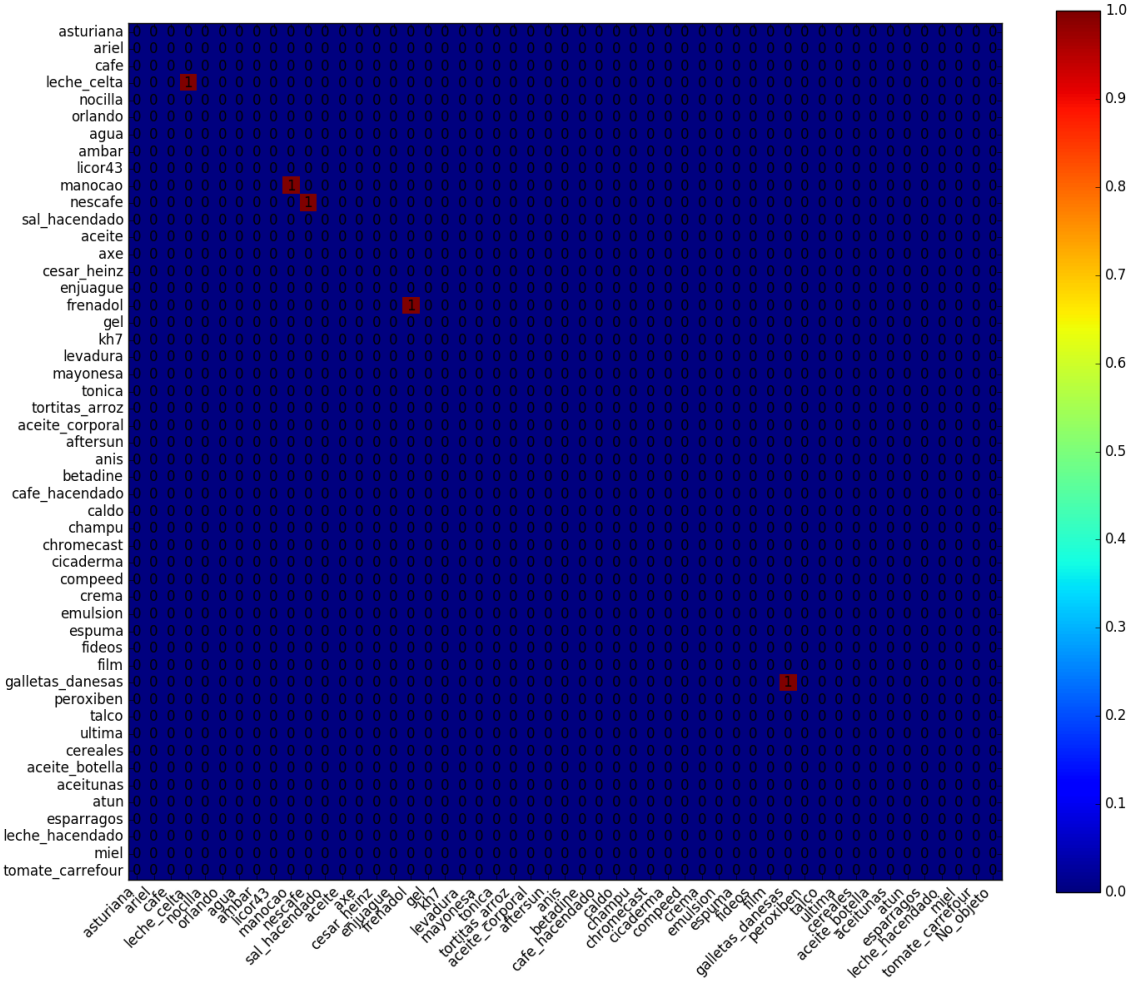


Figura 156: Matriz de confusión de los resultados obtenidos a partir de 5 objetos de test cuyas imágenes se han tomado sin ruido y han sido reescaladas.

Figura 157: Tabla de "True positives" de los resultados obtenidos a partir de 5 objetos de test cuyas imágenes se han tomado sin ruido y han sido reescaladas.

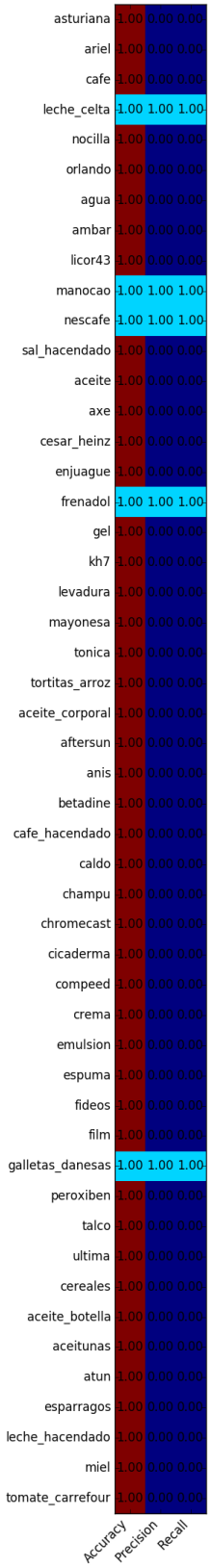


Figura 158: Tabla de precisión de los resultados obtenidos a partir de 5 objetos de test cuyas imágenes se han tomado sin ruido y han sido reescaladas.

F.4. Otras pruebas

- Las diferencias de tiempo y tamaño de imagen entre los distintos niveles de calidad en la compresión *jpeg* se pueden ver en la tabla 15.

Nivel de calidad de la compresión	**Tiempo (segs)**	**Tamaño (kB)**
100 %	99.91	≈300
90 %	79.51	≈175.6
80 %	66.76	≈120

Tabla 15: Diferencias de tiempo y tamaño de imagen entre los distintos niveles de calidad *jpeg* comprobados para el modo cliente-servidor (cliente: Android, servidor: portátil, red: 100mbps, base de datos: 50 objetos.

G. Referencias

[1] P. Föckler, T. Zeidler, B. Brombach, E. Bruns, and O. Bimber. PhoneGuide: Museum Guidance Supported by On-Device Object Recognition on Mobile Phones (2005). *PhoneGuide: Museum Guidance Supported by On-Device Object Recognition on Mobile Phones* . http://dl.acm.org/citation.cfm?id=1149490

[2] S. Gammeter, A. Gassmann, L. Bossard. Server-side object recognition and client-side object tracking for mobile augmented reality (2010). *Server-side object recognition and client-side object tracking for mobile augmented reality.* http://ieeexplore.ieee.org/document/5543248/?section=abstract

[3] M. Merler, C. Galleguillos, S. Belongie. Recognizing Groceries in situ Using in vitro Training Data (2007). *Recognizing Groceries in situ Using in vitro Training Data.* http://vision.ucsd.edu/sites/default/files/grozi_slam.pdf

[4] S. Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor. Caffe: Convolutional Architecture for Fast Feature Embedding (2014). *Caffe.* http://ucb-icsi-vision-group.github.io/caffe-paper/caffe.pdf

[5] Tuytelaars, T., Mikolajczyk, K. Local Invariant Feature Detectors: A Survey. 177-280 (2007). *Local Invariant Feature Detectors: A Survey.* http://www.eng.auburn.edu/~roppeth/courses/7970%202015A%20AdvMobRob%20sp15/literature/%5B2008%5D%20Local%20Invariant%20Feature%20Detectors-%20A%20Survey.pdf

[6] D.J. Fleet, A.D. Jepson. Feature Descriptors, Detection and Matching (2011) Pag 2. *Feature Descriptors, Detection and Matching.* https://www.cs.toronto.edu/~kyros/courses/2503/Handouts/features.pdf

[7] Ramisa, A., Tapus, A., Aldavert, D., Toledo, R. Robust Vision-based Robot Localization using Combinations of Local Feature Region Detectors. *Robust Vision-based Robot Localization using Combinations of Local Feature Region Detectors.* http://www.iiia.csic.es/~mantaras/comb_loc_Feb2009.pdf

[8] Lowe, D. G. Distinctive Image Features from Scale-Invariant Keypoints. University of British Columbia (2004) *Distinctive Image Features from Scale-Invariant Keypoints.* http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

[9] Bay, H., Tuytelaars, T. and Van Gool, L. SURF: Speeded Up Robust Features (2006) *Surf.* http://www.vision.ee.ethz.ch/~surf/eccv06.pdf

[10] E. Rosten, T. Drummond. Machine learning for high-speed corner detection (2006) *FAST.* https://www.edwardrosten.com/work/rosten_2006_machine.pdf

[11] E. Rublee, V. Rabaud, K. Konolige, G. Bradski. ORB: an efficient alternative to SIFT or SURF (2011) *ORB*. http://www.vision.cs.chubu.ac.jp/CV-R/pdf/Rublee_iccv2011.pdf

[12] S. Leutenegger, M. Chli, R. Y. Siegwart. BRISK: Binary Robust Invariant Scalable Keypoints (2011) *Brisk*. https://e-collection.library.ethz.ch/eserv.php?pid=eth:7684&dsID=eth-7684-01.pdf

[13] M. Muja, D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. (2009) *Flann*. http://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_visapp09.pdf

[14] D. Nister, H. Stewenius. Scalable Recognition with a Vocabulary Tree. (2006) *Bag of words*. http://ieeexplore.ieee.org/document/1641018/

[15] K. Grauman, T. Darrell. The pyramid match kernel: discriminative classification with sets of image features. (2005) *Spatial Pyramid*. http://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_visapp09.pdf

[16] R. Hartley, A. Zisserman. Multiple View Geometry in Computer Vision. (2004) *Multiple View Geometry in Computer Vision*. http://www.robots.ox.ac.uk/~vgg/hzbook/

[17] E. Vincent and R. Laganiere. Detecting Planar Homographies in an Image Pair. *RANSAC*. https://www.researchgate.net/profile/Robert_Laganiere/publication/3905826_Detecting_planar_homographies_in_an_image_pair/links/0c96052c70b8baccb6000000.pdf

[18] Definición de Deep Learning. Li Deng, Dong Yu. Deep Learning: Methods and Applications (Pags. 199-200). *Deep Learning*. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/DeepLearning-NowPublishing-Vol7-SIG-039.pdf

[19] Modelos entrenados para el framework Caffe. *Modelos entrenados Caffe*. http://caffe.berkeleyvision.org/model_zoo.html

[20] Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks (2012). *AlexNet*. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[21] O. Miksik and K. Mikolajczyk. Evaluation of local detectors and descriptors for fast feature matching. (2012) *Evaluation of local detectors and descriptors for fast feature matching*. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6460718

[22] E. Mair, G. D. Hager, D. Burschka, M. Suppa, and G. Hirzinger. Adaptive and Generic Corner Detection Based on the Accelerated Segment Test. (2010) *Evaluation of local detectors and descriptors for fast feature matching*. http://www6.in.tum.de/Main/Publications/Mair2010c.pdf

Las Web han sido accedidas por última vez el 25 de noviembre de 2016.

[23] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, P. Fua. BRIEF: Computing a Local Binary Descriptor Very Fast. (2012) *Brief*. http://ieeexplore.ieee.org/document/6081878/

[24] C. Harris and M. Stephens. A combined corner and edge detector. (1988) *Harris*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.434.4816&rep=rep1&type=pdf