Extracción automática de caras en imágenes captadas con móviles Android

Autor: Miguel Delgado Rodríguez

Tutor: JUAN CLIMENT VILARÓ

Cuatrimestre primavera 2012

Tabla de contenido

1. Introducción 4
2. Objetivos 6
2.1. Estudiar el algoritmo de Viola-Jones y la librería OpenCV 6
2.2. Clasificador por defecto 6
2.3. Entrenar un clasificador propio 6
2.4. Experimentos 7
2.5. Aplicación Android 7
2.6. Aplicación Final 8
3. La idea 9
4. Conceptos previos 10
4.1. Comprensión del algoritmo de Viola-Jones 10
4.1.1. Viola-Jones 10
4.1.2. Haar-like features 10
4.1.3. AdaBoost 13
5. La implementación 16
5.1. Pruebas y Entrenamiento guiado 16
5.1.1. Metodología 16
5.1.2. Entorno de trabajo 17
5.1.3. Estudio del clasificador TEST 18
5.1.4. Desarrollo de un aprendizaje guiado 22
5.1.5. Obtención de Imágenes 24
5.1.6. Herramientas de procesado 25
5.1.7. ObjectMarker 26
5.1.8. Ficheros info.txt 28
5.1.9. Haartraining 30
5.1.10. 2ª iteración (Bootstrapping) 31
5.2. Detección de caras en Android 37
5.2.1. Introducción 37
5.2.2. Metodología 38
5.2.3. Entorno de trabajo 40
5.2.4. Uso de la cámara 42
5.2.5. Integración con la librería OpenCV 43

5.2.6. Diseño 44
6. Resultados 47
6.1. TEST 47
6.2. Pruebas Clasificador propio 51
6.2.1. 1ª iteración 51
6.3. Pruebas Android 63
7. Conclusiones 65
7.1. Programación temporal 65
7.1.1. Planificación inicial general 67
7.1.2. Planificación final general 69
7.2. Presupuesto 71
7.2.1. Recursos Humanos 71
7.2.2. Licencias y recursos de Software 71
7.2.3. Amortización de hardware 72
7.2.4. Total 72
7.3. Objetivos Conseguidos 73
7.3.1. Detalle 73
7.4. Trabajos futuros 79
7.4.1. Fase PC 79
7.4.2. Mejoras Android 80
7.4.3. Futuras Aplicaciones 81
8. Referencias 83
ANEXO 1: Preparación entorno PC 85
Crear Proyecto Cmake 85
Cmake 85
Procedimiento de Creación de Proyecto con Cmake y OpenCV 86
Creación estructura (carpetas) 86
Definición del proyecto en CmakeLists.txt 86
Creación de CmakeLists.txt para compilar ejecutables 87
Compilación 88
IDE (Integrated Development Environment) 89
ANEXO 2: Preparación entorno Android 92
Instalación 92
Java 92
Eclipse IDE 92

SDK de Android 93
Plugin de Android para Eclipse 94
Integración con OPENCV 97
ANEXO 3: Planificación Temporal Detallada 99

1. Introducción

En los últimos años, el aumento de la velocidad de computación y sobretodo la expansión masiva de teléfonos inteligentes, que permiten tener en la palma de la mano la potencia de un ordenador, han provocado integración de sistemas de visión por computador en la vida cotidiana.

Este proyecto nace con la idea de profundizar un poco más en alguna técnica de visión que se encuentre en gran investigación en todo el mundo y aplicarlo en un dispositivo móvil, en este caso dentro de una aplicación del sistema operativo Android.

Una de las razones por las que he querido realizar este proyecto es en primer lugar porque un proyecto grande relacionado con la visión por computador me permitía comprobar cómo se encontraba en realidad el estudio de este campo, y adentrarme más profesionalmente. La adaptación al sistema operativo Android nos permite unir dos tecnologías con un gran potencial comercial.

Para realizar mi proyecto necesitaba saber que la información podía ser accesible. La obtención de información sobre el desarrollo en Android no es un problema, ya que el sistema operativo de Google cuenta con una enorme comunidad trabajando en el desarrollo de aplicaciones. En cuanto a la información sobre Visión por computador nos basaremos en el paper del Algoritmo de Viola Jones, y el libro sobre la librería OpenCV: Learning OpenCV.

Durante el proyecto se usarán diferentes metodologías, pero se intentará enfocar desde un diseño practico, donde la implementación y la interpretación de los resultados obtenidos en las pruebas indiquen cual debe ser el siguiente paso. No se desarrollará un diseño formal de las aplicaciones, aunque si se expliquen las tecnologías que se vean involucradas, y como serán usadas para conseguir nuestro objetivo.

En cuanto a la valoración de los resultados, realizaremos pruebas con imágenes donde contaremos cuantas caras se han detectado correctamente, cuantos falsos positivos, es decir, cuantos objetos no deseados son detectados y cuantas caras no han sido detectadas. De esta manera podremos establecer una comparación entre los diferentes clasificadores y su nivel de eficacia.

Finalmente se aplicarán las conclusiones obtenidas para poder elaborar la aplicación Android con las mayores posibilidades de éxito.

Por último, en lo referente a la memoria, hay que indicar que utiliza una estructura secuencial. Es decir, las diferentes fases del proyecto se irán explicando según han ido avanzando.

Cada una de las fases principales consta de una sección de resultados, que nos permitirá valorar el éxito alcanzado en esa fase y como ésta influye en las siguientes. Sobretodo debiéramos tomar alguna decisión que hiciera variar las hipótesis iniciales del proyecto.

Al final se incluirán unas conclusiones finales para poder hacer balance de todo el proyecto y definir si los objetivos se han cumplido y cuáles podrían ser algunas características que sería interesante que nuestro software soportara pero que quedan fuera del objetivo del proyecto.

2. Objetivos

Durante el desarrollo del proyecto nos fijaremos algunos objetivos que desarrollaremos en diferentes etapas. Cada una de estas etapas será dependiente del resultado de la anterior para finalmente obtener un resultado que englobe todo lo investigado.

El objetivo principal será obtener una aplicación Android capaz de detectar caras dentro de un entorno no preparado. Para ello necesitaremos haber realizado un clasificador guiado que nos permita obtener imágenes modificadas donde se enmarque la cara.

Los objetivos principales son los siguientes:

- Estudiar el algoritmo de Viola-Jones y la librería OpenCV.
- Crear un programa que aplique lo aprendido con un clasificador por defecto.
- Entrenar un clasificador propio.
- Realizar experimentos para comprobar la efectividad.
- Realizar una aplicación Android.

2.1. Estudiar el algoritmo de Viola-Jones y la librería OpenCV

En primer lugar nos centraremos en comprender como funciona el algoritmo de Viola-Jones, lo que supondrá parte de los conocimientos previos que necesitaremos para desarrollar el proyecto. Y también estudiaremos cómo funciona la librería OpenCV que nos permitirá utilizarlo en un entorno real.

2.2. Clasificador por defecto

En un segundo paso, crearemos un entorno donde aplicaremos lo aprendido anteriormente. Creamos un pequeño programa que sea capaz de clasificar caras utilizando un clasificador de los que Opencv trae por defecto.

2.3. Entrenar un clasificador propio

Después realizaremos un entrenamiento guiado de manera que podemos comprender como se crea un clasificador, que experimentos deberemos realizar para comprobar su efectividad y como podemos mejorarlo.

Nuestro clasificador tendrá algunas restricciones que debemos dejar claras al inicio. Nos centraremos principalmente en caras frontales aunque intentaremos que una pequeña variación en la inclinación

de la cara no se convierta en un obstáculo para su identificación. Aunque si ignoraremos completamente las caras de perfil. La detección será válida hasta aproximadamente 30º desde la imagen totalmente frontal.

2.4. Experimentos

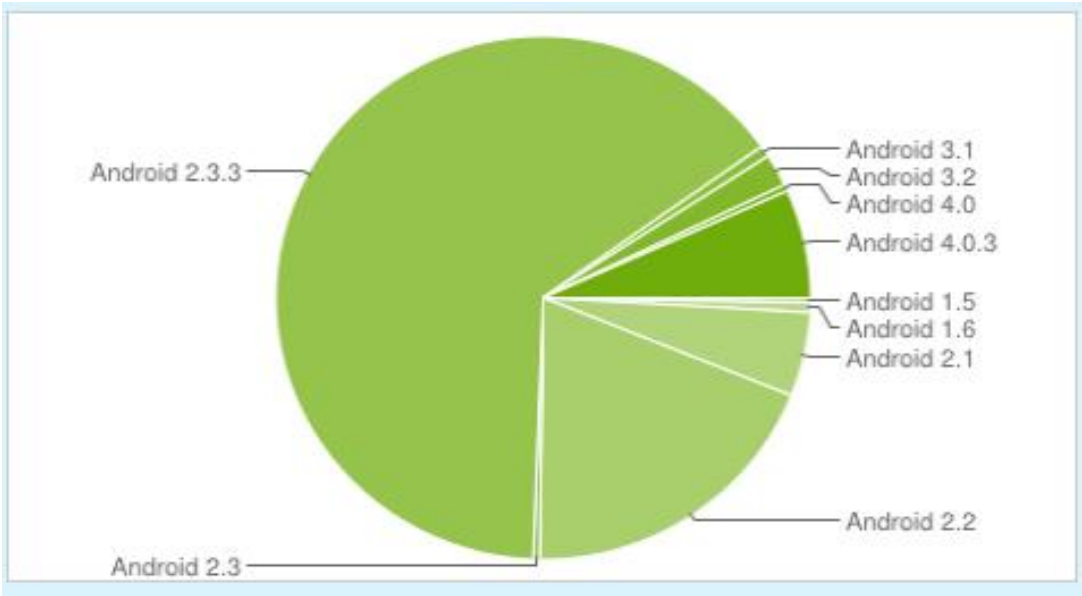
En una cuarta fase, mejoraremos nuestro clasificador en función de los resultados de la fase anterior y determinaremos algunas conclusiones que nos permitan valorar el trabajo realizado hasta ahora.

2.5. Aplicación Android

Por último, y podría definirse como el objetivo principal del proyecto, tenemos la creación de una aplicación Android. Esta aplicación detectará caras en un entorno real, mostrándolas en un dispositivo móvil como puede ser un smartphone y aplicará algún tipo de filtrado sobre la subimagen detectada.

En cuanto a la fase Android hemos de tener en cuenta algunas restricciones dependientes de la arquitectura. El framework[1] Android tiene en este momento 15 versiones de su SDK[2] lo que nos lleva a intentar centrarnos en solo 2 de estas versiones.

Esto puede hacer que en algunos dispositivos nuestra aplicación no pueda ser instalada, pero la compatibilidad de nuestra aplicación con todos los dispositivos móviles es un requerimiento que queda fuera de los objetivos del proyecto. Aun así nos centraremos en las versiones 4.0, la más reciente, y la 2.3.3, la más extendida entre los dispositivos Android.



DISPERSIÓN DE VERSIONES ANDROID 1

[1] Un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática

[2] conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto

2.6. Aplicación Final

Finalmente, aplicaremos la tecnología y los procedimientos investigados para concretar todo en una aplicación práctica. Para ello como objetivo final aplicaremos a la aplicación Android un filtro de difuminado (*blurring*) sobre la subimagen detectada.

El uso principal de este tipo de tecnología puede ser usado, por ejemplo, en sistemas donde debamos tomar una foto, pero queremos preservar el derecho a la intimidad de las personas que salen en ella los cuales de manera automática serán difuminados.

EJEMPLO DETECCIÓN Y DIFUMINADO AUTOM.

3. La idea

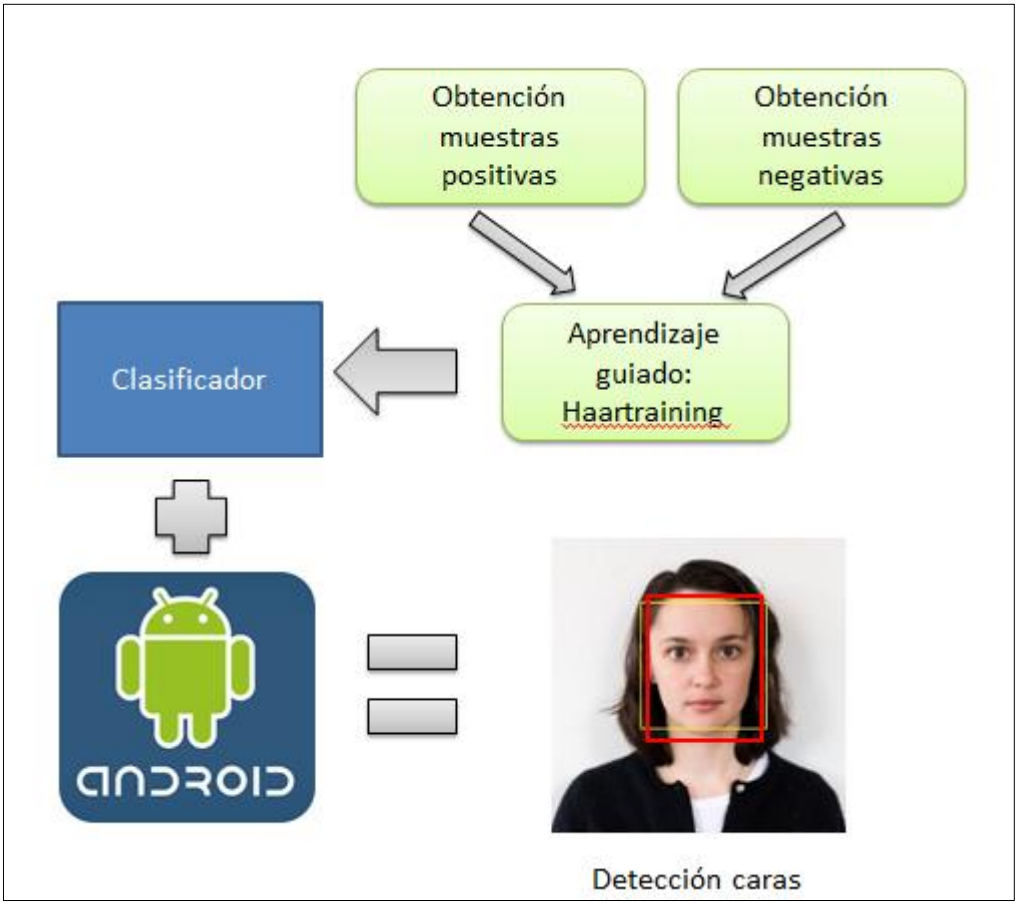
El principal problema es la clasificación de caras partiendo de una imagen donde se encuentran de manera aleatoria y sin ningún tipo de identificar como puede ser un color diferente, alguna marca especial, etc.

De manera que debemos identificar las imágenes por medio del reconocimiento de patrones. Partiendo de muestras positivas de lo que queremos detectar, una cara, obtenemos un árbol de decisión que nos dice si lo es o no.

Nuestro algoritmo obtendrá de cada imagen donde se encuentran las caras por medio del descarte, contra más efectivo es el clasificador mejor será la detección y más eficiente, porque será capaz de descartar mejor las subimagenes que no son caras.

Para ellos usaremos el algoritmo de Viola-Jones que describe como por medio de patrones que coge como muestras, es capaz de crear el árbol de decisión (AdaBoost) y determinar si una subimagen es una cara o no.

Finalmente aplicaremos este algoritmo a una plataforma Android lo que nos permitirá tener una aplicación móvil que desarrolla esta tecnología.



4. Conceptos previos

4.1. Comprensión del algoritmo de Viola-Jones

4.1.1. Viola-Jones

El algoritmo de Viola-Jones permite la detección robusto en tiempo real de caras. El algoritmo de Viola-Jones supone un gran avance dentro de los algoritmos de detección por su gran rapidez, y porque la clasificación se realiza mediante características en vez de píxel a píxel, lo que permite una cierta abstracción del algoritmo respecto el resultado.

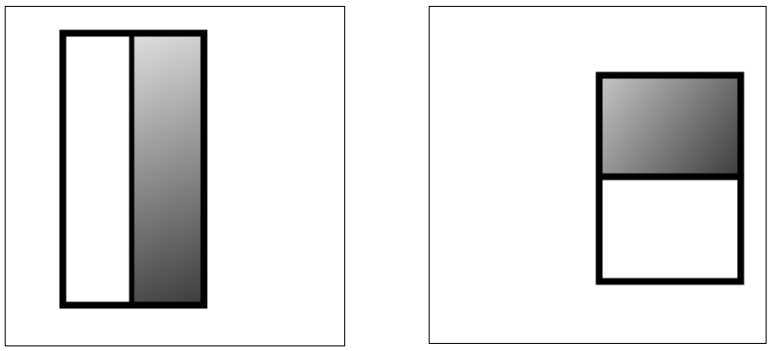
4.1.2. Haar-like features

La principal razón para usar características en el algoritmo es que permite hacer una asociación entre conocimiento aprendido y el análisis de los datos adquiridos. Además, la clasificación por características es mucho más rápida que el procesado por análisis basados en píxel.

Las características usadas son las mismas que fueron usadas por Papageorgiou et al. (1998)[3]. Las características de Haar (*Haar-like features* en inglés) permiten obtener información de una zona concreta mediante una operación aritmética simple: Esto nos lleva a una gran eficiencia tanto de cálculo como espacial.

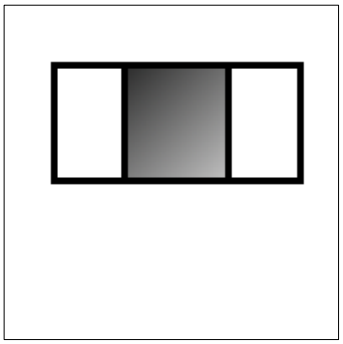
En concreto se usan tres características de Haar:

- Característica de dos rectángulos (*two-rectangle feature*)
- Característica de tres rectángulos *(three-rectangle feature*)
- Característica de cuatro rectángulos (*four-rectangle feature*)

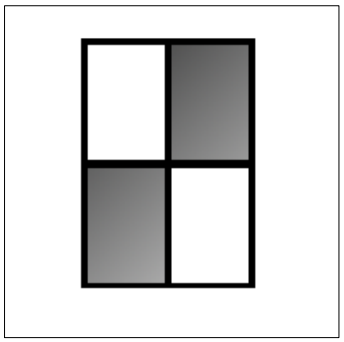


TWO-RECTANGLE FEATURE 1

[3] Paper de sobre clasificadores: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00710772



THREE-RECTANGLE FEATURE 1



FOUR-RECTANGLE FEATURE 1

En todos los casos anteriores la obtención del valor de la característica consiste en la resta entre el valor de una o más subzonas dentro de la zona analizada. Es decir, restamos el valor que damos una subzona, mediante la integral sumada (explicada más adelante en este mismo documento), con el de otra subzona.

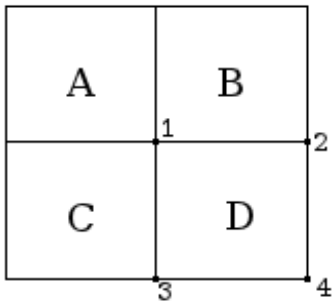
En las figuras antes mostradas se puede ver como la comparación consiste en la resta entre las zonas grises y las zonas blancas de la imagen.

4.1.2.1. Integral Image

Para computar rápidamente cada uno de los rectángulos se usa una representación de la imagen llamada "Integral de la imagen" (Integral Image). La integral de una imagen respecto un punto x,y consiste en la suma de los píxeles por arriba y a la izquierda de dicho puntos, x,y incluidos.

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y'),$$

La integral de la imagen representa en computación una manera elaborada de obtener valores de forma eficiente. Este método permite el uso de programación dinámica[4], que permite la obtención de valores dentro de la imagen a través de otros valores calculados previamente.



Ejemplo:

Para calcular el valor del rectángulo D podemos hacerlo mediante los cálculos ya realizados de las áreas A, B y C.

Sabiendo que el punto 4 se calcula con la suma de todos los puntos por encima y a la izquierda de este, podemos deducir que el valor de D es el valor de 4 menos los valores de las áreas A, B y C.

Por tanto, D= 4 – (1) - (2 -1) – (3 -1) = 4+1-2-3

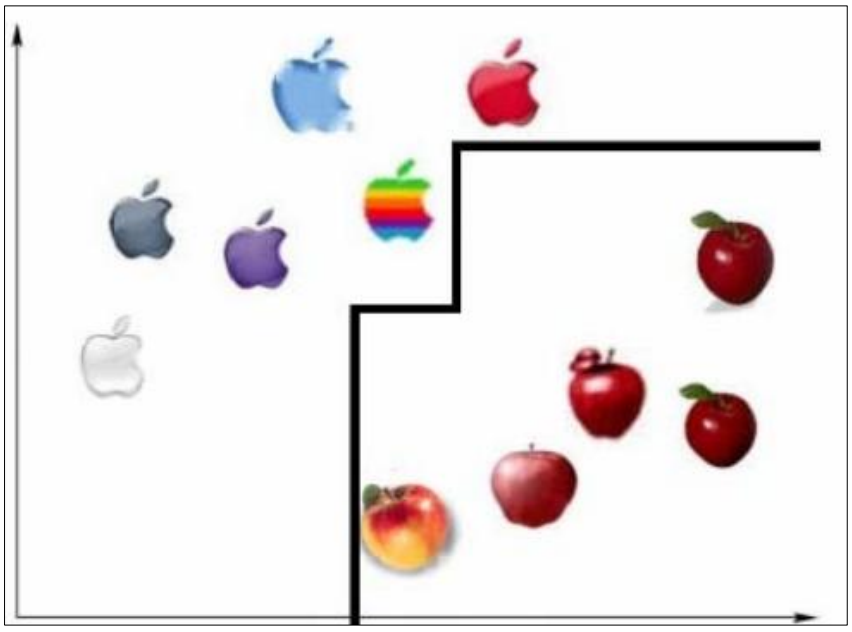
[4] http://es.wikipedia.org/wiki/Programaci%C3%B3n_din%C3%A1mica

De esto podemos obtener que para calcular el clasificador 2-rectángulo (dos áreas de un rectángulo) necesitas 6 puntos para evaluar una característica, el de 3 áreas de un rectángulo necesita 8 a búsquedas y si utilizamos clasificadores de 4 áreas diferentes en un área determinada necesitaremos 9 puntos.

4.1.3. AdaBoost

AdaBoost (Adaptative Boost) es un algoritmo de aprendizaje máquina que consiste en la clasificación de características por medio de un grupo de clasificadores. El algoritmo se basa en la mejor forma de aplicar esos clasificadores para detectar favorablemente algo.

En el algoritmo de Viola-Jones el algoritmo AdaBoost es capaz de elegir entre un gran conjunto de filtros, las características de Haar, para seleccionar en cada momento cuál de ellos se ajusta mejor para que se clasifique satisfactoriamente los diferentes elementos que queremos clasificar.



Ejemplo de intento de clasificación, separar manzanas reales de plásticas

4.1.3.1. Procedimiento

Para describir cómo funciona el algoritmo debemos centrarnos en la idea que intentamos discernir si la muestra que intentamos analizar es de un tipo o de otro.

Por cada muestra:

t = 1, . . . , T,

1. Entrenar un aprendizaje partiendo de una distribución de ejemplo.
2. Computamos los índices de error.
3. Computados los índices de pesos de cada muestra.
4. Actualizamos la distribución de las muestras inicial.

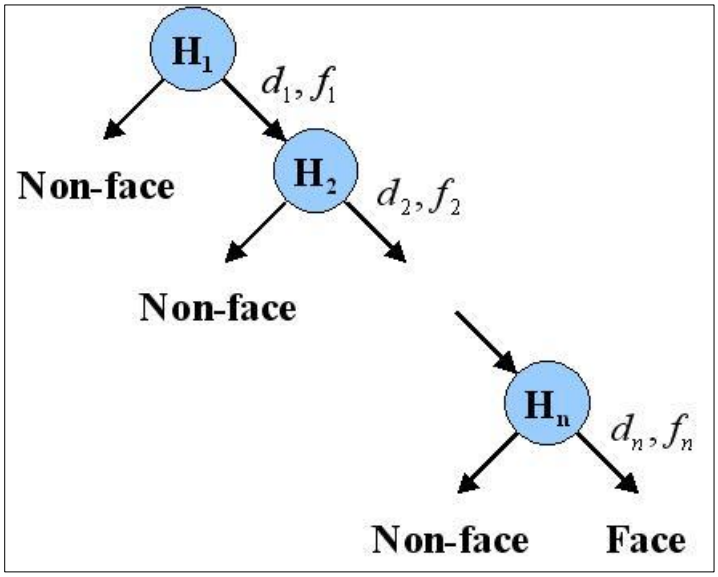
Con este procedimiento obtenemos que después de algunas iteraciones, donde hemos probados todos los clasificadores de Haar para cada una de las muestras, tenemos un clasificador que separa acordemente entre muestras positivas y muestras negativas.

En un primer paso, se genera una detección con algún clasificador. Una vez hecho esto los elementos mal clasificados aumentan su peso para que el siguiente clasificador usado de más importancia a que la clasificación de los elementos con mayor peso sea la correcta.

Una vez realizado este último paso con diferentes clasificadores, obtenemos un único clasificador como combinación de los anteriores y que clasifica todos los elementos correctamente con un valor de error aceptable.

4.1.3.2. Cascada de decisión

El algoritmo de Viola-Jones utiliza un árbol binario de clasificadores para decidir si una región se trata de una cara o no. Esta cascada está formada por nodos en los que se analizan las diferentes características de las diferentes regiones de la imagen. Cada nodo representa un clasificador obtenido con el algoritmo AdaBoost.



El proceso de aprendizaje que utiliza el algoritmo de Viola-Jones permite como valor de entrada un error que permitiremos en la clasificación en el primer nodo del árbol.

Utilizando ese valor lo que intenta el algoritmo es crear un clasificador que en el primer nodo permite el error máximo que hemos permitido en la detección y según el árbol se vuelve más profundo el cri-

terio con el que descartamos es mucho más estricto. De esta manera cuando una muestra llega al último nivel del árbol sabemos que ha pasado por todos los niveles de detección.

Además, esta forma de organizar el árbol de decisión nos permite descartar en un primer momento las muestras más sencillas de detectar y aplicar un mayor esfuerzo a aquellas cuya clasificación sea más dudosa.

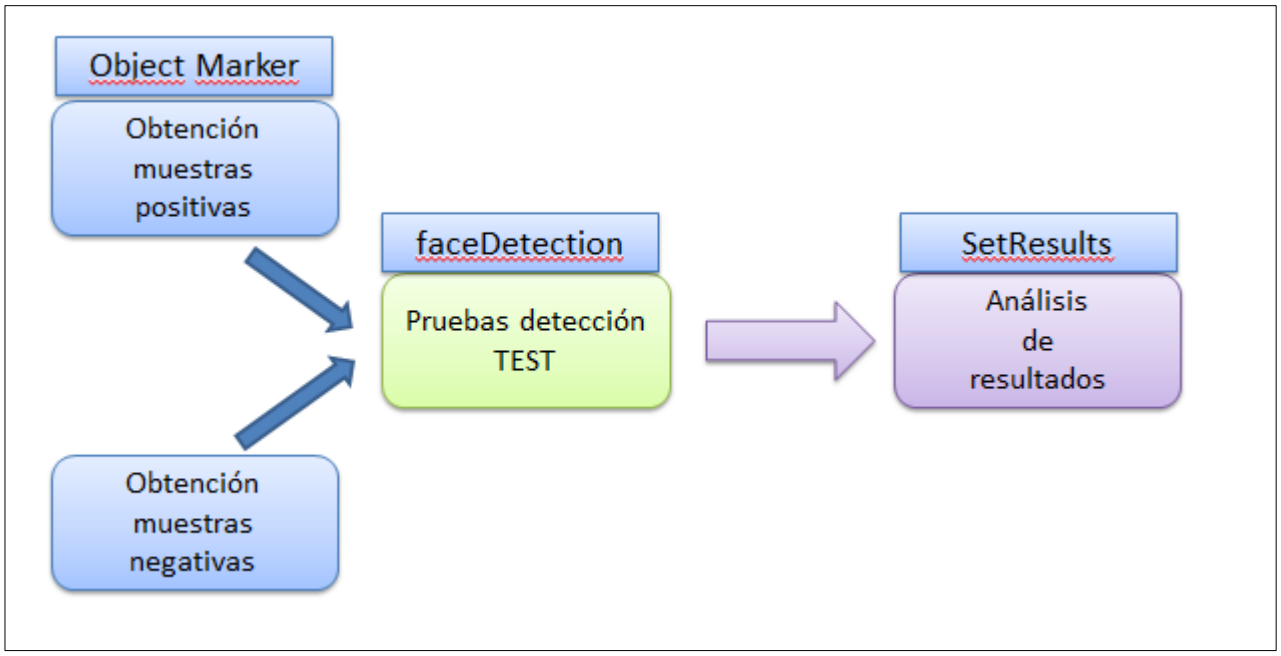
5. La implementación

En este apartado describiremos los diferentes procesos que nos han llevado a cumplir los objetivos que nos hemos planteado. En primer lugar definiremos 2 desarrollos totalmente diferenciados. En primer lugar, una implementación dentro de un entorno de PC donde desarrollaremos aplicaciones para ejecutar en cualquier ordenador y una segunda donde desarrollamos una aplicación para dispositivos Android.

5.1. Pruebas y Entrenamiento guiado

5.1.1. Metodología

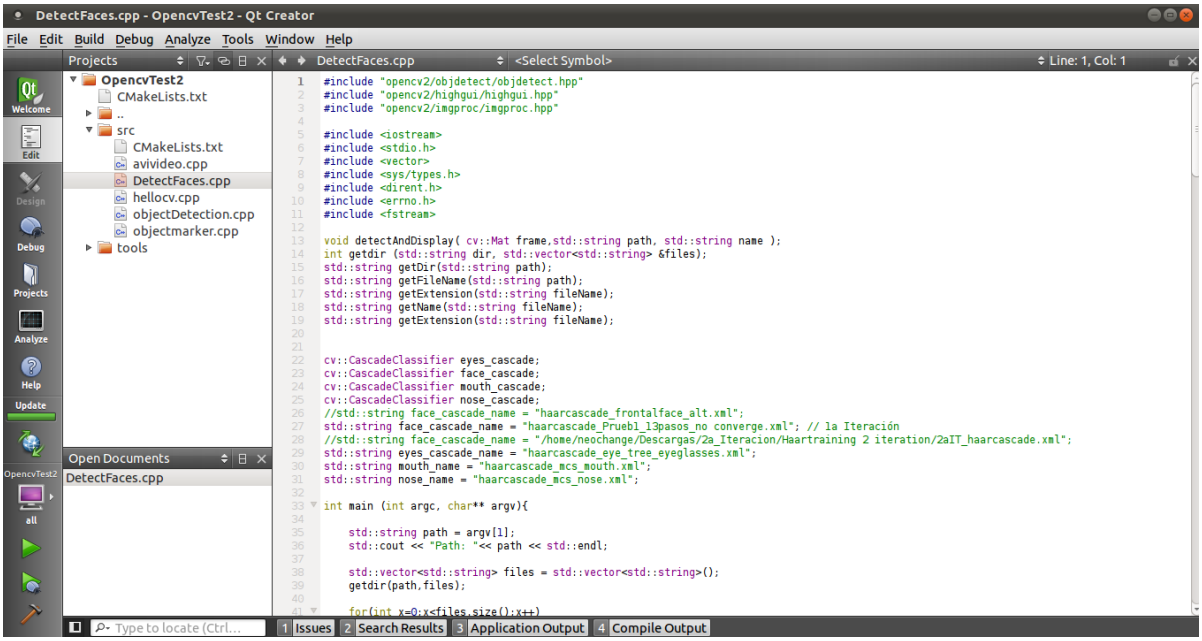
En lo referente a la metodología que usaremos tendremos dos líneas principales de actuación. En primer lugar, desarrollaremos software que nos permita realizar cada una de las fases del entrenamiento con comodidad, es decir, serán programas que necesitaremos en cada una de las fases de entrenamiento guiado.



METODOLOGIA HAARTRAINING 1

5.1.2. Entorno de trabajo

En cuanto al entorno de trabajo usaremos el IDE QtCreator para el desarrollo de las aplicaciones que usaremos durante el entrenamiento. Este IDE está optimizado para poder programar cómodamente aplicaciones en el lenguaje C++ y la librería gráfica Qt.



QTCREATOR 1

Las librerías Qt nos permitirían que las aplicaciones dispongan de una interfaz gráfica para la interacción con el usuario si fuera necesario.

Durante el desarrollo usaremos un programa de control de versiones para poder tener una seguridad mientras hacemos nuestras aplicaciones. Podremos tener cada una de las versiones de nuestro programa que consideremos que tienen un cambio significativo y podremos regresar a esa si lo necesitáramos. Además al no estar alojado en nuestro PC nos aseguramos una copia de seguridad del proceso.

5.1.3. Estudio del clasificador TEST

En su versión más actual, la librería OpenCV dispone de algunos recursos para la clasificación de caras en diferentes ambientes. Entre estos recursos se encuentra el clasificador haarcascade_frontalface_alt. Éste clasificador se ha conseguido por medio de un aprendizaje guiado como el que desarrollaremos más adelante.

Utilizaremos este clasificador como un grupo de control con el que compararemos el que generaremos nosotros mismos. Las hipótesis que sostendremos en esta comparación serán las de determinar que nuestro clasificador es capaz de detectar un mayor número de rostros, con una tasa de falsos positivos menor y que podremos tener una diversidad mayor de posiciones de la caras, es decir, podremos tener desde caras totalmente frontales hasta cercanas al perfil.

5.1.3.1. Programa TEST

Para probar la detección de caras usando la librería OpenCV haremos un programa de prueba que llamaremos faceDetect. Este programa recibirá como entrada una imagen y devolverá la misma imagen con las caras detectadas marcadas con recuadros.

También dentro de los objetivos del programa estará el de obtener un fichero que describa las caras obtenidas, su localización y su tamaño que usaremos más adelante cuando generemos nuestro propio clasificador.

EJEMPLO CLASIFICACIÓN 1

5.1.3.2. Tipos de datos y funciones de OpenCV utilizadas

cvLoadImage(pathFile);

cvLoadImage() nos permite cargar una imagen desde un fichero en un objeto de tipo IplImage también propio de Opencv. Después con ese objeto podemos hacer las modificaciones que necesitemos para nuestra aplicación.

cvSaveImage(cadena.c_str(),im);

cvSaveImage() permite guardar una imagen en un fichero. La ruta del fichero resultante se especifica con la cadena que precede al argumento de la imagen.

int cvNamedWindow(const char* name, int flags=CV_WINDOW_AUTOSIZE);

cvNamedWindow() crea una ventana de opencv. Esta ventana permite mostrar imágenes, pintar objetos o cualquier aspecto visual que se encuentre dentro de la librería opencv. Si una ventana con el mismo nombre ya existe, la función no tiene efecto.

Parámetros:

- name - Nombre de la venta que será usado como identificador y aparece en la barra de título.
- Flags - El único flag soportado es CV_WINDOW_AUTOSIZE. Si se introduce la ventana se redimensiona automáticamente en función del tamaño de lo mostrado en la ventana.

cv::CascadeClassifier face_cascade;

Para llevar a cabo una clasificación de imágenes buscando por el método de Viola-Jones algún patrón dentro de una imagen necesitamos un clasificador en cascada. Dentro de openCv podemos declarar un clasificador cv::CascadeClassifier y después utilizarlo con algún propósito.

face_cascade.load(face_cascade_name)

Los clasificadores en cascada están definidos en un formato concreto dentro de OpenCV, este formato está incluido dentro de ficheros xml que determinan cuantos pasos tiene el clasificador y que características de Haar utiliza.

OpenCV incorpora dentro de la librería algunos operadores y para utilizarlos debemos cargar el clasificador dentro del objeto cv::CascadeClassifier indicando la ruta del fichero en el parámetro name.

void CascadeClassifier::detectMultiScale(const Mat& image, vector<Rect>& objects, double scaleFactor=1.1, intminNeighbors=3, int flags=0, Size minSize=Size())

La función detectMultiScale lleva a cabo la detección de objetos dentro de una imagen. Los objetos encontrados son devueltos en un vector de Rectángulos que indican la localización y el tamaño.

Parámetros:

- *image* – Matriz de tipo CV_8U que contiene una imagen en la que se detectan objetos.
- *objects* – Vector de rectángulos que pasado por referencia retorna un objeto detectado por cada rectángulo.
- *ScaleFactor* – Durante el proceso de detección el sistema prueba diferentes tamaños para encontrar objetos dentro de las imágenes. Este factor indica cuanto se modifica ese tamaño en cada paso de detección.
- *MinNeighbors* – Especifica cuantos vecinos candidatos deberá tener un rectángulo candidato para mantenerlo.
- *flags* – Este parámetro no se utiliza en los nuevos clasificadores en Cascada y tiene el mismo significado que en la función cvHaarDetectObjects. La detección de caras está optimizada por CV_HAAR_SCALE_IMAGE más que por el método por defecto, 0 CV_HAAR_DO_CANNY_PRUNING[5].
- *MinSize* – El espacio mínimo posible de objeto. Objetos más pequeños que eso serán ignorados.

Por último necesitamos pintar los resultados de los objetos obtenidos por **detectMultiScale**:

void **rectangle**(Mat& img, Point pt1, Point pt2, const Scalar& color, int thickness=1, int lineType=8, int shift=0)

Dibuja un rectángulo con el tamaño de línea y tamaño indicado en la imagen introducida por parámetros.

Parámetros:

- *img* – Imagen donde queremos pintar.
- *pt1* – Uno de los vértices del rectángulo.
- *pt2* – El punto opuesto a pt1 formando una de las diagonales del rectángulo.
- *color* – Color del rectángulo o brillo en imagen en escala de grises.
- *thickness* – Grueso de la línea.

[5] Porque el método CV_HAAR_SCALE_IMAGE tiene más optimizado el acceso a memoria (*direct memory access friendly*). La interpretación por defecto (CV HAAR DO CANNY PRUNING) necesita accesos a memoria aleatorios muy frecuentemente.

- *LineType* – Tipos de lineas soportadas:
  - 8 (or omitted) 8-*connected line*.
  - 4 4-*connected line*.
  - CV_AA *antialiased line*.
- *shift* – Number of fractional bits in the point coordinates
- $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}\left(x * 2^{-shift}, y * 2^{-shift}\right)$

Ej: cv::rectangle(frame, rect,cv::Scalar(0, 0, 255),4);

5.1.4. Desarrollo de un aprendizaje guiado

5.1.4.1. Explicación Entrenamiento

Uno de los objetivos principales del proyecto es el de llevar a cabo un aprendizaje guiado que nos permita detectar caras dentro de imágenes.

Para entrenar un clasificador que funcione en primer lugar debemos obtener imágenes positivas que indiquen donde se encuentra el objeto que queremos obtener. En nuestro caso introduciremos caras frontales indicando donde se encuentra la cara dentro de la imagen con un formato adecuado explicado más adelante.

Además también necesitaremos imágenes negativas, es decir, cualquier cosa que no sea una cara. Esto permitirá a nuestro sistema descartar más fácilmente las subimágenes que puedan parecer una cara, pero en realidad no lo son.

Como el algoritmo de Viola-Jones utiliza un árbol de decisión, si podemos en los primeros pasos descartar el mayor número posible de subimágenes conseguiremos una identificación más rápida y más efectiva, y además reduciremos los falsos positivos.

Una vez hecho esto deberemos definir los parámetros configurables que permite la función haar-training de OpenCV en función del tipo de clasificador que queremos obtener.

```
./haartraining
  -data <dir_name>
  -vec <vec_file_name>
  -bg <background_file_name>
  [-npos <number_of_positive_samples = 2000>]
  [-nneg <number_of_negative_samples = 2000>]
  [-nstages <number_of_stages = 14>]
  [-nsplits <number_of_splits = 1>]
  [-mem <memory_in_MB = 200>]
  [-sym (default)] [-nonsym]
  [-minhitrate <min_hit_rate = 0.995000>]
  [-maxfalsealarm <max_false_alarm_rate = 0.500000>]
  [-weighttrimming <weight_trimming = 0.950000>]
  [-eqw]
  [-mode <BASIC (default) | CORE | ALL>]
  [-w <sample_width = 24>]
  [-h <sample_height = 24>]
  [-bt <DAB | RAB | LB | GAB (default)>]
  [-err <misclass (default) | gini | entropy>]
  [-maxtreesplits <max_number_of_splits_in_tree_cascade = 0>]
  [-minpos <min_number_of_positive_samples_per_cluster = 500>]
```

Dentro de nuestro entrenamiento podemos definir la tasa de fallo que queremos permitir y también la tasa de acierto. Una tasa de fallo demasiado baja no permitiría avanzar hacia soluciones locales válidas, es por ello que una tasa adecuada puede ser del 0.5, ya que descartaríamos la mitad de las posibles subimágenes en cada uno de los pasos.

En cuanto a la tasa de acierto pues depende de nuestra aplicación, si no se trata de una aplicación crítica podemos permitir tasas de acierto más baja para que la clasificación se haga más rápida, en cambio si queremos que nuestro análisis se haga de manera exhaustiva deberemos utilizar tasas de acierto elevadas, cercanas al 100%.

También debemos tener en cuenta el parámetro -sym (default)] [-nonsym]. En caso de que nuestro objeto no fuera simétrico horizontalmente podríamos usar -nonsym que obtiene un procesado más costoso. Si utilizamos la opción por defecto -sym la clasificación se realizará sólo sobre una de las mitades horizontales de nuestro objeto, de manera que ahorramos la mitad del procesado.

Además tenemos que considerar -ALL para activar el uso de todas las características de Haar. Si no se utiliza este parámetro el entrenamiento sólo utiliza las características de haar verticales y nos conviene que se usen las horizontales, verticales y las diagonales para identificar nuestros objetos. Esto provoca un aprendizaje más lento pero también más elaborado.

Por último, podemos definir cuanta memoria RAM queremos dedicar al aprendizaje, esto permitirá que se haga más rápido, dependerá por supuesto de las características técnicas del ordenador donde se realice. Para definirla usaremos "-mem 512", el número máximo son 2GB y por defecto la consulta utiliza 200 MB.

5.1.5. Obtención de Imágenes

En nuestro entrenamiento guiado necesitaremos un número de imágenes adecuado para poder conseguir un clasificador adecuado y los resultados sean los esperados.

El número de imágenes que necesitaremos vendrá estrechamente relacionado con la aplicación que queremos darle, si los objetos que debemos detectar representan una gran variación y el entorno puede variar considerablemente entonces necesitaremos un número de imágenes mayor.

En nuestro caso usaremos alrededor de 2000 imágenes positivas y 2000 negativas. Si los resultados no fueran los adecuados variaríamos estos valores. Es decir, si obtenemos un clasificador que no detecta bien las caras, necesitaremos más muestras positivas, si en cambio, si el número de falsos positivos es muy elevado, será el número de imágenes negativas lo que debemos variar.

Para obtener las imágenes que necesitaremos para el aprendizaje usaremos algunas bases de datos públicas que subministran por medio de subscripción o simplemente de manera libre algunas conjuntos de imágenes.

Dentro de estos organismos se encuentran en su mayoría universidades con laboratorios de investigación en Visión por Computador que han tenido que desarrollar una base de datos de caras para sus investigaciones.

- Las bases de datos usadas son:
- The Color FERET database (pertenece al NIST)[6]
- Speech, Vision and Robotics Group of the Cambridge University Engineering Department.[7]
- Audio Visual Technologies Group (Signal Theory and Communications Department de la UPC)[8]
- Algunas imágenes propias obtenidas de mi colección personal de imágenes reales, es decir, imágenes obtenidas sin ninguna intención de estudio.
- Libor Spacek's Facial Images Databases[9]-

Además para las pruebas de test, y con el objetivo de que las imágenes no sean tan preparadas y tenga un componente de aleatoriedad más grande, se han obtenido de Google Imágenes[10].

[6] http://www.nist.gov/itl/iad/ig/colorferet.cfm
[7] http://mi.eng.cam.ac.uk/research/speech/
[8] http://www.upc.edu/
[9] http://cswww.essex.ac.uk/mv/allfaces/index.html
[10] http://images.google.es/

5.1.6. Herramientas de procesado

Durante el proceso de entrenamiento se tiene que tratar con una gran número de ficheros que contienen las imágenes y con una gran número de imágenes que debemos clasificar que servirán como entrada de nuestro entrenamiento.

Es por esto que para hacer más fácil la manipulación he programado algunos scripts que realizan algunas tareas monótonas de manera automática.

En primer lugar, necesité un programa que me permitiera descomprimir todos los ficheros comprimidos que traen las librerías de imágenes, esto me permitía obtener una estructura de carpetas, donde en cada carpeta tenemos un grupo de imágenes relacionadas, ya sean de una misma persona o de un mismo tipo.

Una vez obtenidas las imágenes necesitábamos indicar para cada una donde se encuentran las caras y obtener un fichero que nos guarda esta información. Para ello usaremos el programa ObjectMarker que he modificado para que se adapte mejor al proyecto, pero que solo realiza el análisis para una carpeta concreta.

Por lo tanto, he necesitado un segundo script que ejecutaba ObjectMarker para cada una de las carpetas secuencialmente, de manera que podía para de marcar las imágenes como si estuvieran todas juntas.

Además este script detectaba si dentro de la carpeta ya se encontraba un fichero info.txt, lo que indicaba que ya había analizado esa carpeta. De esta manera podía parar el análisis (hay que pensar que son librerías de miles de imágenes y hacer el trabajo consecutivo es muy tedioso) cuando lo necesitara y continuar más adelante.

5.1.7. ObjectMarker

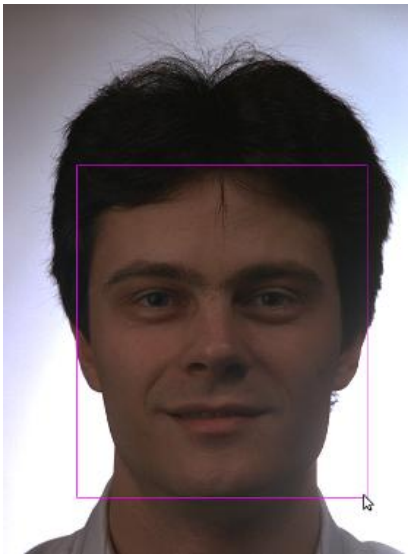
Como ya hemos indicado anteriormente el aprendizaje consiste en introducir una muestra significativa de imágenes positivas para que el algoritmo sea capaz de diferenciar entre lo que realmente queremos encontrar y lo que no.

En primer lugar se necesita un gran número de imágenes, alrededor de 3000 para poder identificar bien qué es una cara. Aunque el número dependerá en gran medida en la fisonomía del objeto que queremos detectar y de las condiciones ambientales, es decir, si el objeto está en un entorno controlado, donde hay poca posibilidad de obtener falsos positivos o falsos negativos el aprendizaje no será necesario que sea tan exigente.

Para realizar esto, debemos recopilar para el aprendizaje la posición de todas las caras dentro de las imágenes obtenidas. Es obvio que no podemos automatizar este proceso, ya que si pudiéramos detectar donde se encuentran las caras dentro de las imágenes iniciales estaríamos resolviendo el problema de base.

Es por ello que hemos tenido la necesidad de desarrollar una aplicación que permita de manera mecánica y manual indicar esto y obtener como resultado los ficheros indicando por cada imagen la posición y tamaño de las caras.

Esta aplicación a la que he llamado ObjectMarker recibe un directorio lleno de imágenes y genera un documento con el formato que haartraining necesita.



Cuando le indicamos un directorio, se explora para comprobar las imágenes que podemos encontrar en él. Hay que añadir que el programa permite la introducción filtros. Si añadimos en nuestra llamada el atributo "-ft" y seguidamente indicamos algunos filtros separados por ":", por ejemplo "ab:bc", solo las imágenes que contengan en su nombre ab o bc serán analizadas.

Una vez obtenidos los ficheros que se serán analizados se abre el primero. Ahora con el ratón debemos hacer un recuadro que englobe donde se encuentra la cara que queremos guardar. Una vez tenemos la cara con un recuadro encima de ella, debemos pulsar "S" (Save) que guarda la posición del objeto dentro de la imagen.

Si existen más caras dentro de la imagen debemos repetir la operación para cada uno de ellos. Una vez hayamos terminado de definir todos los objetos dentro de la imagen podemos pasar a la imagen siguiente con A (Forward).

Cuando hemos analizado todas las imágenes del directorio habremos obtenido un fichero info.txt dentro del directorio con el formato especificado y explicado en el apartado de este mismo capítulo "*info.txt".*

5.1.7.1. La disposición de las teclas

La disposición de las teclas no es casual, se ha intentado que las teclas estén cerca y que se pueda guardar y pasar a la siguiente imagen con una sola mano. En un teclado Qwerty la A está al lado de la S lo que permite que se pueda hacer el entrenamiento mucho más rápido y además dejamos libre la otra mano para hacer los recuadros con el ratón.

5.1.8. Ficheros info.txt

Ficheros que genera el ObjectMaker. Cómo los he procesador y unido. El formato que tienen y para qué sirven el haartraining.

En primer lugar, de la manera que tenía implementado el programa de generación de muestras, en cada carpeta de imágenes se creaba un fichero info.txt de la forma:

Path of the file	#samples	x	y	width	height
…/data/images/00001/00001_930831_fa_a.ppm	1	107	173	331	384

Una vez obtenido los ficheros de cada una de las carpetas, que representan las imágenes separadas por personas que aparecen en ellas, unimos los diferentes info.txt en un único fichero de datos.

Para hacer esto, desde bash podemos usar el comando **cat**. Nos situamos en la carpeta donde están alojadas las carpetas con sus imagenes, es decir, desde donde vemos todas las carpetas y ejecutamos:

cat */info.txt > info_frontal_DVD1-b.txt

De esta manera tenemos un fichero que define todas las posiciones donde se encuentran las caras en cada una de las muestras. Por último, queremos saber cuántas muestras hemos procesado. En nuestro caso, como en cada imagen sólo se podía ver una cara, podemos suponer que cada línea del fichero de clasificación, representa una muestra.

De esta forma si ejecutamos:

cat info_frontal_DVD1-b.txt | wc –l

Obtenemos el número de filas del documento, y sabremos el número de muestras que hemos procesado. Si son suficientes podemos pasar al siguiente caso de generar el fichero .vec de muestras.

5.1.8.1. Creación del fichero .vec de muestras

Ejecutamos:

opencv_createsamples -info info_frontal_DVD1-b.txt -vec salida.vec -num 2018

Desde la carpeta donde tenemos el fichero de datos y nos genera un fichero salida.vec que contiene las imágenes de muestras. En nuestro caso tenemos 2018 muestras, así que debemos especificárselo, si no hacemos esto, se generarán 1000 muestras que es valor por defecto.

En cuanto al tamaño de las muestras serán de 24x24 pixeles por defecto. Aunque podemos modificarlo con los argumentos -w para el width y -h para el height.

5.1.9. Haartraining

Una vez reunidos todos los elementos que necesitamos para poder realizar un entrenamiento guiado y hemos entendido la interacción que tienen cada uno de los parámetros de la función de haartraining sobre el resultado final, pasamos a generarlo.

Con los siguientes parámetros iniciamos nuestro primer entrenamiento guiado:

opencv_haartraining -data haarcascade -vec salida.vec -bg negatives.dat -nstages 20 -nsplits 2 -minhitrate 0.999 -maxfalsealarm 0.5 -npos 2018 -nneg 3019 -w 24 -h 24 -nonsym -mem 1024 -mode ALL

El proceso es altamente costoso en tiempo, en mi primer experimento creando un clasificador, el proceso ha tardado 4 días aproximadamente sin detener el ordenador en ningún momento.

En nuestro caso teníamos fijado que queríamos que nuestro clasificador acabará resultando en un árbol de decisión de 20 pasos, hay que tener en cuenta que si no tenemos las muestras necesarias el proceso llega hasta el último paso que es capaz de generar y entra en un estado de bucle infinito.

En nuestro caso, el proceso se quedó en un bucle infinito y había conseguido completar 13 pasos de la cascada. Cuando el proceso acaba de manera natural se crea un fichero xml que contiene el clasificador. Como no ha terminado tenemos que convertir los datos obtenidos en un fichero xml que se habría creado automáticamente si el proceso hubiera terminado.

Para hacer esto usaremos la herramienta convert_cascade que se encuentra en el código fuente de OpenCV. Descargamos el código fuente, versión 2.3.2 al crear esta documentación y entramos en “samples/c”.

Dentro de esto encontramos el fichero “convert_cascade.c” y lo compilamos. El modo de uso es el siguiente:

```
$ convert_cascade –size="<sample_width>x<sampe_height>" <haartraining_ouput_dir> <ouput_file>
```

Ejemplo:

```
$ convert_cascade --size="24x24" haarcascade haarcascade.xml
```

5.1.10.2ª iteración (Bootstrapping)

Después de las pruebas de rendimiento realizadas en la primera iteración hemos podido comprobar que nuestro clasificador no es todo lo preciso que nos gustaría. Por ello es necesario que realicemos una segunda iteración.

Una segunda iteración consiste en obtener, basándonos en las imágenes usadas para las pruebas en la primera iteración, una versión mejorada del primer clasificador.

Según hemos podido comprobar el clasificador funciona bien para algunos tipos de caras, y suele reconocer de forma correcta donde se encuentran las caras frontales en la imagen, pero sobretodo en imágenes con una resolución grande arroja demasiados falsos positivos. Es decir, reconoce bien los positivos pero no es lo suficientemente preciso para discriminar lo que no son caras.

Es por ello que necesitamos aumentar el número de muestras negativas en nuestro aprendizaje para que el clasificador sea más capaz de descartar subimágenes que correspondan con una cara.

De los resultados de la primera iteración hemos obtenido unos ficheros como los introducidos en el aprendizaje guiado[11] que nos indican donde se han encontrado caras en cada una de las imágenes, y también donde se encuentran los falsos positivos.

La idea principal de un segundo entrenamiento consistirá en introducir los falsos positivos detectados con el primer clasificador como muestras negativas del segundo clasificador para un mayor descarte.

Analizar en cada imagen cuales son correctas, y cuales representan falsos positivos sin un soporte visual sería muy difícil y costoso en tiempo, además de que tendría una gran posibilidad de error.

También se deben introducir como muestras positivas las caras que no hayan sido detectadas en las pruebas, por lo que deberemos también usar ObjectMarker para introducirlas.

[11] Hace referencia a los ficheros descritos en el apartado 5.1.8. Ficheros info.txt

5.1.10.1. Falsos Positivos

La manera más automática para obtener los falsos positivos es desarrollar un software que recoja un fichero de resultados y permita obtener un segundo fichero donde se eliminen las muestras que sean correctas y solo queden los falsos positivos.

En cuanto a los elementos negativos dentro de la segunda iteración, mantendremos las muestras negativas que usamos en la primera iteración (3019 muestras) y añadiremos otras que fueron detectados y que no correspondían a caras, es decir, falsos positivos (853 muestras en total)

Con esto usaremos 3872 muestras negativas para hacer un descarte eficiente en cada peldaño del árbol de decisión.

IMAGEN CON FALSOS POSITIVOS DETECTADOS 1

Este programa será desarrollado en C++ y utilizando la librería QT, la elección de este lenguaje es debido al coste de aprendizaje de usar otro tipo de lenguaje. Además ya tenemos un entorno configurado que hemos usado en la primera fase de uso de la librería OpenCV.

5.1.10.2. SetResults

SetResults es un programa que permite cargar un fichero de resultados de una detección de caras, analizar imagen a imagen cuales de los objetos detectados son en realidad falsos positivos y finalmente obtener un fichero con el mismo formato que contendrá solo estos falsos positivos.

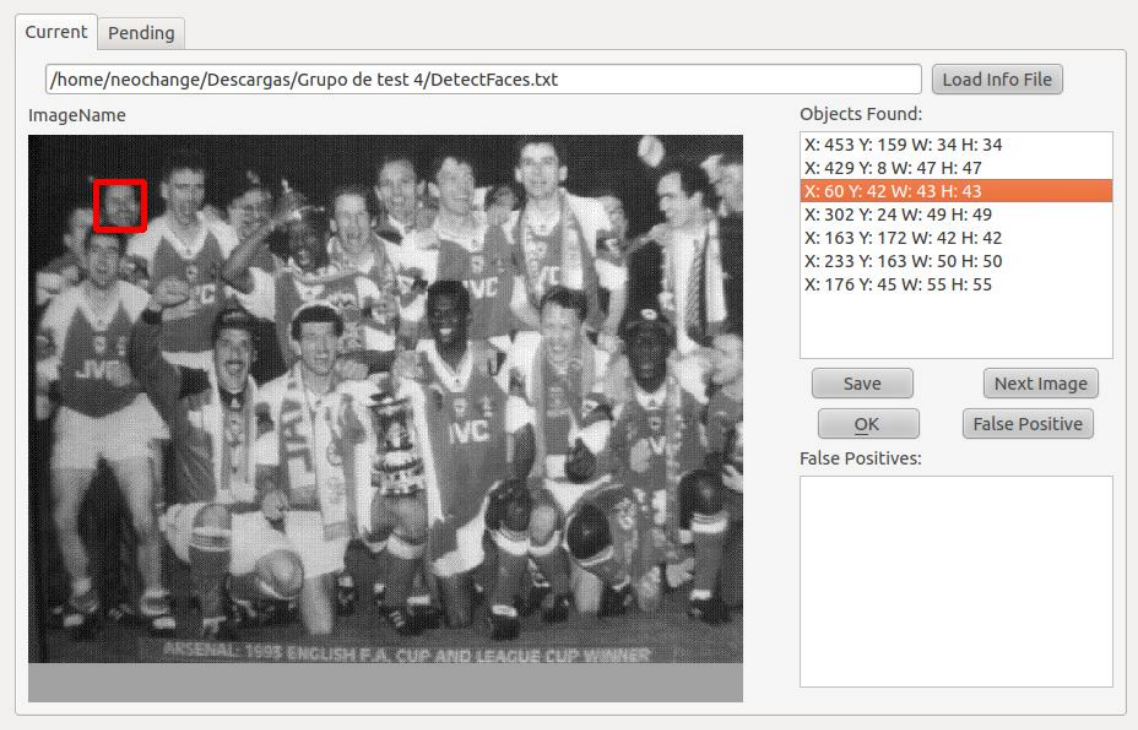
El programa nace de la imposibilidad de realizar manualmente la separación entre los elementos detectados correctamente y las detecciones erróneas.

En primer lugar, el programa permite abrir un fichero que contiene el formato de un fichero de muestras positivas necesitado por haartraining de Opencv:

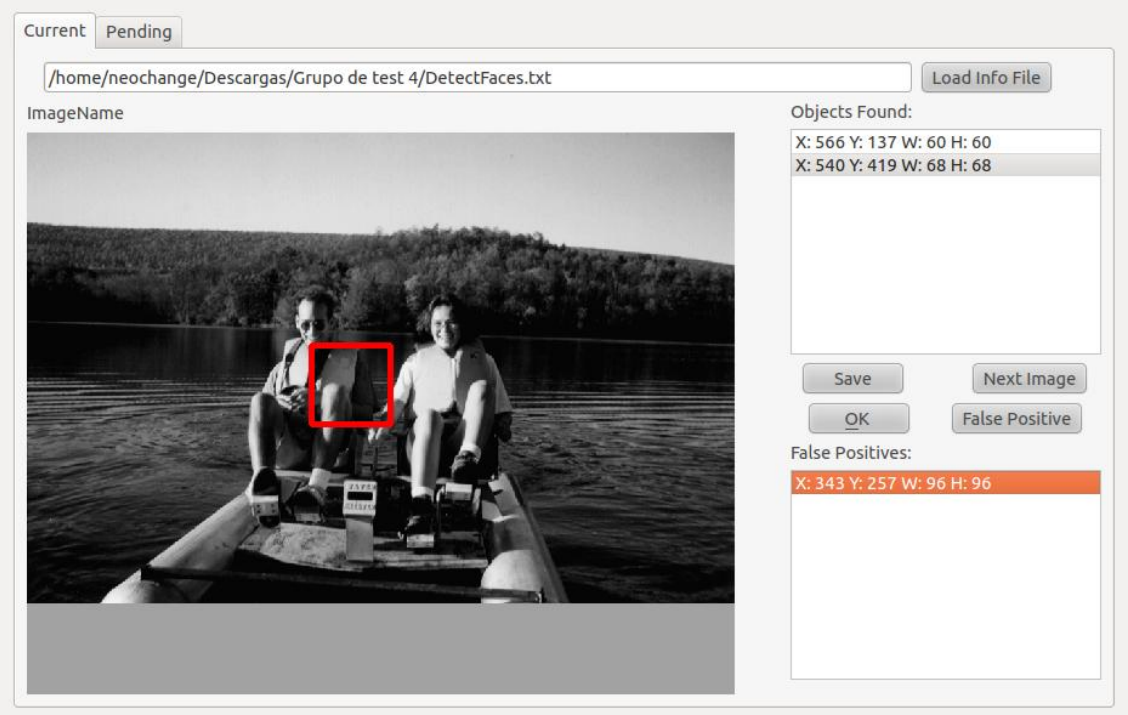
ruta_fichero num_muestras (x y width height)*

Una vez abierto permite visualizar por cada imagen los elementos detectados, y también una muestra redimensionada de la imagen en la izquierda de la interfaz. Si seleccionamos uno de los objetos detectados, este se nos muestra en la imagen de manera que podemos tomar una decisión.

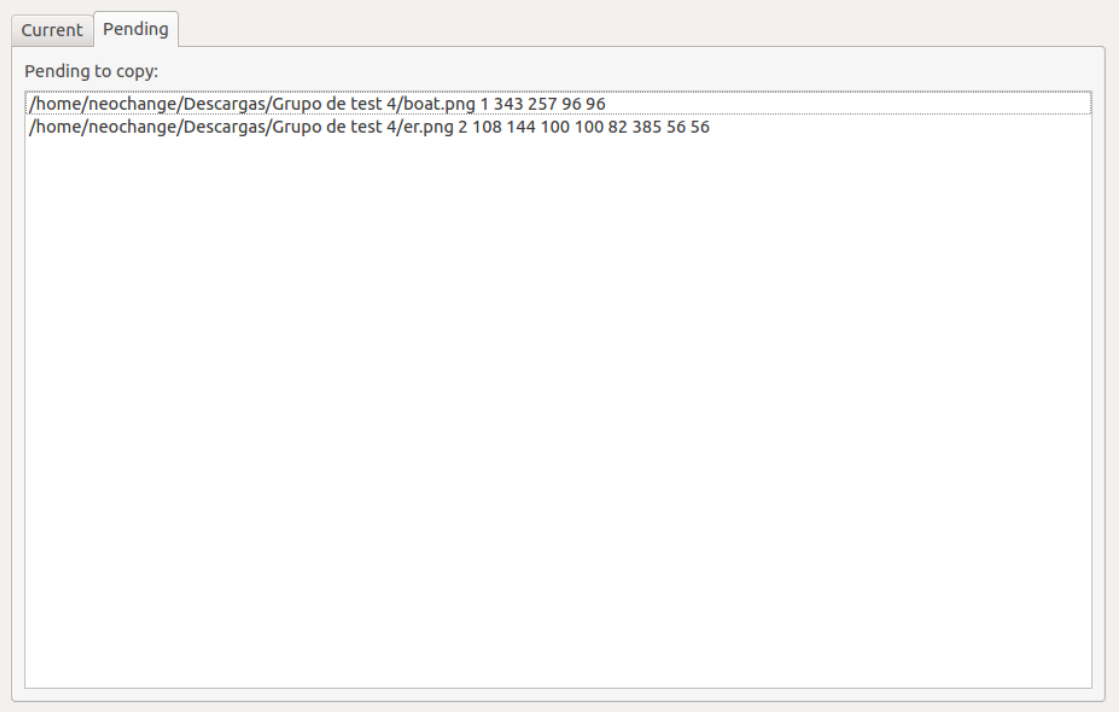
Si consideramos que el objeto es correcto podemos apretar el botón OK, en caso contrario apretaremos False Positive, lo que hará que el objeto se agrupe en una lista a parte.



Cuando ya tenemos todos los elementos de la imagen clasificados podemos apretar el botón NextImage que nos mostrará la siguiente imagen y podremos volver a iniciar el proceso anterior.



Según vamos analizando todas las imágenes, se va creando en la pestaña Pending, un documento con el formato que necesitamos, que contiene sólo las imágenes en las que hayamos seleccionado alguno de los objetos como falso positivo.



Finalmente, podemos apretar el botón Save para guardar el fichero resultado que podremos usar en para mejorar nuestro entrenamiento, indicando estas subimágenes como evitables.

5.1.10.3. Falsos negativos

El tratamiento de los falsos negativos se realiza para indicar al clasificador que debe considerar como muestras positivas, aquellas que no ha considerado antes, y que por inspección visual hemos comprobado que deben ser detectadas.

Como hemos visto en los resultados de la primera iteración, nuestro clasificador no ha sido capaz de detectar todas las caras que nosotros consideramos que debían haberse detectado.

Por ello, en la segunda iteración añadiremos la imágenes que ya teníamos y que usamos en el primer entrenamiento (1977 muestras), también usaremos 700 nuevas para engrosar más el número de muestras y por último añadiremos los falsos negativos detectados en las pruebas de nuestro clasificador (1637 muestras).

Esto hace un total de 4314 muestras positivas que introduciremos en la segunda iteración. El hecho de añadir nuevas subimágenes debería mejorar considerablemente el clasificador, ya que no sólo introducimos más muestras que permitirán hacer una mejor detección. Si no que además incluiremos las que resultaron fallidas en la primera vuelta.

IMAGEN CON ALGUNAS CARAS NO DETECTADAS 1

El número de muestras positivas que introducimos dependerá mucho del nivel de variabilidad que pueden presentar los objetos que queremos detectar. También hay que tener en cuenta que aunque estemos detectando el mismo objeto, las posibles variaciones en la iluminación, enfoque y otros fenómenos físicos, pueden influir en que un objeto sea detectado o no.

De esta manera si hemos hecho unas pruebas exhaustivas en la primera iteración, después del análisis en la segunda obtendremos un clasificador mucho más óptimo.

5.2. Detección de caras en Android

5.2.1. Introducción

Como fase final del proyecto se aplicará la detección de caras en un entorno embebido (embedded system) de manera que obtengamos una aplicación en para el sistema operativo Android que nos permita realizar la identificación de caras en imágenes obtenidas por el mismo dispositivo.

El desarrollo se valorará en 2 pasos, en primer lugar se alcanzará el objetivo de obtener una aplicación que permita realizar una fotografía, la analice y marque donde hemos detectado las caras, y finalmente lo guarde en el dispositivo.

Como mejora del proyecto tendríamos la posibilidad de adaptar la aplicación para que la detección se realice en tiempo real. Este objetivo puede quedar fuera del proyecto si la implementación fuera demasiado costosa en tiempo.

Uno de los problemas que nos encontraremos en esta etapa es la gran variedad de dispositivos que hay en este momento en el mercado. Además no solo la diversidad de resoluciones de pantalla, de tipos de cámara y de otros hardware derivados del dispositivo son importantes.

También deberemos tener en cuenta que existen algunas versiones de Android que pueden ejecutar nuestra aplicación y tendremos que hacerlo de la manera más diversa posible. Las versiones de Android son retrocompatibles entre ellas pero si programamos la aplicación en una versión muy reciente, hemos de tener en cuenta que los dispositivos con una versión anterior no serán capaces de ejecutarla.

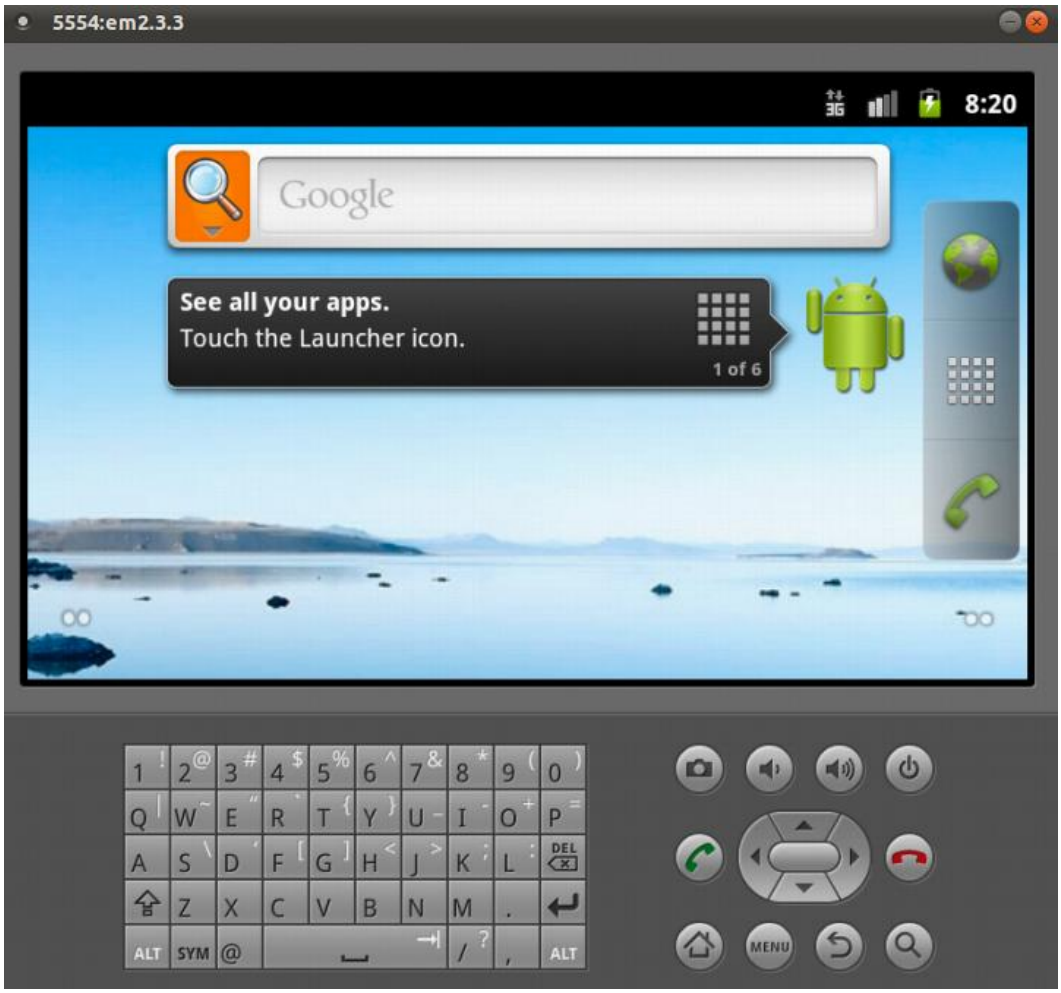
Esta sección pretende ser una guía de referencia para aquellos que quieran realizar una integración de OpenCV con Android en un entorno Linux, usando como entorno de desarrollo principal Eclipse.

5.2.2. Metodología

Durante el desarrollo de esta fase del proyecto centraremos los esfuerzos en programar una aplicación en un entorno controlado para luego aplicarlo en una aplicación real. Esta metodológica nos permitirá tener un prototipo de nuestra aplicación y desarrollar la implementación en 2 fases diferenciadas.

5.2.2.1. Primera etapa

En primer lugar generaremos una aplicación Android y la probaremos en un emulador. De esta manera también podremos realizar pruebas para las principales versiones del SDK de Android sin necesidad de disponer de un dispositivo con cada uno de estas versiones.



EMULADOR ANDROID 1

Android tiene un problema bien conocido en las forma en que se distribuyen las nuevas versiones de su sistema operativo. La instalación de las nuevas versiones depende exclusivamente de las operadoras de telefonía. Si las operadoras no están dispuestas a actualizar un dispositivo, este se queda desactualizado.

Esto ha provocado que haya un gran número de dispositivos en España que se quedarán con la versión que les subministro el proveedor de servicios y nunca se adaptarán a las nuevas versiones aunque su dispositivo se lo permita.

Que haya tantas versiones al mismo tiempo en el mercado provoca que desarrollar una aplicación específica sea muy complicado. Es por ello que por medio del emulador, y teniendo varias versiones de Android instaladas en nuestro equipo podemos probar en diferentes versiones al mismo tiempo.

5.2.2.2. Segunda etapa

En una segunda etapa nos basaremos en realizar experimentos en dispositivos reales, como los que podría tener cualquier persona, centrándonos si nos fuera posible en al menos 2 versiones de Android posibles.

El entorno Android permite 2 modos de depuración de nuestra aplicación. Podemos seleccionar el modo DEBUG de aplicaciones dentro del dispositivo, y probarlo conectando el dispositivo por USB o instalando el paquete .apk de instalación de Android que se crea cada vez que compilamos nuestra aplicación.

Nos decantaremos por la segunda opción. Ya que ésta es mucho más parecida a la forma en que los usuarios se instalarían la aplicación desde la tienda de aplicaciones oficial de Android. Aunque en nuestro caso no lleguemos a publicarla.

5.2.3. Entorno de trabajo

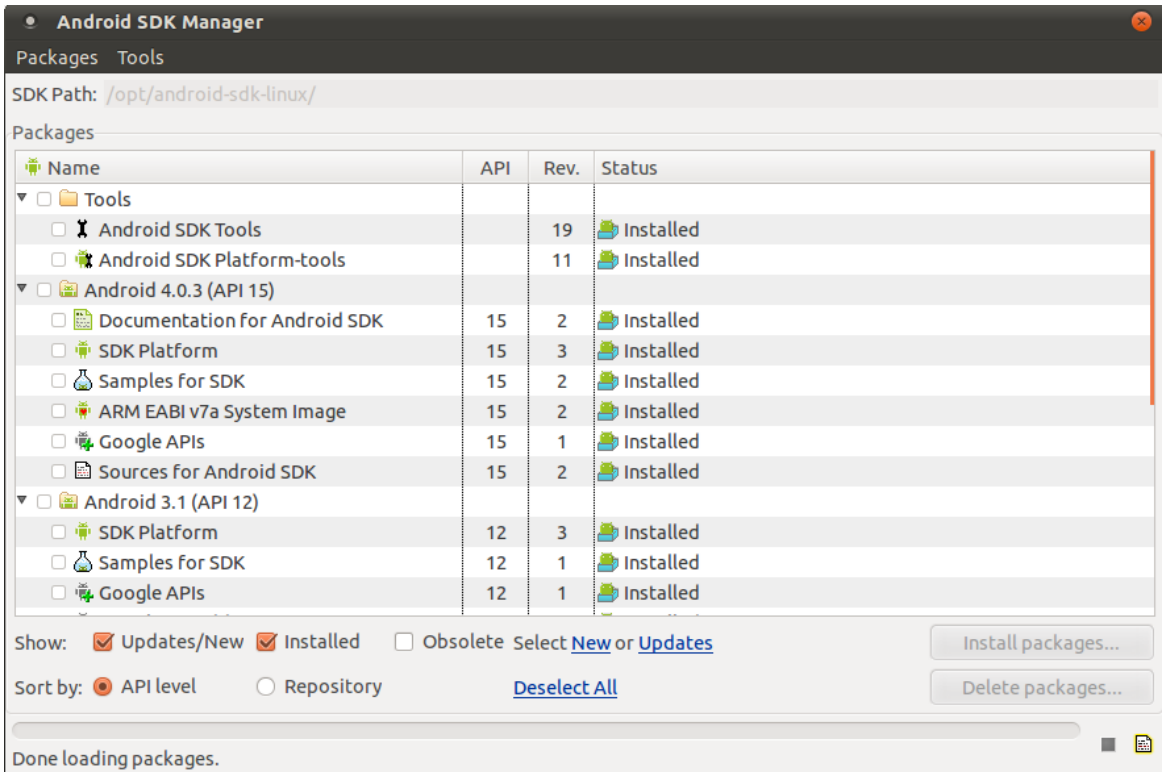
El elemento principal del entorno de trabajo que utilizaremos será el IDE[12] Eclipse. Este entorno de desarrollo basado en java permite la instalación de Add-ons (complementos añadidos para una tarea específica).

En nuestro caso el Add-on para Android nos permite tener algunas herramientas que nos hacen el desarrollo mucho más sencillo.

Entre estas herramientas que usaremos se encuentran:

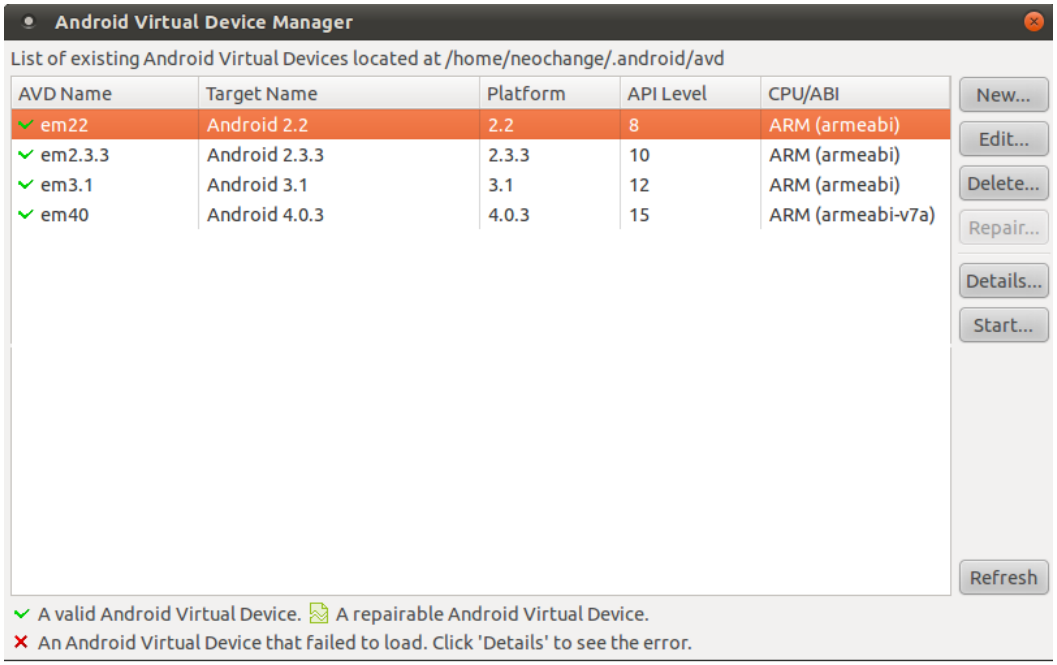
- SDK Manager
- Virtual Device Manager

Con el SDK Manager podemos gestionar las diferentes versiones de Android que tenemos instaladas en nuestro ordenador. De esta manera, si tenemos que instalar varias versiones para probar nuestra aplicación podemos hacerlo eficientemente, sin perder tiempo en instalaciones.



12 *Entorno de desarrollo integrado*

El virtual Device Manager gestionamos los dispositivos que tenemos instalados, es decir, los diferentes dispositivos Android virtuales que podremos usar para realizar las pruebas. Así podemos tener una emulador de cada versión.



5.2.3.1. Dispositivo físico

Una vez implementada la aplicación en un entorno virtual, llega el momento de probarlo en un dispositivo físico. Para ello disponemos de un Samsung Galaxy SII con la versión del SDK de Android 2.3.5.

5.2.4. Uso de la cámara

La librería Android soporta de manera nativa el uso de la cámara en los propios dispositivos. Este módulo nos permite capturar una imagen dándole esa orden o mantener una previsualización en tiempo real de los que se está visualizando en la pantalla.

Para poder usar la cámara en nuestra aplicación hemos tenido que seguir estos pasos:

1. Añadir en el fichero AndroidManifest.xml de nuestra aplicación las siguientes lineas:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

2. Obtener una instancia del objeto cámara con open(int).
3. Obtener los parámetros por defecto que configuran la cámara con getParameters().
4. Si fuera necesario los modificamos con setParameters(Camera.Parameters).
5. Si queremos definir la orientación usamos setDisplayOrientation(int).
6. Inicializamos un SurfaceHolder y lo asignamos con setPreviewDisplay(SurfaceHold).
7. Llamamos a startPreview() para que empecemos a ver los que la camara está captando.
8. En cualquier momento podemos llamar a takePicture(Camera.ShutterCallback, Camera.PictureCallback, Camera.PictureCallback, Camera.PictureCallback) para capturar una imagen.
9. Después de tomar una foto, la previsualización se detiene, para volverla a arrancar llamamos a startPreview() otra vez.
10. Para dejar de actualizar la superficie llamamos a stopPreview().
11. Finalmente llamamos a release() para liberar la cámara de nuestra aplicación. Todas las aplicaciones liberan la cámara cuando se ponen en onPause() y la vuelven a activar cuando se vuelve a ejecutar en onResume().

5.2.5. Integración con la librería OpenCV

La librería OpenCV está totalmente integrada con el framework[13] y la tecnología Android. Aunque OpenCV es una librería desarrollada mayoritariamente en C, para que sea mucho más eficiente, con el tiempo desarrollaron una versión para usar con Java. Esta versión solo fue el paso previo para dar soporte completo al sistema operativo Android de Google.

La librería de OpenCV para Android funciona como cualquier otra librería para Java. Normalmente la librería es un fichero con extensión .jar. Solo debemos incluirla en nuestro proyecto y ya tendremos disponibles todas sus funciones.

[13] Conjunto de librería, métodos y protocolos que nos permiten realizar alguna problemática particular http://es.wikipedia.org/wiki/Framework

5.2.6. Diseño

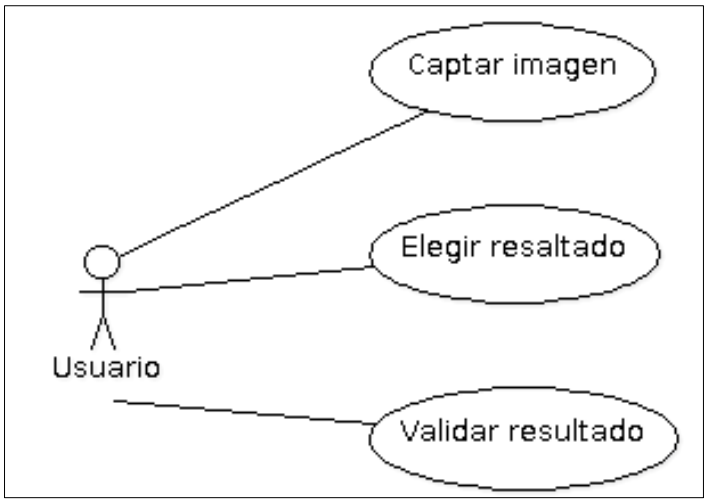
Aunque no realizaremos un diseño formal, basándonos en todas las reglas de la metodología UML[14], utilizaremos 2 diagramas que nos han servido para determinar el esqueleto principal de nuestra aplicación. Y también, nos permitirán tener una visión gráfica de lo que estamos construyendo.

En primer lugar usaremos el Diagrama de clases para poder determinar cual debe ser la mejor estructura de nuestro programa. También usaremos el Diagrama de flujo que nos permite crear las relaciones de dependencia que existen entre los diferentes casos de uso de nuestra aplicación.

5.2.6.1. Procesos

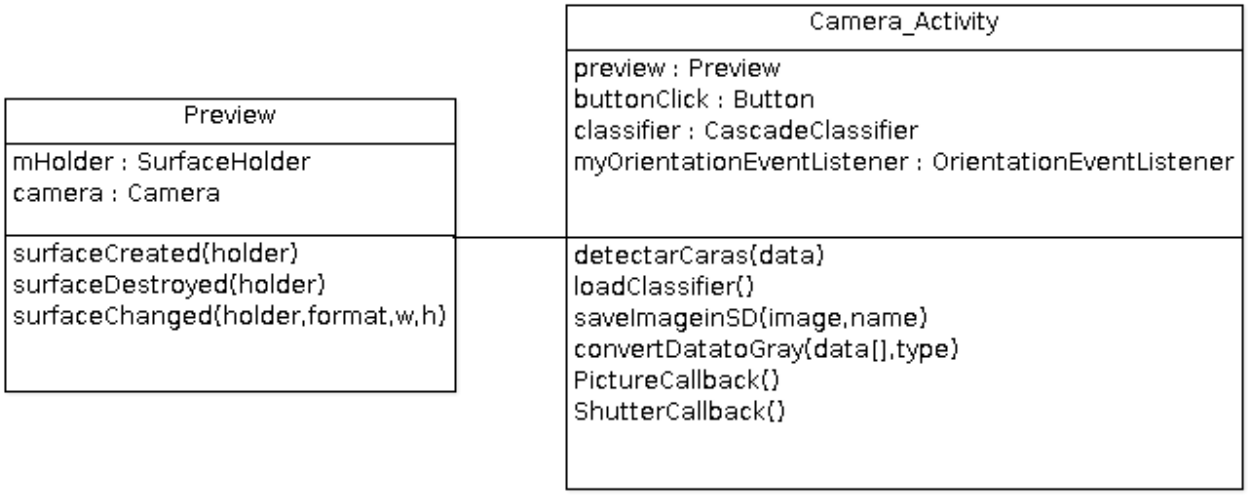
- Mostrar imágenes obtenidas por la cámara.
- Captar imagen.
- Analizar caras dentro de la imagen.
- Realizar aplicación tratado de la zona detectada.
- Validar resultado.

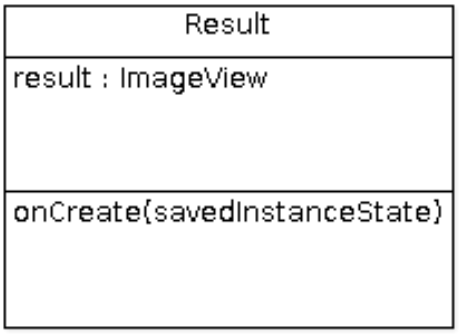
5.2.6.2. Diagrama de casos de uso



[14] Lenguaje Unificado de Modelado

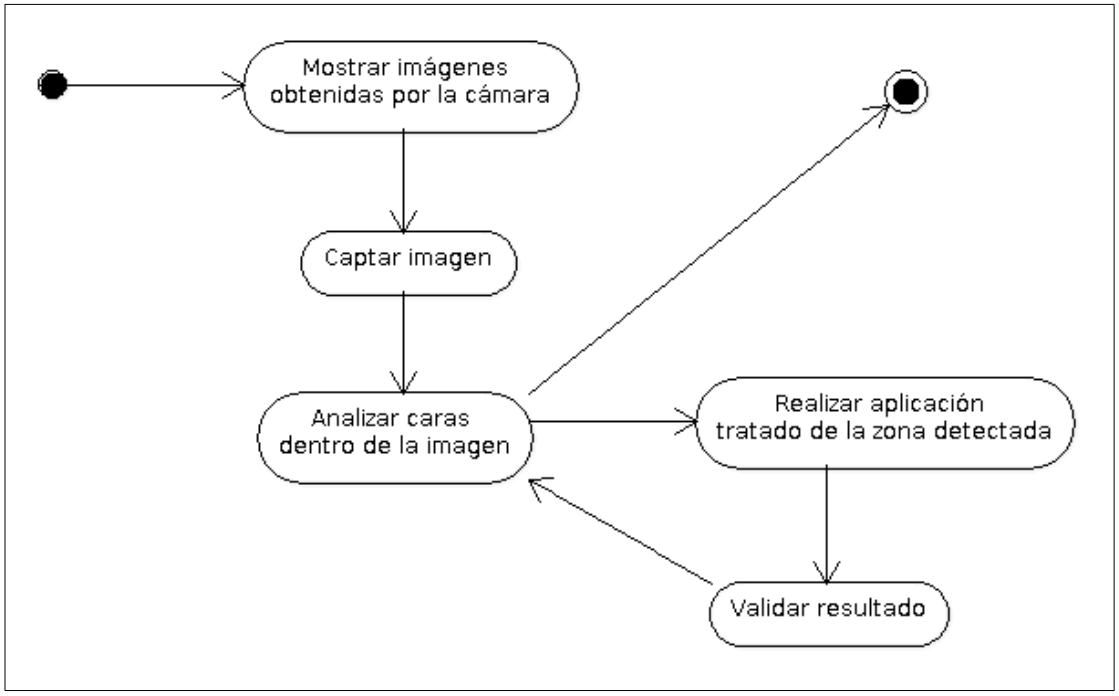
5.2.6.3. Diagrama de clases





5.2.6.4. Diagrama de flujo

El diagrama de flujo nos permite ver los estados por los que pasa nuestra aplicación:



6. Resultados

Durante cada una de las fases que hemos desarrollando del proyecto, hemos realizado algunos experimentos que nos han permitido saber en cada una de las etapas si el resultado era el esperado o si el procedimiento que estábamos siguiendo era el adecuado.

De esta manera hemos desarrollado una metodología para realizar los experimentos que describiremos a continuación:

1. Pruebas del clasificador TEST
2. Comparación clasificador propio con clasificador TEST
3. Pruebas intensivas de pruebas en clasificador propio
4. Bootstraping, comparación con primer clasificador
5. Pruebas aplicación Android

En cada una de las pruebas usaremos el software de apoyo que nos hemos creado para poder obtener resultados más fácilmente y podamos realizar los experimentos con un gran número de imágenes de manera tratable.

6.1. TEST

6.1.1. Preparación de las pruebas

Para ello, en primer lugar hemos recogido algunas imágenes aleatorias, obtenidas al azar con la aplicación Google Imágenes [15] y con las que hemos obtenido los siguientes resultados.

Aunque podemos imaginar lo que queremos decir cuando decimos que queremos detectar caras en una imagen, para un mejor análisis debemos poder describir diferentes grupos de caras.

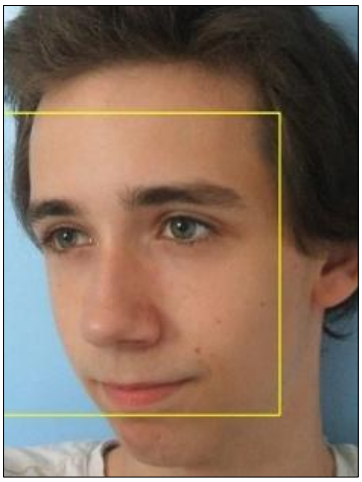
Debemos indicar que tipos de caras tenemos, y que consideramos como un rostro válido. En primer lugar decir que no entraremos en descripciones filosóficas sobre en que consiste una cara, sólo determinaremos algunos grupos.

[15] https://www.google.es/imghp?hl=es&tab=ii

En referencia a nuestro clasificador entendemos 3 tipos de caras:

- Caras Frontales: Caras donde se aprecia completamente la parte frontal de una cara, desde una oreja a la otra.

- Caras laterales: Rostros ligeramente ladeados donde no se pueda apreciar un lado igual que el otro. Es decir, nos perdemos algún detalle de uno de los lados.



- Caras de perfil: Caras donde sólo seamos capaces de ver la mitad de una cara. Las imágenes de perfil serán indetectables en nuestro clasificador.

Nuestro filtro deberá ser eficiente en caras frontales, por lo tanto, deberemos hacer la prueba con una mayoría de este tipo de rostros. En concreto hemos seleccionado 187 caras frontales, 26 laterales y 27 de perfil.

6.1.1.1. Resultados y conclusiones TEST

Para analizar los resultados tendremos en cuenta que caras han sido detectadas correctamente, cuales no se han detectado de ninguna manera (falsos negativos) y que detecciones no corresponden con ninguna cara (falsos positivos).

Las gráficas muestran resultados en función de las caras que se detectaron correctamente y las que no. Es decir, nos centramos en cuanto de lo que queríamos detectar conseguimos detectarlo.

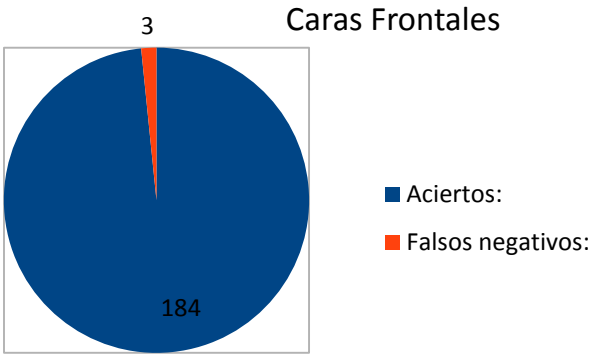
Imágenes Frontales

Caras: 187

Aciertos: 184

Falsos positivos: 2

Falsos negativos: 3



Observamos que para caras totalmente frontales el clasificador de TEST es bastante efectivo, y no genera demasiados falsos positivos.

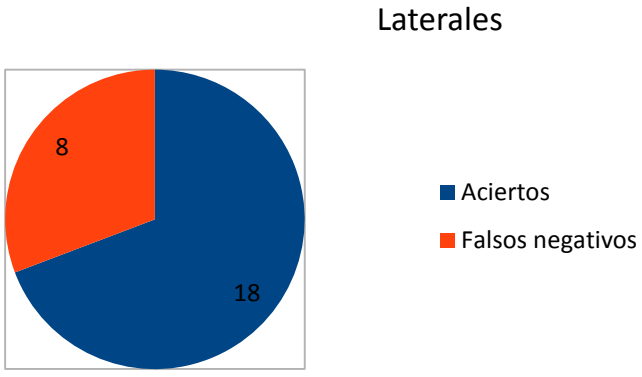
Imágenes laterales

Caras: 26

Aciertos: 18

Falsos positivos: 5

Falsos negativos: 8



En cuanto a las caras laterales, es decir. Ligeramente ladeadas, vemos que disminuyen los aciertos.

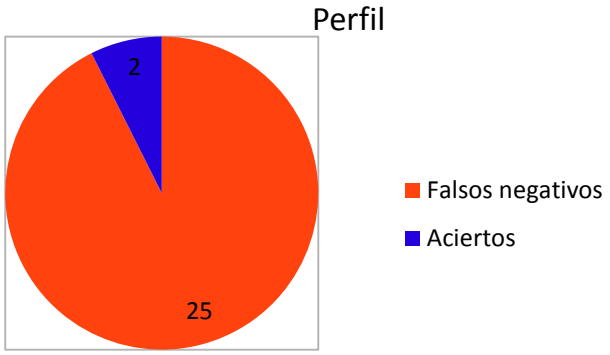
Imágenes de perfil

Caras: 27

Aciertos: 2

Falsos positivos: 1

Falsos negativos: 25



Finalmente vemos que no es nada efectivo para las imágenes de perfil. Solo ha conseguido detectar un porcentaje muy bajo de las caras.

En nuestro clasificador debemos intentar que sea más versátil, aunque no llegue a detectar las caras de perfil, ya que esto podría provocar que dejara de detectar bien las caras frontales, sí que debemos conseguir que una cara ladeada ligeramente sea detectada correctamente. Esto permitirá detectar caras en fotografías reales donde raramente se mira frontalmente a al objetivo. Nuestro clasificador debe ser más versátil.

6.2. Pruebas Clasificador propio

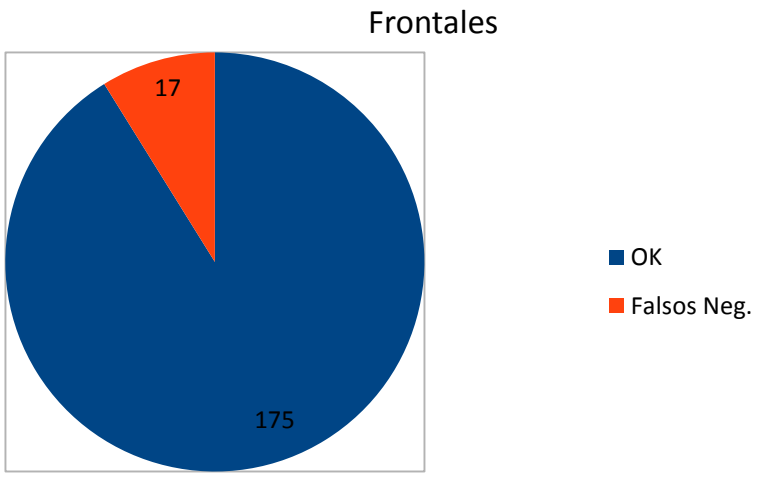
6.2.1. 1ª iteración

Una vez hemos realizado nuestro propio clasificador, las pruebas se basarán en compararlo con el clasificador que la librería OpenCV trae por defecto. Esto nos permitirá hacernos una idea de si nuestro entrenamiento ha sido acertado, o debemos mejorarlo.

Los resultados de clasificar las imágenes que hemos usado en el clasificador TEST con nuestro clasificador son:

Frontales

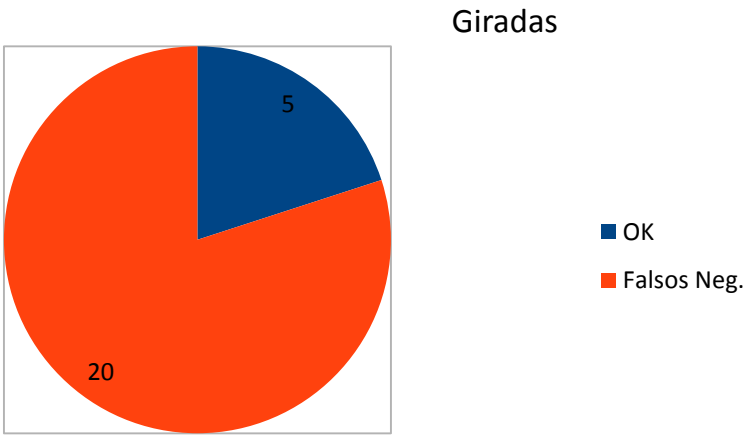
OK	175
Falsos Neg.	17
Falsos Pos.	54



Después del análisis de las caras frontales podemos ver que aunque no es muy significativo, nuestro clasificador arroja algunos aciertos menos que el clasificador de serie. También que se detectan más falsos positivos que se deberían reducir.

Giradas

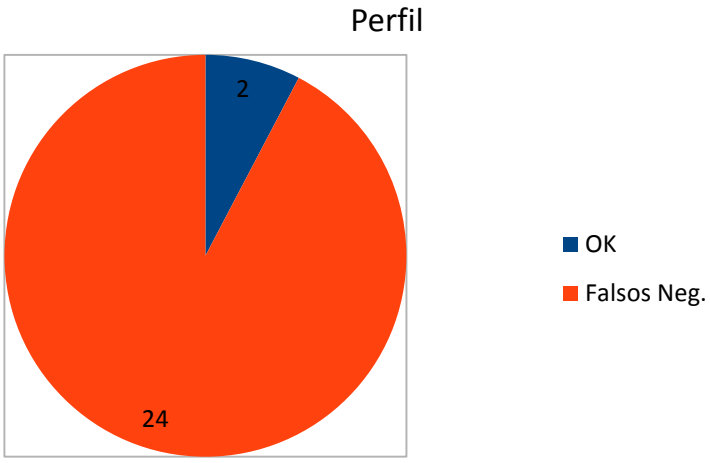
OK	5
Falsos Neg.	20
Falsos Pos.	7



En cuanto a las caras giradas hemos obtenido 5 caras bien detectadas, un porcentaje bajo frente a los 18 aciertos del clasificador TEST.

Perfil

OK	2
Falsos Neg.	24
Falsos Pos.	8



Por último hemos podido observar que las imágenes de perfil no son bien analizadas. Obtenemos unos resultados escasos aunque no queremos centrarnos en este grupo así que es normal que los resultados sean poco positivos. Si intentamos tener un abanico de caras diferentes demasiado algo puede provocar que deje de detectar correctamente.

Pruebas extensas IT1

Para realizar una mejor comprobación de la efectividad de nuestro clasificador hemos realizado unos grupos de imágenes de tipos variados que nos permitan hacernos una idea de la efectividad de nuestro clasificador en situaciones mucho más variadas.

En el momento de realizar las pruebas hemos realizado 5 grupos:

1. Imágenes pequeñas, en blanco y negro y totalmente frontales
2. Imágenes en color, posición variada (frontal, perfil y ladeadas) con fondo neutro
1. Imágenes totalmente frontales, con fondo verde no variable, diferente iluminación, solo modelos femeninos
3. Imágenes grandes y muy variadas, con cambios de luz, color, posición y con fondo totalmente cambiante.
4. Imágenes totalmente frontales, similares al grupo 3 pero solo modelos masculinos.

Con la definición de estos grupos se pretende identificar en que puede estar fallando nuestro clasificador y cómo podemos mejorarlo en el caso de realizar una segunda iteración.

El primer grupo por ejemplo, se intenta comprobar si el clasificador es capaz de detectar caras donde el tamaño del objeto a detectar esté cerca del mínimo permitido. En cuanto al segundo es una prueba con imágenes en color, donde se puede ver la versatilidad de detectar caras en diferentes posiciones.

El tercer y quinto grupo nos permite comprobar si hay variaciones entre hombres y mujeres, por si necesitáramos incorporar imágenes de uno de los grupos, ya que la identificación puede ser diferente debido a elementos en la cara como barbas en caso de los hombres o común pelo largo en mujeres, aunque esta característica también se da en hombre aunque menos frecuente.

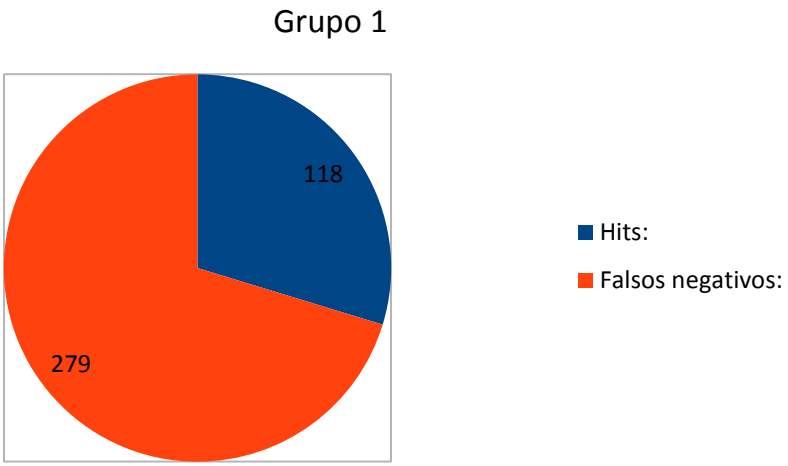
Por último, el grupo 4 es una prueba dentro de un entorno real. Utilizamos fotos reales, es decir, que no han sido preparadas en un laboratorio, y que contienen en su mayoría grandes tamaños y una gran variación de colores. La luz y el fondo tampoco está estandarizado por lo que es posible que se aumente la detección de falsos positivos.

6.2.1.1. Resultados

Los resultados de las pruebas con los grupos de imágenes del mencionados en el apartado anterior son los siguientes.

Grupo 1

	Numero	Porcentaje
Hits:	118	29,72%
Falsos Positivos:	3	
Falsos negativos:	279	70,28%
Total:	397	

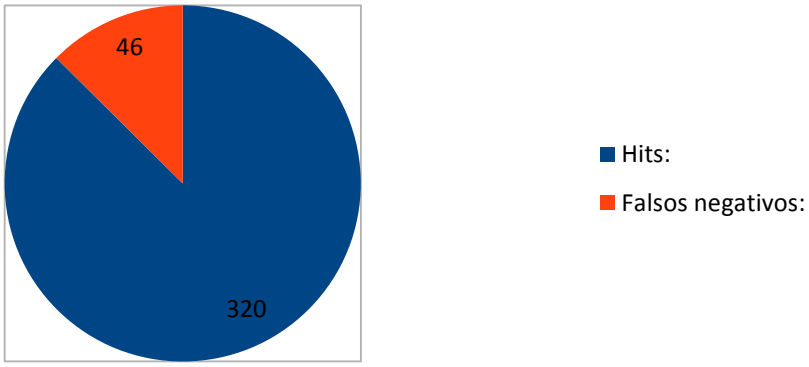


Analizando los resultados del primer grupo vemos que el clasificador tiene problemas para detectar caras con un tamaño muy pequeño. Esto puede solucionarse reduciendo el tamaño mínimo de detección.

Grupo 2

	Numero	Porcentaje
Hits:	320	87,43%
Falsos Positivos:	53	
Falsos negativos:	46	12,57%
Total	366	

Grupo 2

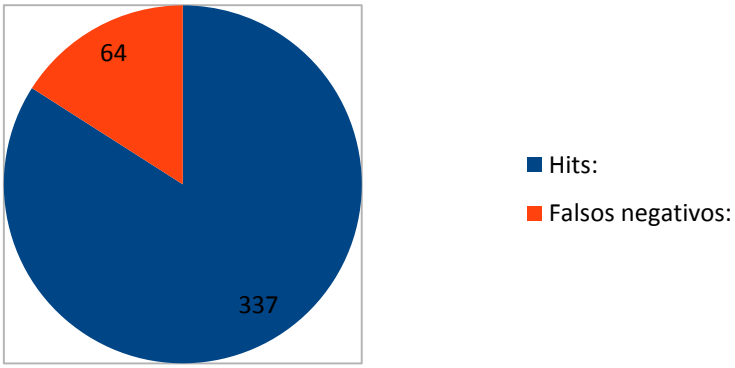


En el segundo grupo mejora el índice de detección respecto al primer grupo pero no aún así no detecta un 10.98% de las caras, esto es debido a las variaciones en la posición de este grupo, deberemos hacer más versátil nuestro clasificador.

Grupo 3

	Numero	Porcentaje
Hits:	337	84,04%
Falsos Positivos:	2	
Falsos negativos:	64	15,96%
Total	401	

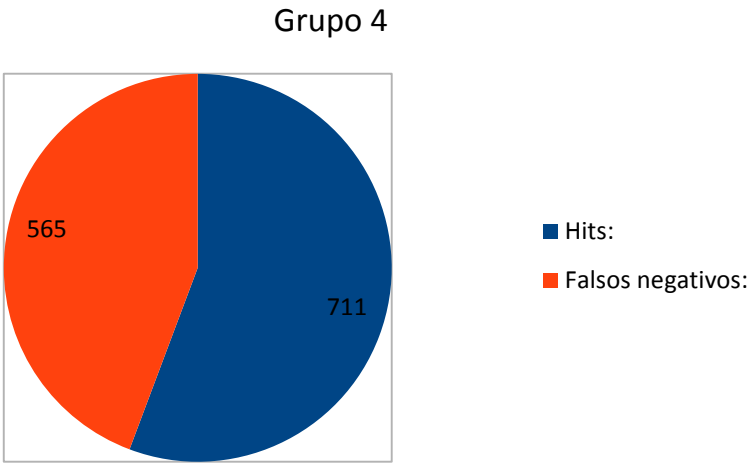
Grupo 3



Los resultados del grupo 3 indican que no acaba de detectarse correctamente algunas caras. Un aumento de las muestras positivas permitiría un nivel de detección mayor.

Grupo 4

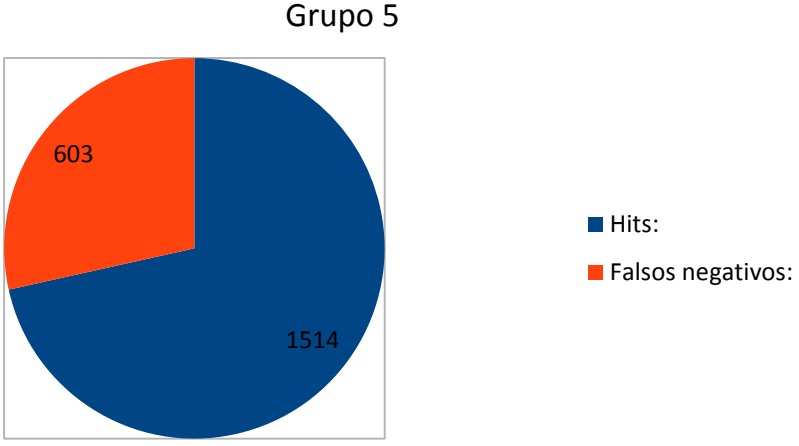
	Numero	Porcentaje
Hits:	711	55,72%
Falsos Positivos:	799	
Falsos negativos:	565	44,28%
Total	1276	



El cuarto grupo es el menos heterogéneo de todos. Podemos ver que el número de caras detectadas correctamente es bajo. En este caso, como los elementos analizados son muy variables, debemos garantizar que nuestro clasificador obtiene como muestras positivas las imágenes no detectadas en caso de realizar una segunda iteración.

Grupo 5

	Numero	Porcentaje
Hits:	1514	71,52%
Falsos Positivos:	144	
Falsos negativos:	603	28,48%
Total	2117	



En cuanto al último grupo podemos concretar lo mismo que para el grupo 3, aunque podemos deducir que nuestro clasificador es mejor detectando muestras de mujeres que de hombres, lo que quizás nos haga centrarnos más en grupos de hombre para aumentar el rango de detección.

Consideraciones finales 1a Iteración

De los resultados obtenidos podemos sacar una conclusión de la primera iteración, nuestro clasificador no es lo preciso que nos gustaría y deberemos hacer una segunda iteración.

En primer lugar, debemos mejorar la clasificación de imágenes pequeñas, esto podemos hacerlo reduciendo el tamaño mínimo de detección. Aunque esto puede provocar que se generan más falsos positivos, tendremos que analizar los resultados de hacer esto para comprobar si vale la pena.

También tendremos que incrementar el número de caras detectadas, así que habrá que añadir más caras positivas, una buena estrategia será añadir las que no han sido detectadas en esta iteración, para que el clasificador las reconozca.

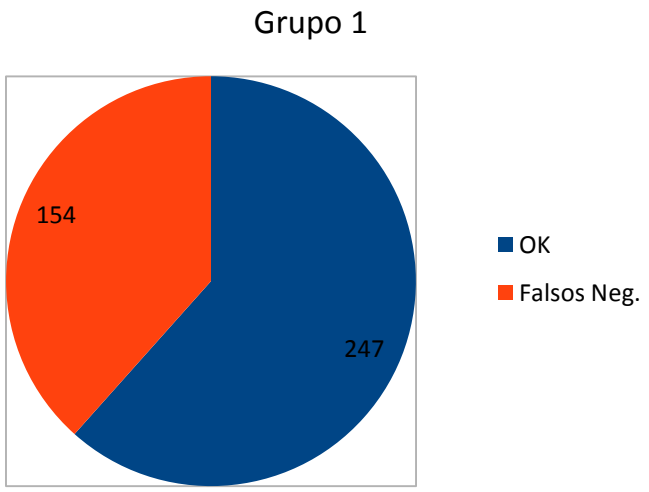
Por último deberemos conseguir que es descarten los falsos positivos, así que sería conveniente utilizar estas muestras como muestras negativas, para que no vuelvan a ser detectadas incorrectamente.

6.2.1.2. Resultados 2 IT

Para valorar si la segunda iteración ha realizado un procedimiento correcto, debemos compararlo con el primer clasificador. Las pruebas consistirán en realizar detecciones en todas las imágenes en las que probamos el rendimiento del primer clasificador. Estás pruebas nos permitirán tener la certeza que la segunda iteración representa una mejora.

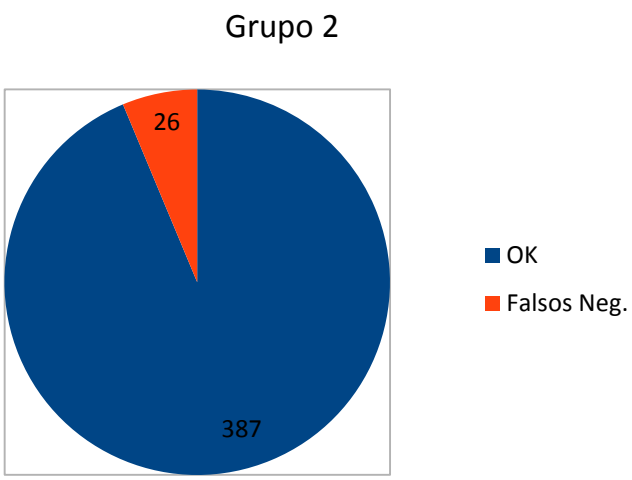
Los resultados se hacen en comparación con los resultado de la primera iteración donde teníamos 5 grupos de imágenes. Para esos grupos con el segundo clasificador propio hemos obtenido los siguientes resultados:

	Num
OK	247
Falsos Pos.	40
Falsos Neg.	154
Total	401



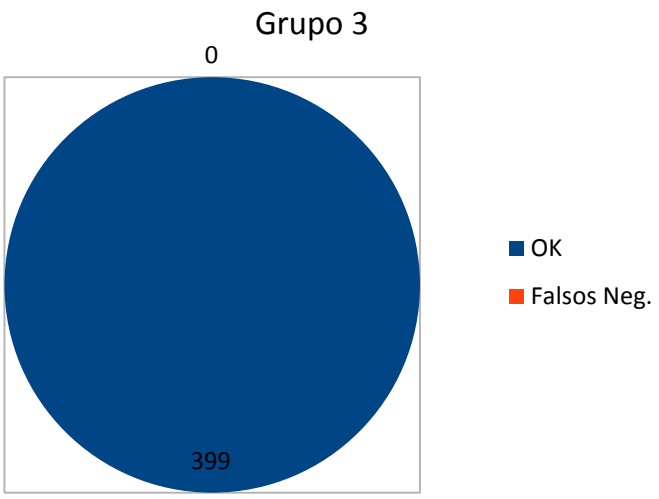
El primer grupo de imágenes, las imágenes pequeñas en blanco y negro, representan una gran mejora respecto el primer clasificador. Estas imágenes tan pequeñas debemos decidir si son nuestro objetivo o por el contrario deberíamos centrarnos en mejorar otro grupo en función de la aplicación final para la que está pensada el clasificador.

	Num
OK	387
Falsos Pos.	66
Falsos Neg.	26
Total	413



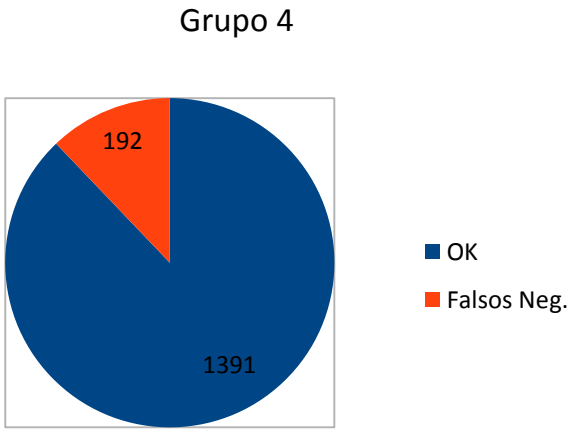
El segundo grupo vemos que no representa una mejora significativa respecto la primera iteración ya que el porcentaje de acierto era del 87% ya en esta. Aun así, vemos que hemos reducido los falsos positivos casi a la mitad.

	Num
OK	399
Falsos Pos.	23
Falsos Neg.	0
Total	399



El tercer grupo es exactamente lo que estamos buscando. Hemos conseguido un índice detección del 100% en caras totalmente frontales en este grupo de imágenes. Además podemos ver que los falsos positivos no han aumentado en exceso, lo que nos permite pensar que hemos logrado el ratio de detección que queríamos en este tipo de imágenes.

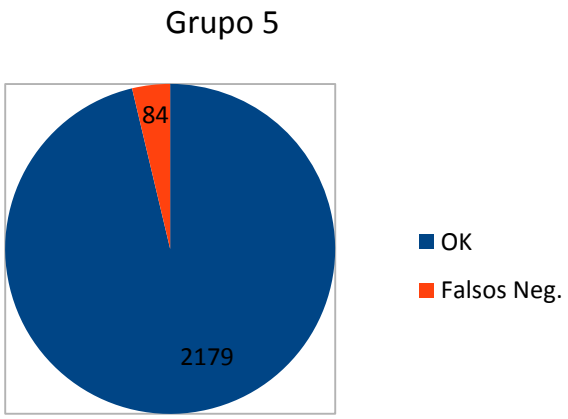
	Num
OK	1391
Falsos Pos.	1451
Falsos Neg.	192
Total	1583



En el grupo 4 hemos detectado un posible problema. Aunque el porcentaje de aciertos ha mejorado considerablemente hasta el 87.87% de acierto, podemos ver que los falsos positivos han aumentado considerablemente.

Esto hace que nuestro clasificador no sea muy fiable si se trata de imágenes con una resolución elevada y donde necesitemos descartar muy fielmente las cosas que no son caras.

	Num
OK	2179
Falsos Pos.	109
Falsos Neg.	84
Total	2263



El grupo 5 es el más similar al grupo 3 aunque no obtenemos un porcentaje tan elevado, obtenemos un porcentaje de aciertos del 96.28% lo que podemos determinar que para caras totalmente frontales nuestro clasificador es bastante fiable.

6.2.1.3. Tabla comparativa

Tabla comparativa que nos permite comparar las pruebas entre la primera iteración y la segunda en función del número correcto de detecciones.

		Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5
1ª Iteración	**Hits:**	118	320	337	771	1514
	Falsos Neg.	279	46	64	565	603
	Porcentaje Hits:	29,72	87,43	84,04	57,71	71,52
2ª Iteración	**Hits:**	247	387	399	1391	2179
	Falsos Neg.	154	26	0	192	84
	Porcentaje Hits:	61,60	93,70	100,00	87,87	96,29

Con los datos de la segunda iteración podemos comprobar que el primer grupo ha mejorado un 31,87%, un 6.27% el segundo grupo que tenía un índice de acierto alto (87,43%), en tercer y quinto grupo han mejorado un 15,96% y un 24,77% alcanzando porcentajes de acierto del 100% en el grupo 3 y cercanos a esto en el grupo 5. En cuanto al grupo 4 ha mejorado un 30,16% que le ha permitido mejorar bastante.

Analizando detenidamente las tasas de detección vemos que excepto en el primer grupo se ha conseguido un porcentaje de detecciones correctas por encima del 87%. Esto nos permite determinar que para las situaciones donde detectamos caras frontales nuestro clasificador funcionará correctamente, aunque para imágenes grandes con muchas variaciones el fondo puede suponer algún problema y podría mejorar.

Por estos datos deducimos que hemos mejorado el clasificador pero el aumento de detecciones ha creado una contrapartida. Aumentar el índice de detección ha provocado que también aumente el número de falsos positivos. Sobre todo en el grupo donde tenemos una variación mayor en el grupo de fondos o tamaños de las imágenes.

El hecho de que aumente el número de positivos puede contraproducente solo en función de la utilidad que le vamos a nuestro clasificador. Si nuestra aplicación debe tener discriminados, desde el inicio, posibles falsos positivos, nuestro clasificador debería mejorar en ese aspecto. En cambio, si por medio de precondiciones, por ejemplo la aplicación final sabe que los elementos positivos estarán en un rango de tamaño determinado y puede discriminar, entonces sería un clasificador valido para uso intensivo.

6.2.1.4. Valoración general final

Después del estudio del clasificador y analizar los resultados podemos determinar que hemos realizado estudio exhaustivo sobre la clasificación de caras y la certeza de poder generar un entrenamiento guiado que nos ha llevado a un clasificador de caras basado en los características de Haar y a un clasificador ADABoost.

Una vez dicho esto, hemos de valorar el resultado que hemos obtenido con nuestro clasificador. En cuanto a eficacia, hemos obtenido una media de 87,4% de porcentaje de acierto, lo que nos permitiría sobradamente detectar objetos en un entorno controlado.

Antes de usar nuestro clasificador podemos plantearnos de realizar una tercera iteración. Una tercera iteración nos permitiría reducir el número de falsos positivos y hacerlo más robusto.

En cuanto a nuestra aplicación no sería necesario, ya que en la detección podemos descartar aquellos objetos que sean poco sean más pequeños que un umbral, ya que una detección en que tuviéramos en cuenta objetos muy pequeños no sería rápida ni eficiente, y podría llevar a error en el momento de determinar que objetos son válidos.

La conclusión final es que hemos creado un clasificador robusto, que nos permitirá detectar caras en la mayoría de las situaciones y hemos realizado un entrenamiento guiado, en que hemos obtenido un conocimiento importante, además hemos podido analizar los diferentes aspectos que engloba todo el proceso de aprendizaje.

6.3. Pruebas Android

La versión 4.0 es la primera que tiene soporte para la cámara de Android desde el emulador con una cámara web.

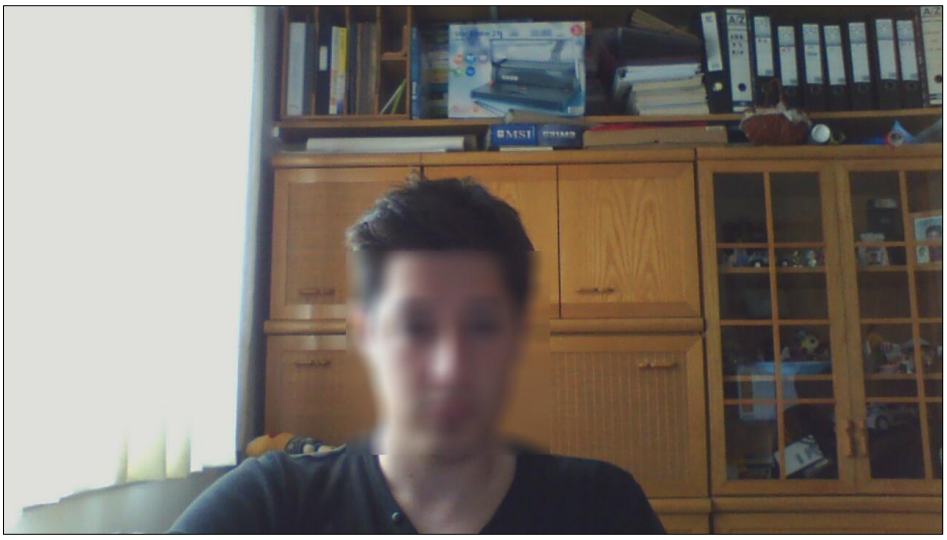
Las pruebas de la aplicación las he realizado en el emulador, de esta manera he podido debuggar la aplicación hasta que ha estado lista para ser usada en un dispositivo.

Una vez hemos tenido una aplicación final hemos conseguido un fichero .apk y lo hemos instalado en el dispositivo físico que usamos en las pruebas.

La respuesta del dispositivo frente a la aplicación es buena y fluida, hemos optimizado la detección para que funcione bien en dispositivos con una resolución de cámara elevada. En este caso, el dispositivo dispone de una cámara de 8Mpx lo que hace la clasificación un poco lenta. Considerando esto hemos seleccionado el tamaño mínimo de detección, es decir, la ventana mínima en un 15% del ancho y alto del dispositivo.

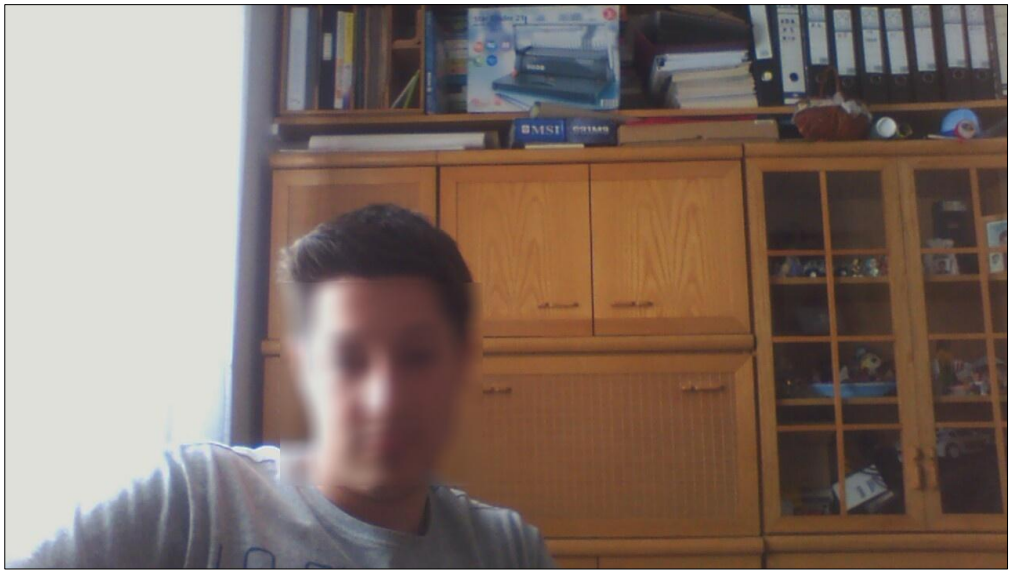
6.3.1. Resultados Aplicación Android y difuminado

Finalmente hemos obtenido una aplicación que nos permite hacer fotografías donde todas las caras serán difuminadas automáticamente. Para comprobar el nivel de satisfacción de la aplicación hemos tomado algunas imágenes aleatorias, y mostraremos algunos ejemplos, aunque no hemos generado ninguna métrica que nos permita tener un valor exacto de satisfacción.



EJEMPLO DE DIFUMINADO 1

Como vemos el clasificador ha detectado correctamente la cara y después se ha aplicado un difuminado que preservaría la identidad de la persona fotografiada.



EJEMPLO DE DIFUMINADO 2

7. Conclusiones

7.1. Programación temporal

En este apartado describiremos como hemos organizado las diferentes fases del proyecto en el tiempo y las posibles variaciones que se han producido.

El proyecto se inicia el 15 de septiembre de 2011 y se había planificado para terminar el 12 de marzo de 2012. Como describiremos dentro de este apartado durante el desarrollo del proyecto ha sido necesario añadir algunas fases nuevas y algunas otras se han alargado más de lo esperado.

Aun así, se han podido solventar la mayoría de problemas que nos hemos encontrado en la realización del proyecto.

El proyecto se había planificado para ser dividido en las siguientes fases:

1. Comprensión del algoritmo de Viola-Jones.
2. Desarrollo de un aprendizaje guiado, obteniendo un clasificador.
3. Fase de prueba de mi clasificador.
4. Adaptar la clasificación de objetos al Android.
5. Pruebas en dispositivos para comprobar la efectividad.

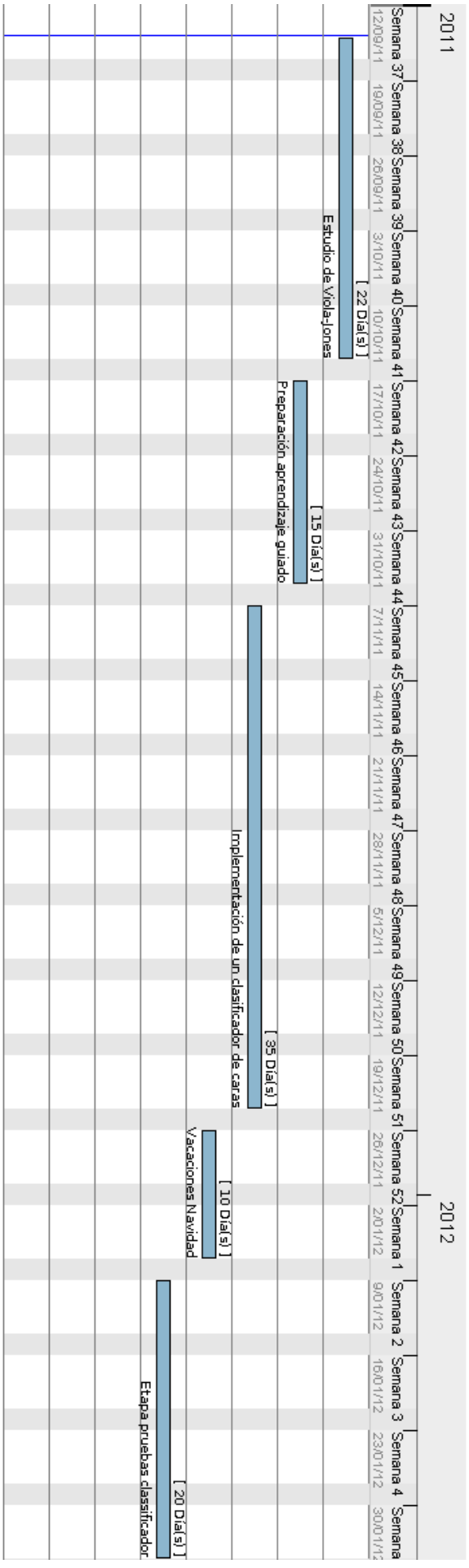
Pero finalmente, las fases desarrolladas han sido:

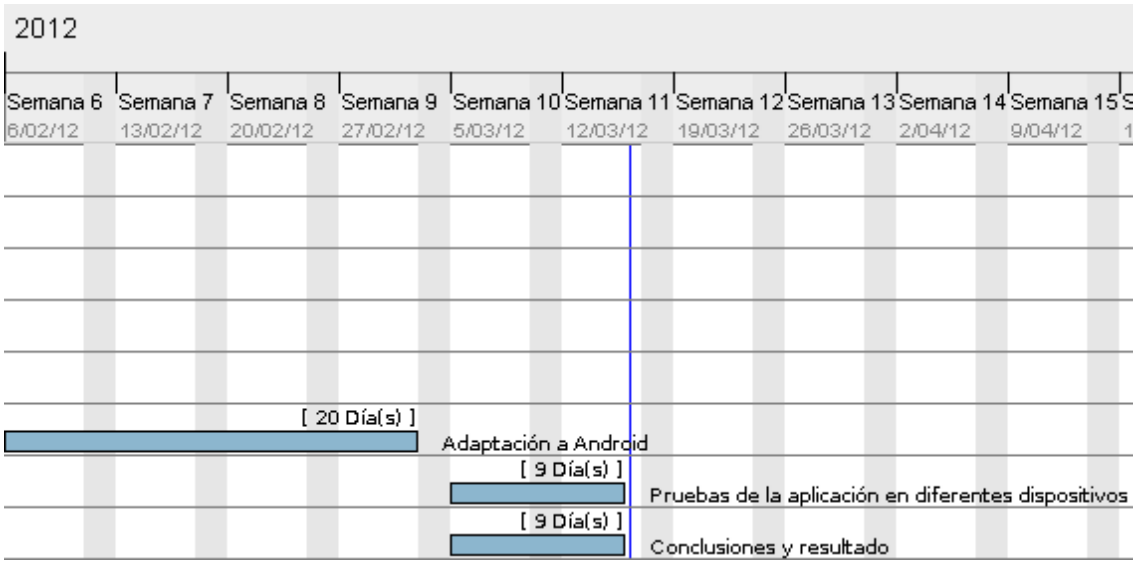
1. Estudiar el algoritmo de Viola-Jones y la librería OpenCV.
2. Crear un programa con un clasificador por defecto.
3. Implementar un clasificador propio.
4. Preparación de elementos para la segunda iteración.
5. Obtener un Segundo Clasificador.
6. Realizar una aplicación Android.
7. Difuminado de Caras en Android.

Las principales diferencias están en haber tenido que añadir una fase de creación de un segundo clasificador que consiguiera una mayor eficiencia que el primero y la aplicación final donde hemos aplicado toda la tecnología aprendida a un caso práctico.

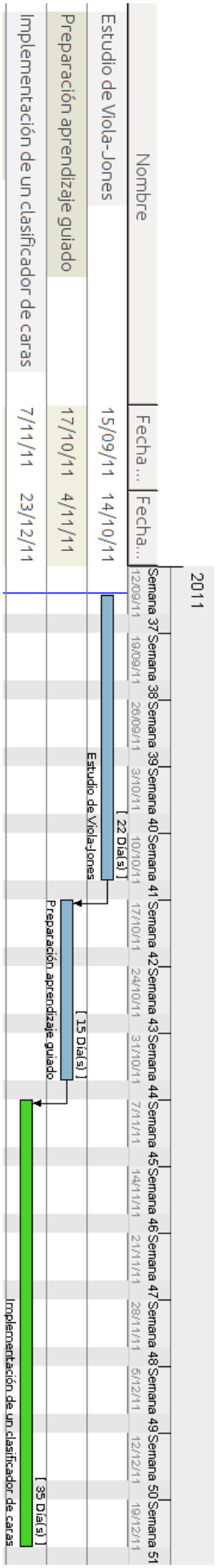
Además algunas tareas han llevado a la programación de software de gestión especifico, como es la etapa de preparación de elementos para la segunda iteración, que nos llevó al desarrollo del software SetResults que nos ocupó la mayoría del tiempo de la fase Preparación de elementos para la segunda iteración.

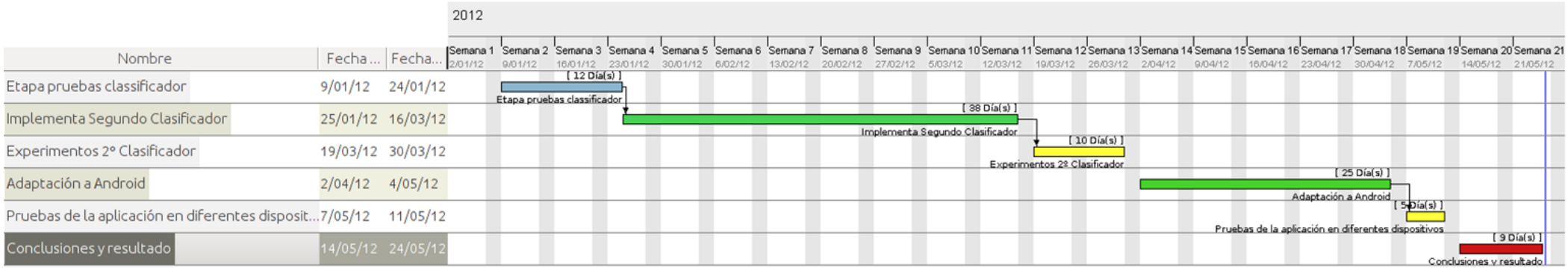
7.1.1. Planificación inicial general




2012
Semana 6
6/02/12
Semana 7
13/02/12
Semana 8
20/02/12
Semana 9
27/02/12
Semana 10
5/03/12
Semana 11
12/03/12
Semana 12
19/03/12
Semana 13
26/03/12
Semana 14
2/04/12
Semana 15
9/04/12
[20 Día(s)]
Adaptación a Android
[9 Día(s)]
Pruebas de la aplicación en diferentes dispositivos
[9 Día(s)]
Conclusiones y resultado

7.1.2. Planificación final general




2012
Nombre
Fecha ...
Fecha...
Semana 1 2/01/12
Semana 2 9/01/12
Semana 3 16/01/12
Semana 4 23/01/12
Semana 5 30/01/12
Semana 6 6/02/12
Semana 7 13/02/12
Semana 8 20/02/12
Semana 9 27/02/12
Semana 10 5/03/12
Semana 11 12/03/12
Semana 12 19/03/12
Semana 13 26/03/12
Semana 14 2/04/12
Semana 15 9/04/12
Semana 16 16/04/12
Semana 17 23/04/12
Semana 18 30/04/12
Semana 19 7/05/12
Semana 20 14/05/12
Semana 21 21/05/12
Etapa pruebas classificador 9/01/12 24/01/12
Implementa Segundo Clasificador 25/01/12 16/03/12
Experimentos 2º Clasificador 19/03/12 30/03/12
Adaptación a Android 2/04/12 4/05/12
Pruebas de la aplicación en diferentes disposit... 7/05/12 11/05/12
Conclusiones y resultado 14/05/12 24/05/12
[12 Día(s)]
Etapa pruebas classificador
[38 Día(s)]
Implementa Segundo Clasificador
[10 Día(s)]
Experimentos 2º Clasificador
[25 Día(s)]
Adaptación a Android
[5 Día(s)]
Pruebas de la aplicación en diferentes dispositivos
[9 Día(s)]
Conclusiones y resultado

7.2. Presupuesto

En este apartado están descritos los costes monetarios que van sujetos a nuestro proyecto y como las variaciones en la planificación han afectado en este.

7.2.1. Recursos Humanos

Los recursos humanos cubren las necesidades de personal que han realizado este proyecto. El número de desarrolladores ha sido de solo uno por lo que a diferencia de otros proyectos, todo el peso de diseño e implementación del software ha recaído sobre una única persona.

En una primera valoración se tuvo en cuenta que el proyecto se iba a desarrollar en 720 horas que si tenemos en cuenta que la valoración de la hora de ingeniero la fijamos en 40€ y la jornada es de 6 horas diarias teníamos una valoración inicial de **28880€.**

Debido a los cambios en la planificación que ha aumentado las horas del proyecto hasta las 960 horas que dejan un total de **38400€.**

7.2.2. Licencias y recursos de Software

Durante el proyecto hemos usado algunos recursos de software que nos han permitido realizarlo y que pueden tener un coste.

Software	Precio Licencia
QtCreator	0 €
Eclipse	0 €
OpenCV	0 €
Android	0 €

Como podemos ver todo el software y librerías usadas tienen licencias de código abierto lo que permiten ser usadas para el desarrollo de software sin coste alguno aunque su intención sea para desarrollar software privativo.

7.2.3. Amortización de hardware

En el desarrollo del proyecto se han usado dos dispositivos de hardware. En primer lugar se ha utilizado un portátil HP Pavilion dv6-6070es y para la fase de pruebas de la aplicación Android, se ha usado un Samsung Galaxy SII.

Para calcular el costo de amortización del portátil, tendremos en cuenta que la vida útil de un portátil es de aproximadamente 3 años, podemos determinar que el precio por día es de 800 € (precio de compra) entre 365 días*3 lo que hace un total de **116,89 €** de uso del portátil por 960 horas de uso.

En cuanto al teléfono móvil no se ha usado directamente en el desarrollo, si no solo para realizar algunas pruebas de rendimiento, es por esto que podemos considerar su coste de amortización despreciable.

7.2.4. Total

En conclusión podemos decir que el coste total del proyecto es de **38.516,89 €**

7.3. Objetivos Conseguidos

Al empezar este proyecto nos habíamos propuesto unos objetivos que se han ido cumpliendo mientras se iba desarrollando cada una de las fases. En una valoración global del proyecto podemos decir que se han cumplido todos los objetivos aunque con algunas variaciones respecto a lo esperado.

En primer lugar ha habido desviaciones de tiempo en la previsión inicial como hemos detallado en el apartado de programación temporal. Esto ha provocado que el presupuesto también se viera afectado. En cuanto a los objetivos concretos que nos planteamos al inicio, se han cumplido correctamente aunque en algunos de ellos sea difícil valorar el nivel de satisfacción en cuanto al resultado.

7.3.1. Detalle

7.3.1.1. Estudiar el algoritmo de Viola-Jones y la librería OpenCV

En esta fase el objetivo principal era entender las bases teóricas que estaban detrás del proyecto que quería desarrollar y poder utilizar una implementación del algoritmo de Viola-Jones que se encuentra dentro de la librería de visión OpenCV.

En cuanto al estudio teórico fue un poco complicado encontrar mucha información sobre ello, principalmente se ha usado el paper [16]que explica el procedimiento a utilizado pero existe muy poca bibliografía escrita respecto a estas técnicas.

Gracias que la librería OpenCV dispone de una licencia de código libre, no fue difícil encontrar una comunidad trabajando con ellas. Aunque la tecnología usada esté en el state-of-the-art [17] y no es muy popular.

De todas formas puede encontrar información que me permitió obtener los conocimientos, pero además, y a mi parecer más importante, he encontrado una comunidad y algunos grupos donde muchos desarrolladores de todo el mundo comparten experiencias y proyectos en el campo de la visión por computador.

[16] Artículo Científico

[17] Hace referencia al nivel más alto de desarrollo conseguido en un momento determinado sobre cualquier aparato, técnica o campo científico plural.

7.3.1.2. Crear un programa con un clasificador por defecto

Si nos referimos a la creación de software con la librería OpenCV hemos desarrollado dos programas que realizan la detección de caras y que permiten seleccionar entre diferentes clasificadores. Los dos programas que hemos llamada detectfaces y faceDetect iban enfocados a probar 2 variantes en la detección.

Con detectfaces tenemos un programa que obtiene imágenes desde una webcam y permite la detección en tiempo real de caras frontales.

En segundo lugar, faceDetect era capaz de analizar carpetas donde contenemos imágenes y crear un fichero donde se guarda la ruta de la imagen y los objetos detectados. Estos ficheros han sido usados en fases posteriores.

Para concluir podemos determinar que los objetivos de esta fase se han cumplido con creces y que además hemos desarrollado satisfactoriamente las bases para poder cumplir con éxito las etapas siguientes de nuestro proyecto.

7.3.1.3. Implementar un clasificador propio

Esta sin duda fue la etapa que más variaciones ha provocado en la planificación del proyecto. Este fue sin duda el objetivo más complejo, en parte por la cantidad de subtareas que se podían derivar y de su complejidad.

En primer lugar, debíamos obtener las muestras tanto positivas como negativas para nuestro entrenamiento. Como todo el procedimiento está en un estado experimental fue un poco difícil encontrar las muestras, aunque finalmente, buscando detenidamente, algunas facultades dejan descargar bajo demanda sus bases de datos de caras y pude obtener las que necesitaba.

Segundo, se necesitaba introducir muestras positivas, en las que definíamos mediante un fichero donde se encontraban las caras en cada una de las imágenes. Esto no podíamos hacerlo de manera automática, ya que ese es el objetivo de hacer un entrenamiento guiado. Por esto tuvimos que desarrollar un software que hemos llamado **ObjectMarker** que permitía marcar imagen por imagen cada una de las caras de nuestro entrenamiento hasta un total bastante abultado.

Por último, realizar el entrenamiento nos llevó más tiempo del que esperaba, ya que tardó 4 días enteros en generarse nuestro clasificador, tiempo que tampoco había tenido en cuenta al iniciar el proyecto.

Aunque con algunos problemas conseguimos generar con éxito un clasificador aunque como hemos explicado en el apartado de resultados no conseguimos los resultado esperados.

Es por esto que aunque cumplimos el objetivo, decidimos por medio de la técnica del bootstraping[18] desarrollar un segundo clasificador más eficiente.

[18] Hace referencia a la técnica que consiste en utilizar el resultado de un proceso como entrada de otro. http://es.wikipedia.org/wiki/Bootstrapping_%28inform%C3%A1tica%29

7.3.1.4. Preparación de elementos para la segunda iteración

Debido a los resultados obtenidos en la primera iteración debemos preparar las imágenes para la segunda, en la que debemos introducir las caras mal detectadas como muestras positivas y los falsos positivos como muestras negativas para que no fueran detectadas.

Aunque este objetivo se ha cumplido satisfactoriamente, provocó que hubiera que crear un nuevo software de gestión.

Este nuevo software que hemos llamado **SetResults** nos ha permitido partiendo de imágenes ya detectadas y el fichero que arroja **DetectFaces** generar un fichero resultante de una manera ágil. Aunque el desarrollo de este software también ha supuesto un tiempo inesperado en la consecución del proyecto.

El objetivo se ha cumplido ya que pudimos obtener los ficheros descriptivos y las imágenes que nos hacían falta para la siguiente iteración.

7.3.1.5. Obtener un Segundo Clasificador

El objetivo de generar un segundo clasificador más eficiente se ha podido cumplir satisfactoriamente aunque con matices. Como hemos descrito en el apartado de resultados nuestro clasificador obtiene un índice de aciertos del **87%** lo que nos permite decir que el clasificador ha mejorado bastante respecto a la primera iteración.

Aun así, debemos tener en cuenta que un aumento en la detección, provocó también un aumento de falsos positivos en imágenes muy grandes.

Esto puede ser un problema si queremos detectar caras muy pequeñas en imágenes con una resolución muy elevada, pero en nuestro caso hemos hecho una precondiciones, determinar un tamaño mínimo de detección del 20% del ancho y alto de la imagen, en la etapa Android para que la detección sea ágil que nos permite evitar también este problema ya que el tamaño mínimo nos filtrara esos falsos positivos.

Como último apunte, podemos determinar que se pueden superar estos problemas realizando una tercera iteración realizando el mismo procedimiento que hemos desarrollado para la segunda iteración. Aunque hemos decidido no hacerlo porque creemos que los resultados son aceptables para nuestros propósitos, una tercera iteración permitiría acabar de pulir los resultados obtenidos y conseguir un clasificador que sea capaz de descartar mejor los falsos positivos.

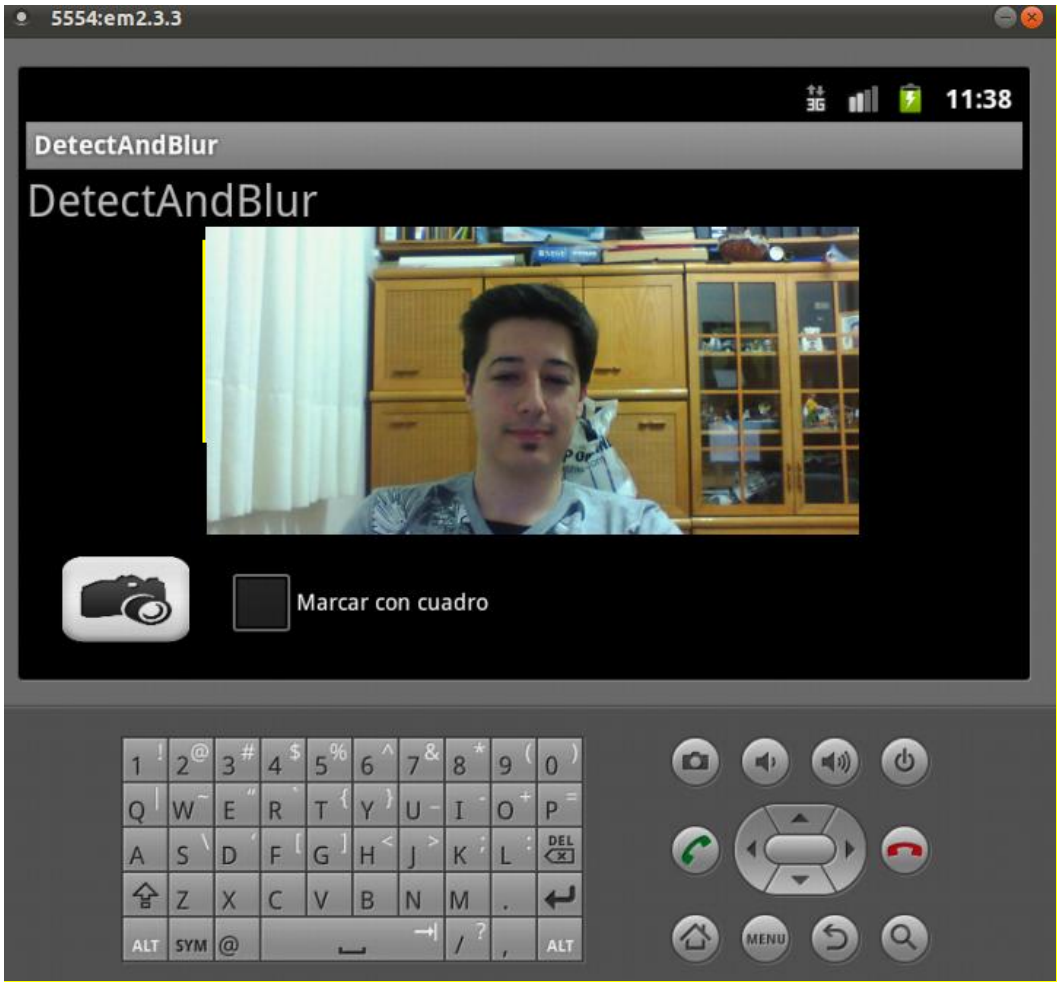
7.3.1.6. Realizar una aplicación Android

Desarrollar la aplicación Android ha sido la parte más gratificante de este proyecto. En las etapas anterior hemos creado las bases para entender cómo funciona el algoritmo de Viola-Jones y poder desarrollar nuestro propio clasificador, pero es en esta etapa donde realmente se aplica todo este conocimiento y en una tecnología que ahora mismo es puntera en cuanto a desarrolladores trabajando en ella.

El desarrollo en Android en este momento ha sido relativamente factible gracias a la gran comunidad que hay trabajando en ella. La dificultad principal radicaba en poder usar la librería OpenCV en su versión para Android.

Esta librería fue lanzada en su primera versión en 2010 para ser compatible con la versión 2.2 de Android. Con solo 2 años en el desarrollo de la librería aún hay poca documentación al respecto lo que ha hecho un poco difícil la programación en esa parte, que por otro lado era la principal de nuestro proyecto.

Aun así, hemos podido crear una aplicación que nos permite detectar caras en una imagen captada con la cámara de un dispositivo Android. Además se muestra el resultado en pantalla marcando la cara detectada con un recuadro.



Esta etapa ha cumplido las expectativas, aunque hemos tenido que hacer algunas concesiones para que el rendimiento de la aplicación fuera aceptable. En primer lugar hemos tenido que fijar el tamaño mínimo en un 15% de la imagen, porque si desarrollamos una clasificación con subimagenes muy pequeñas la aplicación tarda mucho y no es práctico.

Además que podemos suponer que cuando queremos detectar las caras de las personas en una fotografía, estas se encontraran en el primer plano de la imagen.

Con esto hemos obtenido una aplicación que es capaz de procesar las imágenes en menos de 2 segundos lo que hace que sea muy dinámica.

Por último debemos decir que de este objetivo se ha derivado uno nuevo que no habíamos contemplado en las especificaciones iniciales. Una vez tenemos desarrolla nuestra aplicación y que hemos hecho algunos test de su uso, se ha pensado en enfocar la aplicación en un aspecto práctico. Es así como hemos desarrollado un último objetivo de desarrollo de un difuminado automático de caras.

7.3.1.7. Difuminado de Caras en Android

Este último objetivo se creó para darle una vertiente práctica al proyecto. El objetivo principal fue tratar la subimagen detectada con un difuminado y entonces obtener la imagen resultante con las caras detectadas difuminadas.

Este objetivo se ha cumplido también, culminando el proyecto pero hemos tenido que hacer una precondición a la hora de desarrollar el difuminado (*blurring*). En las características del difuminado se encuentra la de definir cuál será el tamaño del píxel tanto en ancho como en alto.

Si se utiliza un tamaño de píxel más pequeño se consigue un difuminado más suave, y si por el contrario se usa un difuminado con un tamaño de píxel más grande se hace un difuminado más tosco, más parecido a un pixelado. En nuestro caso hemos determinado que el tamaño de píxel será del 10% del ancho y 10% del alto.

Con estos valores hemos conseguido que el difuminado no suponga un coste muy elevado en la ejecución del programa y además, para el nuestra aplicación, un difuminado tosco no es ningún inconveniente ya que queremos ocultar las caras detectadas. Y además, el hecho de usar porcentajes hace que la aplicación sea más escalable para otros dispositivos donde la resolución de la cámara sea diferente.

7.4. Trabajos futuros

En esta sección describiremos las posibles tareas que se podrían desarrollar partiendo del trabajo realizado en este proyecto. En la parte de preparación y la generación del clasficador han quedado algunos flecos que se podrían mejorar para hacer que la solución fuera más robusta.

En cuanto a la parte del desarrollo en Android, en la que se usa una tecnología más puntera, las posibilidades son muchas. El desarrollo para plataformas móviles es uno de los activos y combinado con la visión por computador puede suponer la integración de esta tecnología en la vida cotidiana.

7.4.1. Fase PC

7.4.1.1. Mejora en el clasificador

Como hemos visto, nuestro clasificador de caras tiene porcentaje de acierto bastante alto, pero cambio arrojaba demasiados falsos positivos. Una posible mejora seria realizar una tercera iteración.

Con un nuevo procesado de las imágenes donde introduciremos los falsos positivos como muestras negativas para que el clasificador las descarte. Aunque el índice de detección no mejore recibiríamos unos resultados más limpios, y no necesitaríamos hacer preprocesados en el resultado para asegurarnos que los que estamos recibiendo es correcto.

7.4.1.2. Mejoras en SetResults

SetResults nos ha permitido procesar los resultados de una detección para poder hacer una segunda, pero aun así tiene algunas característica que se pueden mejorar.

En primer lugar, la decisión de si una muestra es correcta o incorrecta se realiza mediante ratón, apretando una serie de botones. Para que la inspección sea más rápida, se pueden detectar pulsaciones de teclas del teclado que permitan de manera ágil clasificar las muestras obtenidas.

Además se podría añadir la posibilidad de guardar el contexto de un procesado en cualquier momento y continuarlo más tarde. Así no se tiene que procesar un fichero de resultados desde principio a fin sin poder apagar el programa, y podríamos pasar a analizar otros ficheros sin descartar el trabajo realizado hasta ese momento.

Por último, y para mi uno de los cambios más importantes sería un posible integración de SetResults con ObjectMarker. Integrar la posibilidad de marcar como muestras positivas algunas imágenes vacías y poder obtener un fichero con esas muestras.

De esta manera dispondríamos de una suite integrada para el procesado de ficheros de muestras tanto para obtener las muestras positivas como las negativas para la siguiente iteración de mejora de nuestro clasificador.

7.4.2. Mejoras Android

En cuanto a la parte que afecta al desarrollo en Android es la que tiene más potencial. Dentro del diseño de la aplicación el principal hilo de desarrollo ha sido que la aplicación fuera capaz de realizar los casos de uso.

7.4.2.1. Diseño de la interfaz

Es por esto que la interfaz de es la más sencilla que se podía hacer. Un primer paso sería la mejora del diseño gráfico de la aplicación que ha quedado un poco fuera de los objetivos del proyecto por no considerarlo prioritario, ya que la aplicación no se iba a publicar, ni tenía que formar parte de ningún proyecto comercial.

7.4.2.2. Aplicación tiempo real

Nuestra aplicación Android necesita de la interacción de alguna persona para realizar el análisis de las imágenes, es necesario que alguien apriete un botón para empezar el proceso.

Es por esto que otras de las mejoras que podrían representar un salto de calidad podría ser que todo el análisis se realizara en tiempo real. Esto podría representar problemas de rendimiento en función de cada dispositivo pero con el crecimiento en las prestaciones que están sufriendo los smartphones de gama alta y media sería muy interesante disponer de esta característica.

Además, que el análisis se produzca en tiempo real hace que todo sea mucho más interactivo, y la forma de uso de la aplicación sea mucho más natural.

7.4.3. Futuras Aplicaciones

La tecnología estudiada puede usarse en múltiples aplicaciones que con un poco de desarrollo podrían incluso materializarse en aplicaciones comerciales. En esta sección comentaremos algunas ideas que podrían convertirse en proyectos.

7.4.3.1. Fase previa a reconocimiento

La principal utilidad de una aplicación de detección de caras es el de poder usarlo como la base para un sistema de reconocimiento facial. Antes de poder usar algoritmos que nos permitan reconocer si una cara corresponde a alguien debemos encontrarla en la imagen.

Es por esto que un sistema de detección de caras es perfecto para buscar la localización de los rostros en una imagen antes de aplicar un reconocimiento facial.

7.4.3.2. Seguridad

Otra posible aplicación de un sistema así sería la utilización dentro de un sistema de seguridad. Imaginemos que queremos detectar todas las personas que pasan por una puerta y registrar sus cara en el momento que pasan un control de acceso.

En el momento que las personas pasaran por ejemplo un control con tarjeta podría registrarse la cara de la persona, de esta manera tendríamos registrados las personas que usaron esa tarjeta y quedaría como una prueba en un caso de un uso indebido del acceso.

Otro posible uso dentro de la seguridad sería la de detectar las personas que se encuentran en un sitio o en un caso de robo poder tener un registro de caras y no tener que revisar toda la grabación.

7.4.3.3. Contar personas

Un sistema de detección de caras puede tener también grandes aplicaciones en el conteo de personas en eventos donde se mire fijamente un punto como puede ser una pantalla o un escenario.

También se podría usar en las cuentas en manifestaciones o grandes aglomeraciones donde se suelen utilizar las cifras de asistencia como un dato importante en la prensa.

7.4.3.4. Reconocimiento de Objetos para aprendizaje

Por último, me gustaría destacar que este sistema no se usa solamente en el reconocimiento de caras, realizar un aprendizaje guiado puede hacerse para reconocer cualquier objeto siempre que se introduzcan las muestras adecuadas.

De esta manera un sistema de detección de patrones puede usarse en un entorno industrial para cualquier sistema de control, gestionado mediante visión por computador.

Un ejemplo de esto sería una empresa donde tengamos que ordenar un varios productos, y en función de estos realizar una acción u otra.

Imaginemos una situación donde un brazo robótico deba recoger algún objeto y posicionarlo en una bandeja diferente en función del objeto. Con un sistema de reconocimiento por patrones podríamos reconocer donde se encuentran los puntos de apoyo y que tipo de objeto es para actuar en consecuencia.

8. Referencias

http://opencv.willowgarage.com/wiki/ - Wiki de Opencv

http://www.vision.caltech.edu/html-files/EE148-2005-Spring/pprs/viola04ijcv.pdf – Paper Viola-Jones

http://cmp.felk.cvut.cz/~sochmj1/adaboost_talk.pdf – Adaboost

http://cseweb.ucsd.edu/~yfreund/adaboost/index.html – Pagina para entender cómo funciona Adaboost

http://www.cmake.org/ – Página principal de CMAKE

http://www.cs.swarthmore.edu/~adanner/tips/cmake.php – Tutorial de Cmake

http://note.sonots.com/SciSoftware/haartraining.html – Web de ayuda en la realización del Haartraining

http://blog.damiles.com/2010/09/opencv-and-cmake/ - Integración de Cmake con OpenCV

http://opencv.willowgarage.com/wiki/CompileOpenCVUsingLinux – Compilar software de la librería en Linux

https://groups.google.com/forum/?fromgroups#!forum/android-opencv - grupo de Google Visión por Computador

http://tech.groups.yahoo.com/group/OpenCV/files/ - Grupo visión de Yahoo (Open Source Computer Vision Library Comm)

http://dev.kofee.org/projects/qtcreator-doxygen/wiki – Integración Qtcreator con Doxygen

http://linux301.wordpress.com/2009/06/11/qt-creator-svn-function/ - Qtcreator con Subversion

http://www.oracle.com/technetwork/java/javase/downloads/index.html – Pagina de descarga de Java

http://www.eclipse.org/downloads/ - Página de descarga de Eclipse

Bases de datos de caras

http://www.face-rec.org/databases/ - Compilación de webs con bases de datos

http://gps-tsc.upc.es/GTAV/ResearchAreas/UPCFaceDatabase/GTAVFaceDatabase.htm - GTAV Face Database

http://www.cs.cmu.edu/afs/cs/project/vision/vasc/idb/www/html_permanent//index.html – Carnegie Mellon Image Database

http://cswww.essex.ac.uk/mv/otherprojects.html - Dr Libor Spacek, Computer Vision Research Projects

http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html – AT&T Laboratories Cambridge

OpenCV

http://opencv.willowgarage.com/documentation/cpp/index.html – Wiki OpenCV

http://www.slideshare.net/devaldiviesoxxx/estimacin-de-proyectos-de-software – Estimación de presupuestos de proyectos de software

Android

http://developer.android.com/sdk/index.html – Página de descarga SDK de Android

http://opencv.itseez.com/trunk/doc/tutorials/introduction/android_binary_package/android_binary_package.html#manual-environment-setup-for-android-development – Tutorial configuración entorno de desarrollo Android+Eclipse

http://www.android.es/android-ui-utilities-crea-tu-propia-interfaz.html#axzz1dncHJWak – UI Utilities, ayuda para diseñar interfaces Android

http://opencv.itseez.com/doc/tutorials/introduction/android_binary_package/android_binary_package.html – Usando Android con Eclipse

http://www.amazon.com/Pro-Android-3-Satya-Komatineni/dp/1430232226/ref=sr_1_15?s=books&ie=UTF8&qid=1339455781&sr=1-15&keywords=android – Libro para el desarrollo en Android

http://www.tomgibara.com/android/camera-source – Ejemplo de uso de la cámara en Android

http://developer.android.com/resources/dashboard/platform-versions.html – Web oficial de google de Android Developers

ANEXO 1: Preparación entorno PC

En este anexo se explicará paso a paso como se debe preparar el entorno para el desarrollo de las pruebas con la librería OpenCV y C++.

Crear Proyecto Cmake

Cmake

En la primera etapa diseñaremos un detector de caras fuera de una plataforma android. Usaremos la librería OpenCV de la misma manera que haremos en android pero utilizaremos código compilado con un compilador GNU/C++ en Ubuntu 11.04.

Para crear una estructura que nos permita trabajar con facilidad, y no debamos preocuparnos en como enlazar las librerias o definir donde están los ejecutables usaremos Cmake. Cmake genera makefiles and espacios de trabajo que crean un entorno para compilar a tu elección.

En primer lugar, Cmake se basa en unos ficheros CmakeLists.txt que definen el proyecto.

```
cmake_minimum_required(VERSION 2.6)
project(OpencvTest)

#There are lots of scripts with cmake
#for finding external libraries.
#see /usr/local/share/cmake-2.6/Modules/Find*.cmake for more examples
FIND_PACKAGE( OpenCV REQUIRED )

add_subdirectory(src)
```

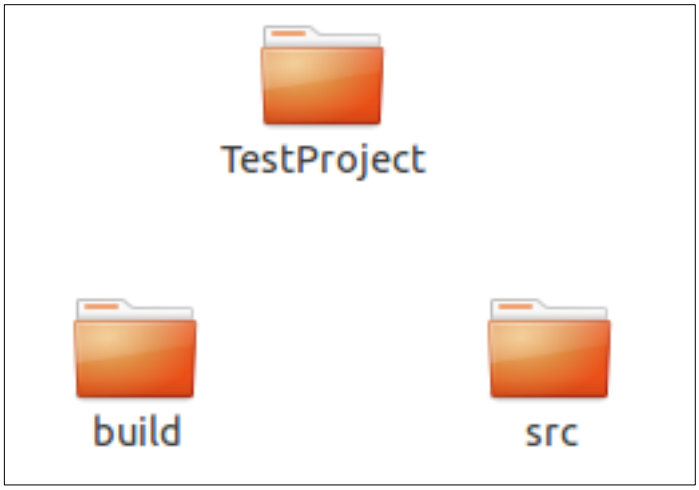
Procedimiento de Creación de Proyecto con Cmake y OpenCV

1. **Creación estructura (carpetas)**
2. **Definición del proyecto en CmakeLists.txt**
3. **Creación de CmakeLists.txt para compilar ejecutables**
4. **Compilación**

Todo el procedimiento se llevará a cabo mediante bash (terminal por defecto de ubuntu) para centralizar el tutorial, pero puede hacerse de manera gráfica si se desea.

Creación estructura (carpetas)

En primer lugar creamos una carpeta para el Proyecto y dentro de esta, otra para el código compilado y una última para nuestros ficheros origen.



Definición del proyecto en CmakeLists.txt

En la carpeta raiz del proyecto generamos un fichero CmakeLists.txt.

```
cmake_minimum_required(VERSION 2.6)
project(OpencvTest)

#There are lots of scripts with cmake
#for finding external libraries.
#see /usr/local/share/cmake-2.6/Modules/Find*.cmake for more examples
FIND_PACKAGE( OpenCV REQUIRED )

add_subdirectory(src)
```

Ejemplo de fichero cmakeLists.txt

Lo principal consiste en un fichero de definición, donde con directivas previas podemos definir la estructura de ficheros del proyecto, el nombre o si tiene alguna dependencia importante.

Algunas directivas usadas:

cmake_minimum_required Usado para eigir una versión concreta de cmake.

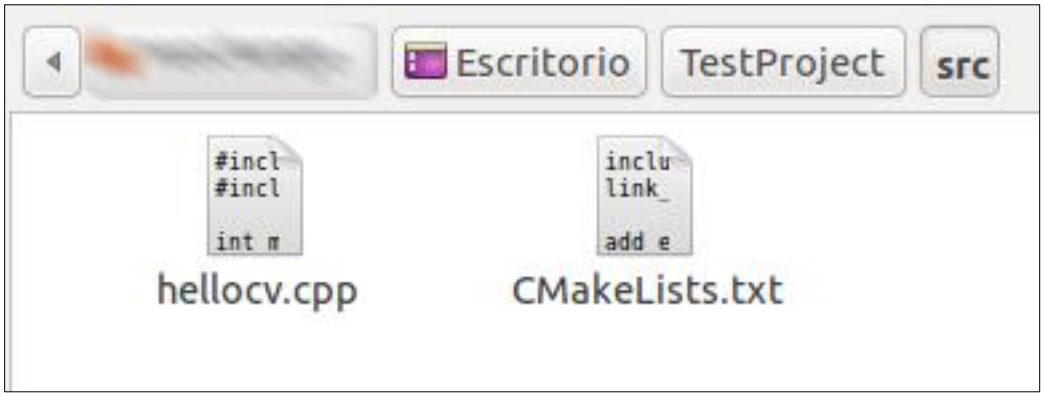
project(name) Define el nombre del proyecto.

FIND_PACKAGE (OpenCV REQUIRED) Indicamos que el programa no puede ser compilado si no tenemos opencv instalado en nuestro equipo.

add_subdirectory (src) Definimos que vamos a usar un subdirectorio para el proyecto.

Creación de CmakeLists.txt para compilar ejecutables

Ahora introducimos nuestro ejecutable en la carpeta src, o el nombre que hayamos elegido para la carpeta que contendrá nuestro código fuente.



Vemos como tenemos el fichero origen de nuestro código y un nuevo fichero CmakeLists.txt que contendrá los parametros que permiten compilar nuestro ejecutable.

```
include_directories(${CMAKEDEMO_SOURCE_DIR})
link_directories(${CMAKEDEMO_BINARY_DIR})

add_executable(hellocv hellocv.cpp)

TARGET_LINK_LIBRARIES( hellocv ${OpenCV_LIBS} )
```

include_Directories Incluye el directorio indicado para coger librerias de él

link_directories Incluye el directorio deseado en la etapa de linkado en la compilación.

add_executable Es el parametro más importante, y indica que queremos generar un ejecutable, con nombre hellocv y que el fichero de inicio será hello.cpp.

TARGET_LINK_LIBRARIES Da visión al ejecutable sobre donde se encuentran las librerias que le introducimos.

Compilación

Por último debemos dirigirnos a la carpeta build creada anteriormente en la raiz del proyecto y configurar nuestro proyecto. Para ello ejecutaremos en un terminal:

```
neochange@neochange-HP-Pavilion-dv6-Notebook-PC:~/Escritorio/TestProject/build$ cmake ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/neochange/Escritorio/TestProject/build
neochange@neochange-HP-Pavilion-dv6-Notebook-PC:~/Escritorio/TestProject/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile  src
```

Vemos como dentro de la carpeta build se han creado una serie de carpetas autogeneradas, que se han creado MakeFiles y que se han detectado las librerias necesarias para el proyecto.

Por último entramos en la carpeta src y hacemos make.

```
Scanning dependencies of target hellocv
[100%] Building CXX object src/CMakeFiles/hellocv.dir/hellocv.cpp.o
Linking CXX executable hellocv
[100%] Built target hellocv
```

IDE (Integrated Development Environment)

Para poder trabajar mejor en un entorno de desarrollo es muy importante usar programas que agilizan la manera de trabajar con autocompletado de código y aviso de error en el código sin necesidad de compilar.

En la primera etapa de mi proyecto está programa en C++ por lo que usaré un IDE versátil que me permita compilar sin necesidad de usar el terminal y que además tenga soporte para Cmake.

El único que he podido encontrar que cumplía estos requisitos era Qtcreator, el IDE oficial de las Qt de Nokia. Además tiene soporte para Subversion (que me permite tener mi código en línea y llevar un control de versiones) y otros controladores de versiones, y también soporta Doxygen para generar una documentación automática.

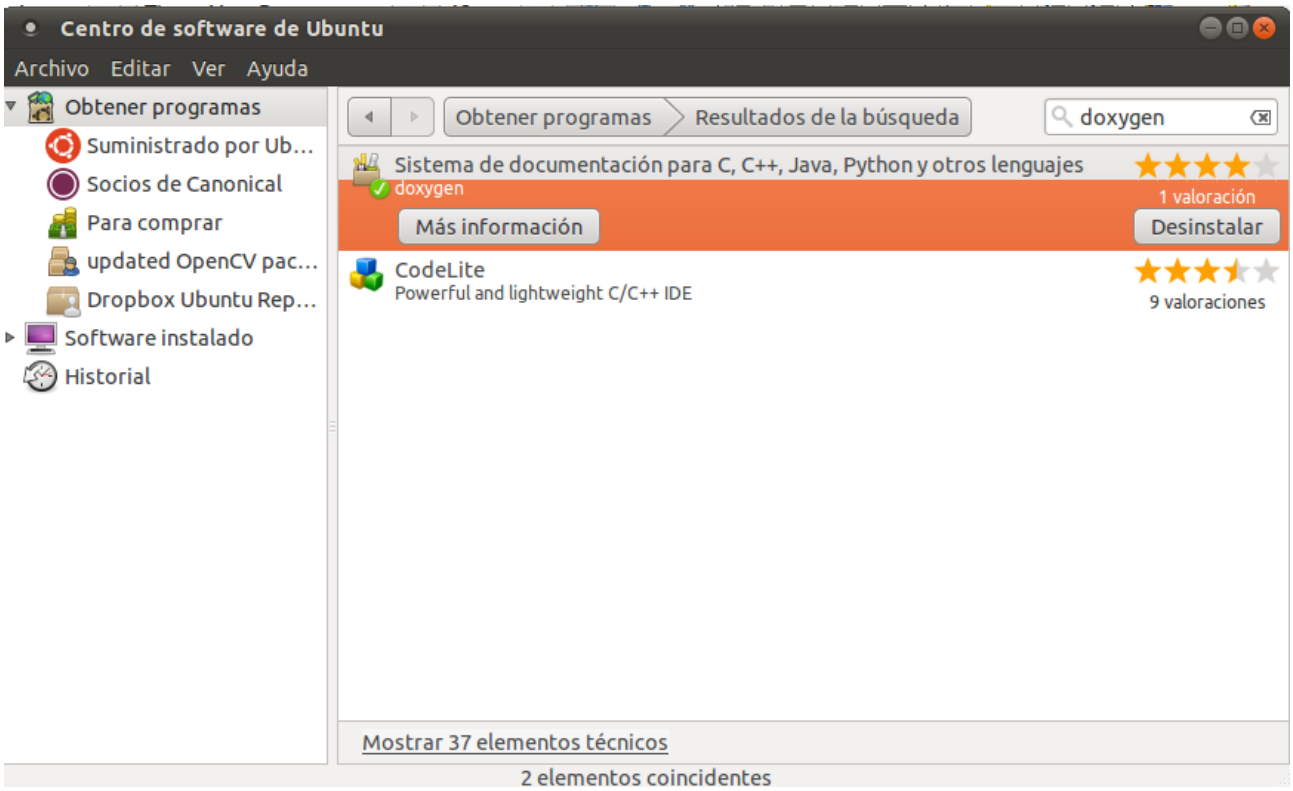
Instalación de Doxygen

Doxygen es un programa que genera documentación a través de notas dejadas en el código. Es muy útil y se pueden encontrar multitud de librerías muy utilizadas, o diseñadas por grandes compañías documentadas de esta manera.

Para usar Doxygen en Qtcreator en primer lugar tenemos que instalarlo en nuestro sistema. En mi caso he usado un único comando:

Sudo apt-get install Doxygen

O podemos buscarlo en el centro de software de Ubuntu y instalarlo.



Una vez instalado debemos ir a la página de Doxygen y descargar el plugin de Qtcreator para nuestra versión de Qtcreator, en el momento de realizar esta documentación la versión 2.3.0.

Lo descomprimimos en /carpeta_qtcreator/lib/qtcreator/plugins y al arrancar Qtcreator dispondremos de un nuevo menú para generar la documentación.

Subversion en Qtcreator

En Ubuntu el soporte para SVN está incluido por que no debemos instalar nada. En sus versiones recientes Qtcreator tiene incorporado el soporte. Sólo se debe abrir un proyecto que ya esté sujeto a control de versiones.

Para importar un proyecto a un repositorio sólo debemos hacer:

Y después ya podemos borrar los datos de la carpeta y con el comando checkout descargar el contenido de que hay en el repositorio.

ANEXO 2: Preparación entorno Android

En el siguiente anexo se describe la instalación del entorno en la etapa Android. Todo el desarrollo al igual que en las etapas anteriores se llevará a cabo sobre un portátil con el sistema operativo Ubuntu.

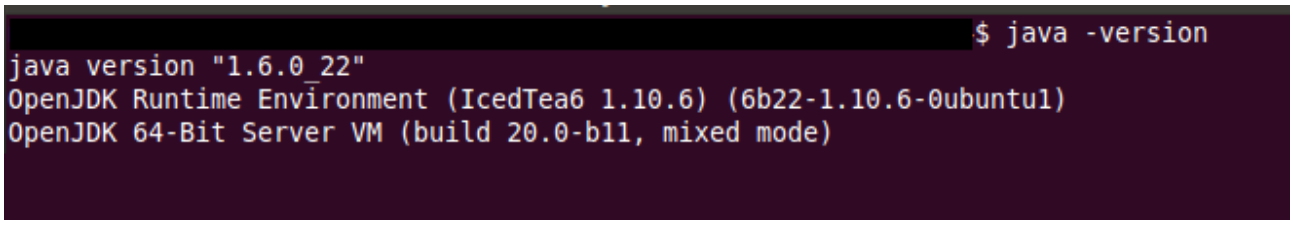
Instalación

Para poder usar Android en nuestro sistema operativo Ubuntu necesitamos:

- Java a partir de la versión 1.6 (JDK 6)
- IDE Eclipse
- SDK de Android
- Plugin de Android para Eclipse

Java

La versión de java que necesitamos es a partir de la 1.6. Para comprobar si la tenemos instalada, abrimos un terminal y apretamos "**java -version**"



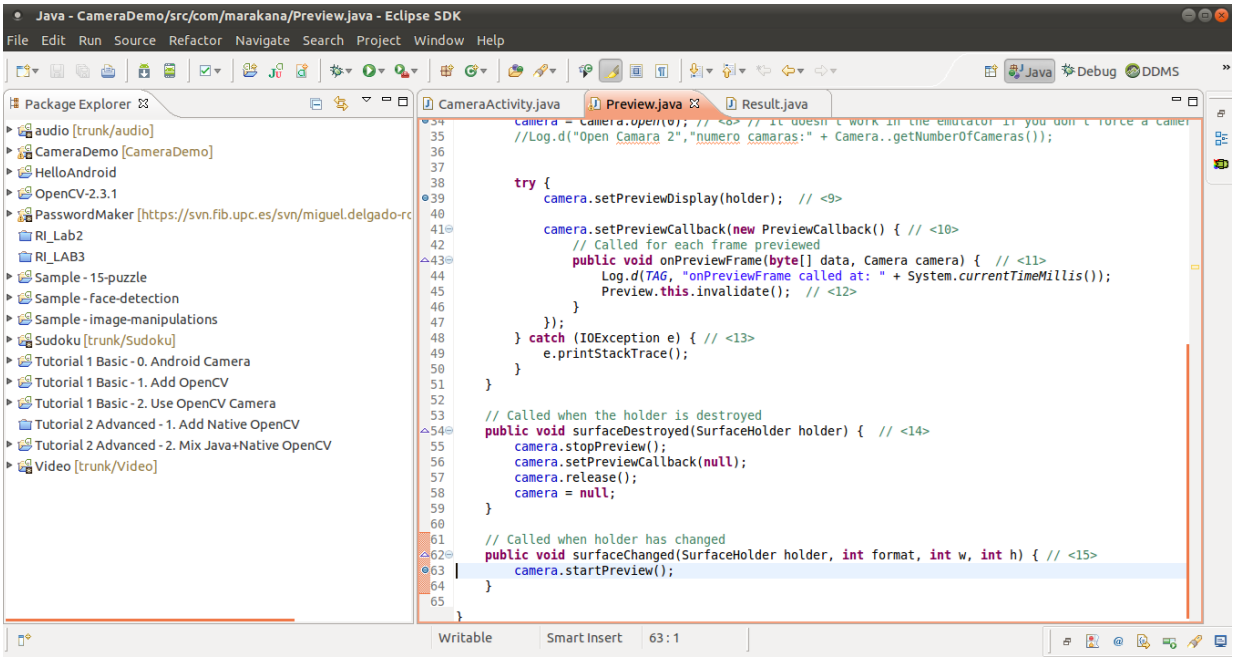
Si recibimos algo parecido a lo anterior ya tenemos lo que necesitamos. En caso de no aparecer nuestra versión debemos instalar el JDK 6 que podemos obtener en la pagina de Java. (http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u4-downloads-1591156.html)

Eclipse IDE

El IDE [19](Entorno de desarrollo integrado) nos permitirá disponer de un entorno donde realizar el desarrollo de manera cómoda y mucho más eficiente. Además Eclipse está totalmente integrado con la librería Android gracias a sus plugin de instalación.

[19] ***http://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado***

Debemos decargar la versión Eclipse Classic que en el momento de realizar este documento es la 3.7.2.[20]



SDK de Android

Para poder compilar con Android necesitamos las librerías de cada una de las versiones. Pero a diferencia de otras plataformas el SDK de Android no está compactado en un único paquete de instalación.

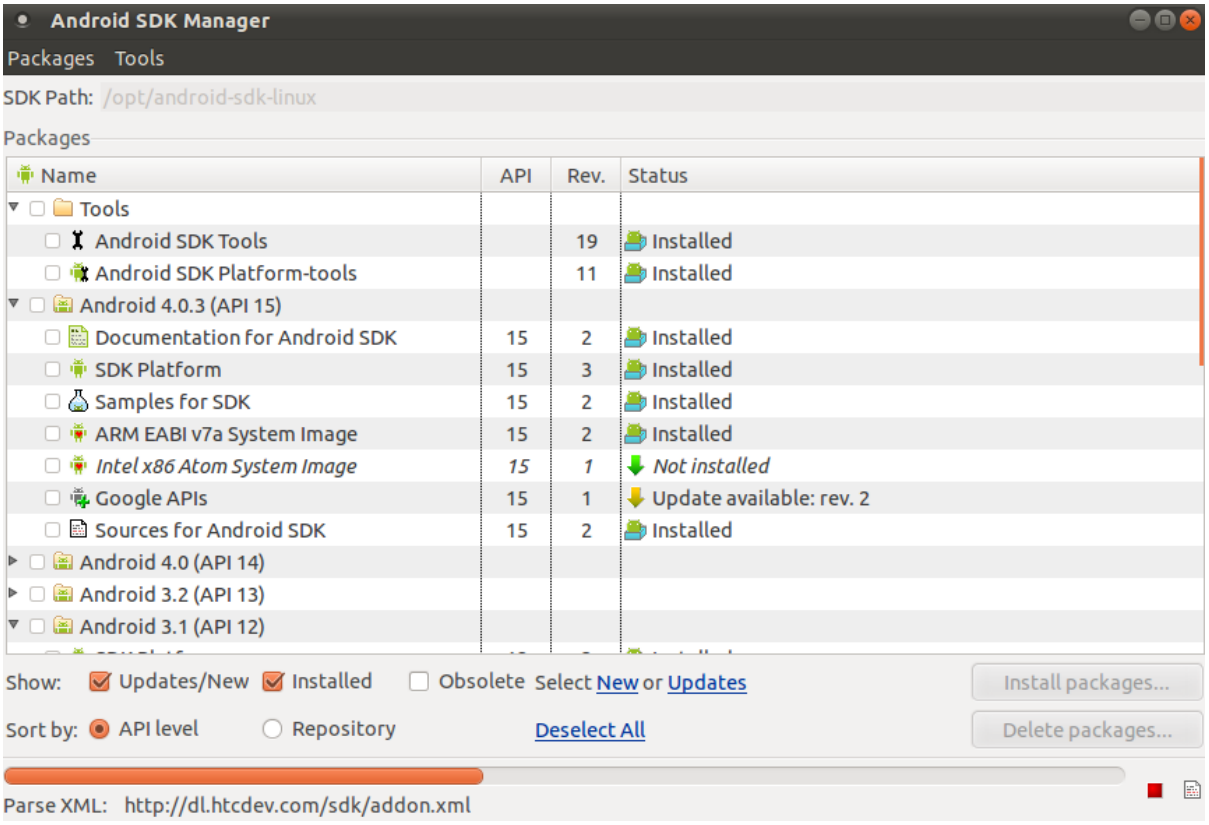
El SDK de Android se divide entre el Android SDK Starter Package y el Android SDK Components. Con el Starter Package disponemos de los paquetes principales para poder instalar el resto que necesitamos.

El paquete del Starter Package lo podemos descargar desde la web de desarrolladores Android. http://developer.android.com/sdk/index.html) y no necesitamos de realizar ninguna instalación singular. Debemos descargar el paquete para nuestro sistema operativo y desempaquetar en alguna carpeta persistente.

Para poder disponer de las herramientas que se disponen es recomendable añadir la carpeta donde se encuentren los ficheros en la variable PATH del sistema.

[20] ***http://www.eclipse.org/downloads/packages/eclipse-classic-372/indigosr2***

Una vez realizado esto nos dirigimos a las carpeta /tools dentro de lo que hemos descomprimido y ejecutamos "*./Android*" esto nos abrirá el gestor de Versiones o SDK Manager que nos permitirá gestionar las versiones o herramientas que usaremos para poder generar nuestra aplicación.



En nuestro caso, seleccionamos todos los paquetes de Android 4.0.3 y de Android 2.2.

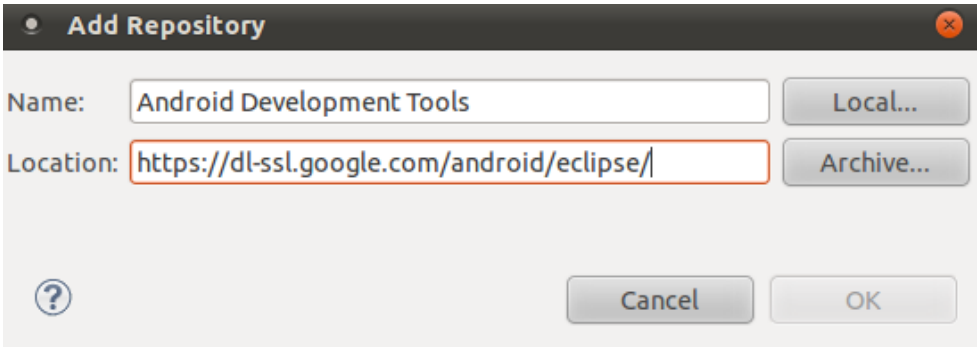
Plugin de Android para Eclipse

Para hacer el desarrollo más fácil, Google ha programado un plug-in para Eclipse llamado Android Development Kit (ADT). Para instalar el plug-in podemos seguir estos pasos (las instrucciones son para la versión 3.5 de Eclipse y con los menús en ingles originales, otras versiones pueden variar un poco en situación o nombres de menús):

1. Iniciamos Eclipse y seleccionamos un directorio de trabajo (Workspace) si no lo hemos hecho antes.
2. Seleccionar el menú Help y clicamos en Install New Software.
3. Seleccionamos la opción de "Avaiable Software Sites".

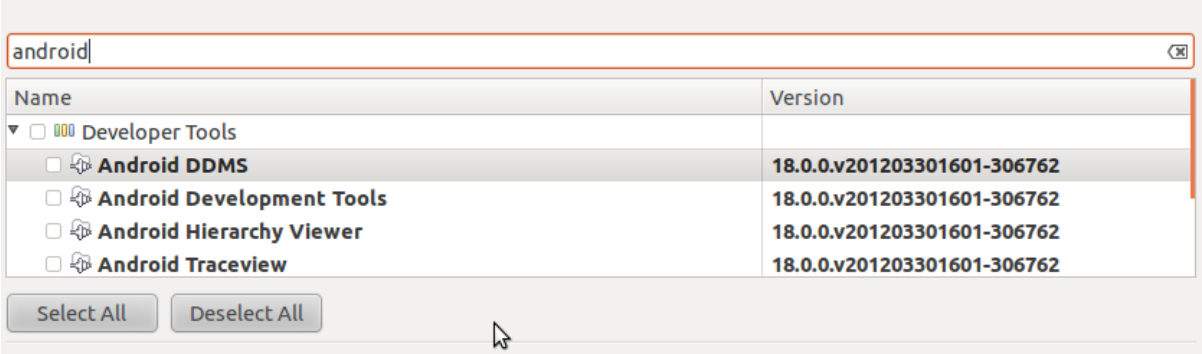
4. Clicamos en el Botón Add.

5. Introducimos la dirección de actualizaciones de software de Android: https://dl-ssl.google.com/android/eclipse/



6. Apretamos OK y volvemos a la pantalla de nuevo software.

7. Escribimos “android” en la casilla de filtro y apretamos intro. “Developer Tools” debería aparecer en la lista abajo.

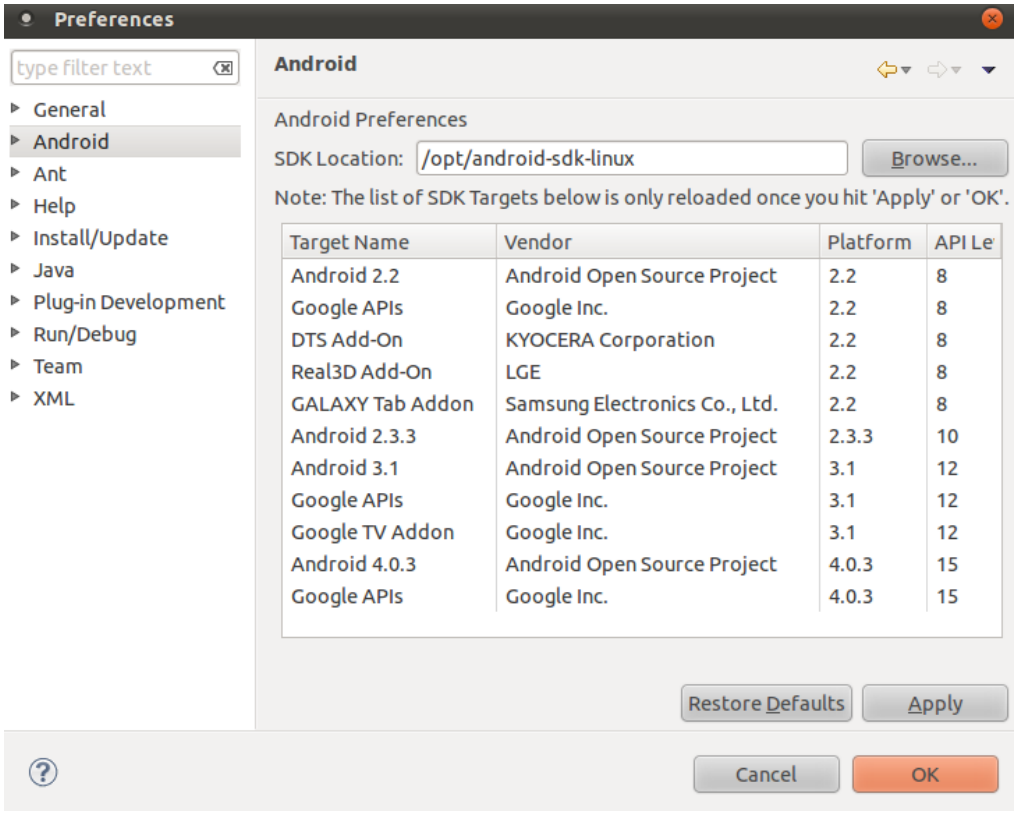


8. Seleccionamos el checkbox al lado de Developer Tools y apretamos siguiente.

9. Revisamos la lista de paquetes que queremos instalar y aceptamos la licencias de uso para que empiece la descarga y la instalación.

10. Una vez instalado, reiniciamos Eclipse.

11. Cuando Eclipse vuelve a encenderse, recibirás un error porque hay que especificar donde se encuentra el SDK de Android. Entramos en la pestaña Windows > Preferences > Android y configuramos el directorio del SDK de Android que descomprimimos al principio del manual y que introdujimos en la variable PATH.


Preferences
type filter text
General
Android
Ant
Help
Install/Update
Java
Plug-in Development
Run/Debug
Team
XML
Android
Android Preferences
SDK Location:
/opt/android-sdk-linux
Browse...
Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.
Target Name
Vendor
Platform
API Le
Android 2.2
Android Open Source Project
2.2
8
Google APIs
Google Inc.
2.2
8
DTS Add-On
KYOCERA Corporation
2.2
8
Real3D Add-On
LGE
2.2
8
GALAXY Tab Addon
Samsung Electronics Co., Ltd.
2.2
8
Android 2.3.3
Android Open Source Project
2.3.3
10
Android 3.1
Android Open Source Project
3.1
12
Google APIs
Google Inc.
3.1
12
Google TV Addon
Google Inc.
3.1
12
Android 4.0.3
Android Open Source Project
4.0.3
15
Google APIs
Google Inc.
4.0.3
15
Restore Defaults
Apply
Cancel
OK

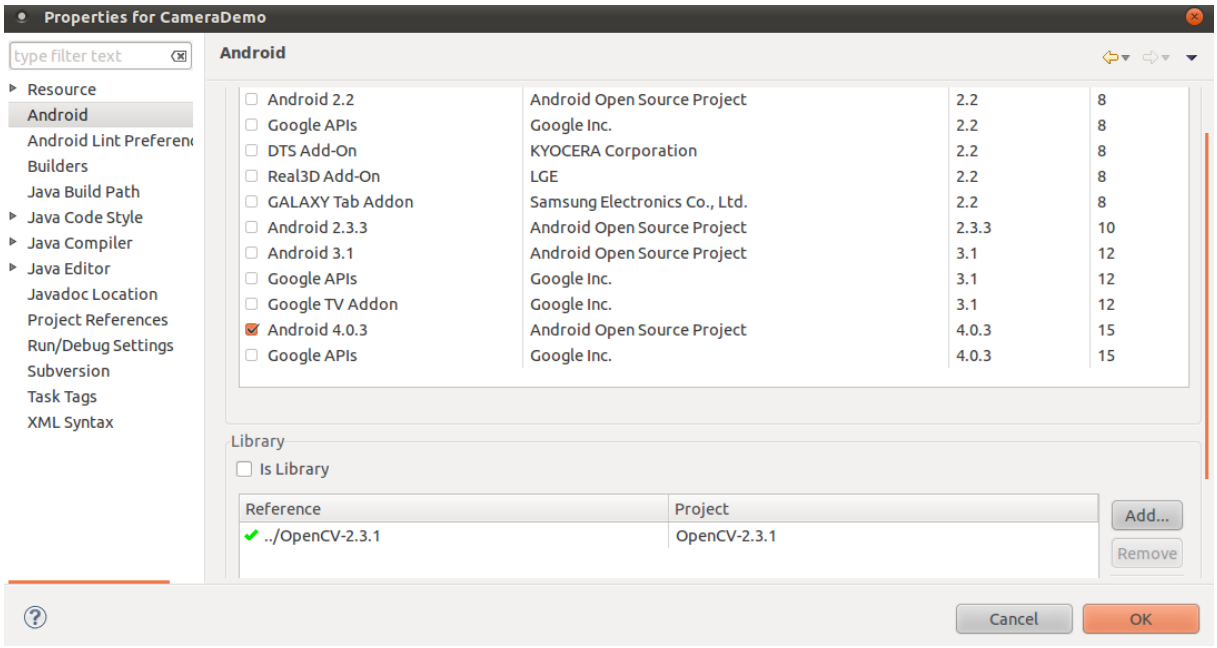
Integración con OPENCV

Para poder utilizar las funciones de OpenCV en Android solo debemos incluirlo en nuestro proyecto como haríamos con cualquier proyecto Java. Para ello en primer lugar descargamos la librería, versión para Android en la sección Downloads.

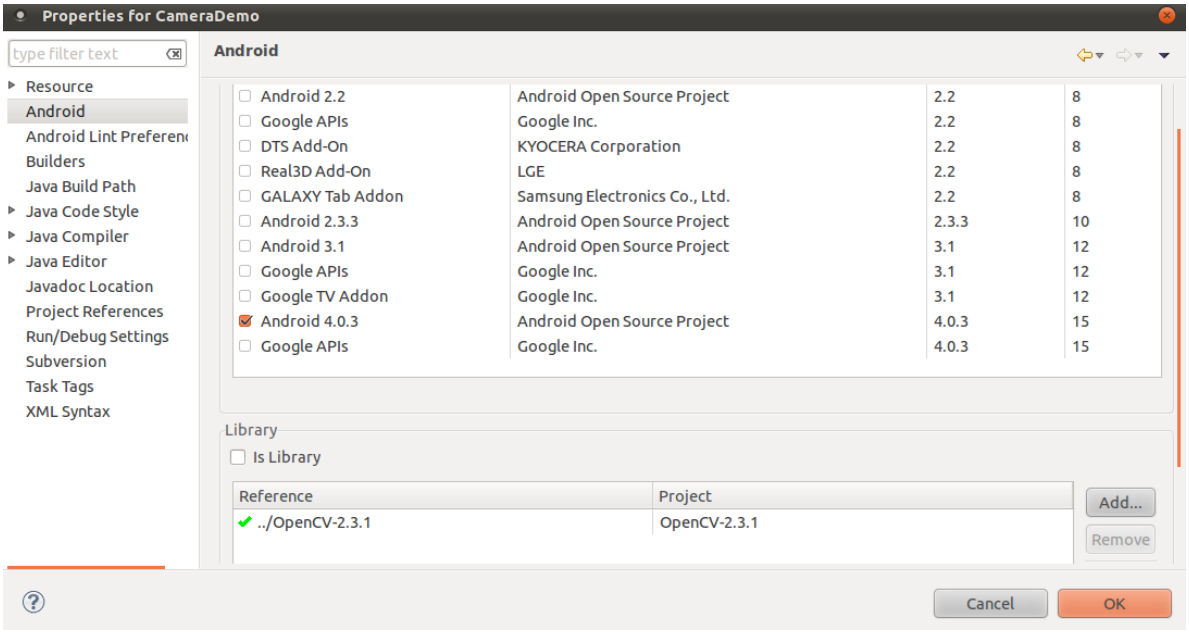
Una vez hecho esto descomprimimos la carpeta en cualquier parte, no es necesario que sea dentro de la carpeta del proyecto.

En ese momento nos dirigimos a Eclipse, y entramos dentro de las propiedades de nuestro proyecto. Una vez hecho esto tenemos dos manera de hacerlo.

Podemos dirigirnos a la pestaña Android y en la casilla ***Library***, apretar el botón ***Add*** y añadir la carpeta donde hemos descomprimido la librería.

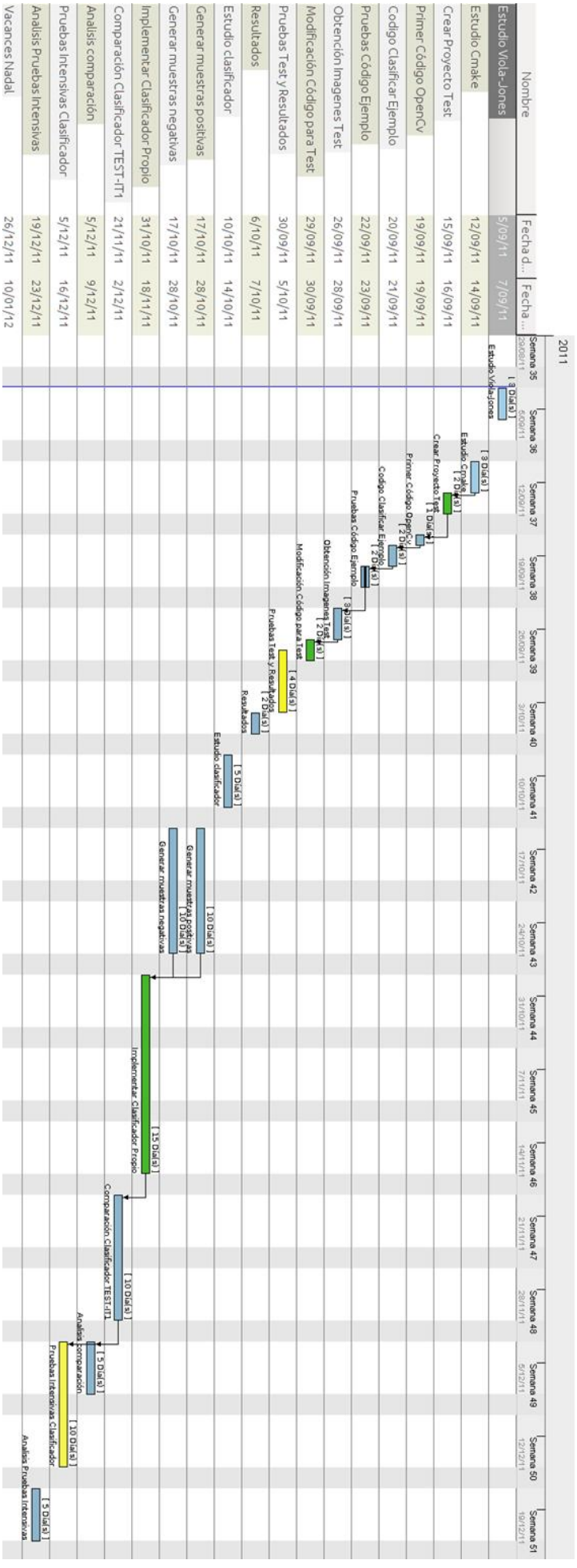


O ir a la pestaña ***Java Built Path***, seleccionar la pestaña ***libraries*** y apretar ***Add External JAR***. En este caso no incluiremos la carpeta de la librería si no el fichero opencv-2.3.1.jar dentro de la carpeta /bin en la carpeta de la librería.



De las dos maneras tendremos nuestro sistema listo para usar con las librerías OpenCV y podremos disponer de las funciones que nos permiten la manipulación de imágenes dentro de nuestro entorno de desarrollo Android.

ANEXO 3: Planificación Temporal Detallada



Nombre	Fecha d...	Fecha ...
Vacances Nadal	26/12/11	10/01/12
Analisis resultados 1ª Iteración	11/01/12	24/01/12
Definición siguente fase	25/01/12	27/01/12
Analisis sobre como obtener imagen...	30/01/12	3/02/12
Programación SetResults	6/02/12	24/02/12
Obtener Imagenes 2ª Iteración	27/02/12	9/03/12
Entrenamiento 2ª Iteración	12/03/12	16/03/12
Experimentos 2º Clasificador	19/03/12	30/03/12
Estudio Android	2/04/12	13/04/12
Programación aplicación Android	9/04/12	20/04/12
Pruebas aplicación Android	23/04/12	27/04/12
Aplicación Difuminado	23/04/12	4/05/12
Pruebas Difuminado en Android	7/05/12	11/05/12
Conclusiones Finales	14/05/12	25/05/12
Finalización Documentación	21/05/12	8/06/12