MÁSTER EN INGENIERÍA DE SISTEMAS Y DE CONTROL

Universidad Complutense de Madrid

Universidad Nacional de Educación a Distancia

Trabajo Fin de Máster

Desarrollo de aplicaciones para biometría sobre dispositivos de bajo coste y bajo consumo

Septiembre 2019

Autor: Víctor J. González Rodríguez

Director: Guillermo Botella Juan

MÁSTER EN INGENIERÍA DE SISTEMAS Y DE CONTROL

TRABAJO FIN DE MÁSTER

Desarrollo de aplicaciones para biometría sobre dispositivos de bajo coste y bajo consumo

Autor: Víctor J. González Rodríguez

Director: Guillermo Botella Juan

Autorización de difusión

Autorizamos a la Universidad Complutense y a la Universidad Nacional de Educación a Distancia a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado el autor:

Víctor J. González Rodríguez

Septiembre 2019

Resumen

Este Trabajo Fin de Máster describe el trabajo realizado para analizar las posibilidades de crear un sistema de reconocimiento facial en tiempo real en dispositivos de bajo coste, concretamente en la plataforma Raspberry Pi.

Para conseguir este objetivo, se ha implementado una aplicación modular que abarca todo el proceso de detección y reconocimiento facial. Esta aplicación ha sido diseñada de forma que permita intercambiar fácilmente los algoritmos utilizados en cada uno de los pasos este proceso. Tras ello, se han realizados diversas pruebas para evaluar la eficiencia de estos algoritmos, así como el impacto computacional de los mismos, con objeto de determinar cuál sería la mejor arquitectura para su ejecución en un dispositivo de bajo coste.

Dado el carácter portable y la importancia de la economía energética en estos dispositivos, se ha realizado también un análisis de la eficiencia energética de la aplicación.

Abstract

This Master's Thesis describes the work realized to analyze the possibilities of creating a facial recognition system in real time focused on low cost devices, specifically in the Raspberry Pi platform.

In order to achieve this goal, a modular application has been created, covering the whole process of facial detection and recognition. This application has been designed focusing in the capacity of interchange easily the algorithms used in every step of this process. After that, different tests have been realized in order to evaluate the efficiency of these algorithms, as well as their computational impact. The final goal is to determinate which is the best architecture for their execution in a low-cost device.

Given the portable character and the importance of the energy economy in these devices, an analysis of the energy efficiency of the application has also been realized.

Palabras clave

Detección facial, reconocimiento facial, Raspberry Pi, Python, OpenCV, Viola Jones, Histograma de Gradientes, Máquinas de Soporte Vectorial, k Vecinos más Próximos, Bolsa de Palabras Visuales, SIFT, SURF, Fisherfaces, Eigenfaces, Local Binary Pattern

Keywords

Facial detection, facial recognition, Raspberry Pi, Python, OpenCV, Viola Jones, Histogram of Gradients, Support Vector Machines, K-Nearest Neighbours, Bag of Visual Words, SIFT, SURF, Fisherfaces, Eigenfaces, Local Binary Pattern

Tabla de contenido

Tabla de ilustraciones

1.- Introducción

1.1.- Motivación del proyecto

En la actualidad, podemos encontrar sistemas de detección y reconocimiento facial en prácticamente cualquier lugar: sistemas de videovigilancia, redes sociales, sistemas de clasificación de fotografías, aplicaciones de ocio, mecanismos de identificación para sistemas de pago, … Por tanto, se puede decir que es una tecnología madura que ha sido estudiada profusamente, aunque eso no impide que continuamente se sigan produciendo nuevos avances en la misma.

Uno de los puntos clave para el avance en el estudio de este campo ha sido la creciente potencia de los microprocesadores y de los sistemas informáticos en general, ya que todas las técnicas utilizadas en los sistemas de detección y reconocimiento facial son grandes consumidores de CPU. A medida que la ciencia avanza en este campo, también avanzan los requisitos computacionales de los algoritmos utilizados, resultando inviable su ejecución en sistemas menos recientes. Así, por ejemplo, durante los últimos años se ha investigado en sistemas de reconocimiento mediante redes neuronales que tienen una muy alta tasa de éxito, pero a costa de grandes consumos de CPU.

Este avance ha dejado de lado aquellos sistemas más limitados en recursos, tales como la Raspberry Pi, pero en los que sería interesante poder lanzar sistemas de detección y reconocimiento facial, por ejemplo, para su uso en sistemas de videovigilancia, aprovechándose de su alta portabilidad y pequeño tamaño.

1.2.- Planteamiento y objetivos

Teniendo en cuenta lo expuesto en el apartado anterior, el objetivo del presente proyecto será el diseño e implementación de un sistema que, a partir de las imágenes obtenidas de la Raspberry Pi, detecte todas las caras presentes en cada imagen e identifique la persona de la que se trata.

Para esto, hay que tener en cuenta que hay que combinar dos procesos claramente diferenciados: la detección de las áreas de una imagen que contengan una cara y, partiendo de un conjunto de personas conocidas por la aplicación, la identificación de cada una de las caras.

Una máxima que hay que tener en cuenta durante todo el proceso de desarrollo es la necesidad de optimizar el tiempo de computación, siendo el objetivo final perseguido el implementar un sistema que permita realizar los procesos anteriores en tiempo real. Dado que, para que el ojo humano detecte movimiento a partir de una serie de imágenes es necesario que estas se reproduzcan a una tasa aproximada de 24 imágenes por segundo, se establecerá este valor como parámetro de consecución del objetivo. Es decir, se debería conseguir que el procesamiento de cada una de las imágenes individuales del vídeo obtenido se analice en aproximadamente 42 milisegundos.

Algunos aspectos que hay que tener en cuenta para alcanzar este objetivo serán:

- Se evitará el uso de los algoritmos más recientes, especialmente los basados en redes neuronales, por su excesivo consumo de CPU, centrando el estudio en los algoritmos más antiguos y con menos necesidades de computación.
- El procesador de la Raspberry Pi tiene una frecuencia de reloj de únicamente 1,2 GHz, aunque dispone de 4 núcleos. Por tanto, será necesario potenciar el uso de todos los núcleos en aquellos procesos en los que sea posible aprovechando las funcionalidades de programación multihilo de Python.
- Asimismo, será prioritario detectar los cuellos de botella en todas las etapas del procesamiento de una imagen, desde su obtención hasta el etiquetado de las caras en ella, con objeto de modificarlos o eliminarlos para reducir su impacto en el resultado final.

1.3.- Metodología y plan de trabajo

La consecución de los objetivos anteriores requiere del estudio individual de un gran número de etapas, que se pueden dividir en dos grandes grupos: detección y reconocimiento facial, cada uno de los cuales se divide a su vez en diversas fases.

Para cada una de estas fases hay diversos algoritmos disponibles que permitirán alcanzar los objetivos de esta. Hay que tener en cuenta que cada uno de estos algoritmos tiene sus características propias que se traducirán en una serie de ventajas e inconvenientes, siendo uno de los objetivos identificar aquellos más adecuados en cada fase, considerando como adecuado aquellos que requieran menos tiempo de procesamiento para conseguir una tasa de éxito aceptable.

Teniendo lo anterior en cuenta, se diseñará un sistema modular que permita intercambiar los algoritmos de cada fase de una forma sencilla y transparente, sin afectar al resto de fases. Lo más adecuado para alcanzar este objetivo es la utilización de una metodología orientada a objetos.

Otro aspecto importante es cómo se evaluará el rendimiento de los algoritmos utilizados. Como se ha dicho anteriormente, habrá dos métricas de interés, el tiempo de computación y la eficiencia de los algoritmos. Evaluar el tiempo requerido es relativamente sencillo, pero la precisión será algo más complicado, a fin de cuentas, ¿qué se entiende por buenos resultados para un sistema de detección facial? ¿qué detecte correctamente todas las caras, aunque también detecte como tales imágenes que no sean caras? ¿o que todo lo que detecte sean caras, aunque se deje alguna cara sin detectar?

Para darle una medida a estas respuestas se utilizará la métrica conocida como *precision and recall*, que utiliza diversos parámetros para determinar el rendimiento de un algoritmo, tales como la precisión, la exhaustividad o la exactitud.

En este punto hay que tener en cuenta que esta técnica requiere su uso sobre un conjunto de imágenes ya etiquetadas y correctamente identificadas, es decir, sobre un entorno prefijado y muy controlado, alejado del entorno real sobre el que se ejecutará finalmente la aplicación. Por lo tanto, es probable que los resultados obtenidos no se puedan trasladar directamente a

los que se obtendrán en un entorno real, donde influirán otros factores como la iluminación ambiental, aunque sí que permitirán comparar el rendimiento de los diversos algoritmos.

2.- Estado del arte

2.1.- Raspberry Pi

La Raspberry Pi es una familia de placas de computadoras de bajo coste desarrollada en el año 2011 por la Fundación Raspberry Pi en la Universidad de Cambridge de Reino Unido, siendo su objetivo la promoción de la enseñanza de la informática en las escuelas, así como en los países en vías de desarrollo.

Con un precio muy contenido, en torno a los 35€, se ha convertido en un mini-ordenador muy popular con más de 20 millones de unidades vendidas a fecha de junio de 2019. Se puede decir que no solamente ha alcanzado los objetivos para los que se ha desarrollado, sino que también es una plataforma ampliamente utilizada en aplicaciones relacionadas con el Internet de las Cosas (IoT) y todo lo relacionado con el Hardware Libre.

El nombre de Raspberry Pi supone un guiño a todos los ordenadores con nombres basados en frutas tales como Apple, BlackBerry o la compañía inglesa Apricot, junto con la referencia al concepto de un ordenador sencillo que puede ser programado utilizando Python (acortado como Pi).

Desde su aparición ha habido diferentes modelos con diferentes características y funcionalidades, otorgando a la plataforma de mayor potencia y capacidad de cómputo a medida que han surgido nuevas versiones.

								Conectividad		
	Modelo	**Fecha**	**RAM**	**CPU**	**Cores**	**Frec.**	**USB**	**Ethernet**	**Wi-Fi**	**Bluetooth**
Modelo B	Model B	Mar-2012	512 MB	ARMv6Z (32 bit)	1	700 MHz	2	Si	No	No
	Model B+	Jul-2014	512 MB	ARMv6Z (32 bit)	1	700 MHz	4	Si	No	No
	2 Model B	Feb-2015	1 GB	ARMv7-A (32 bit)	4	900 MHz	4	Si	No	No
	2 Model B v1.2	Oct-2016	1 GB	ARMv8-A (32 bit)	4	900 MHz	4	Si	No	No
	3 Model B	Feb-2016	1 GB	ARMv8-A (64/32 bit)	4	1.2 GHz	4	Si	Si	Si
	3 Model B+	Mar-2018	1 GB	ARMv8-A (64/32 bit)	4	1.4 GHz	4	Si	Si	Si
Modelo A	Model A	Feb-2013	256 MB	ARMv6Z (32 bit)	1	700 MHz	1	No	No	No
	Model A+	Nov-2014	512 MB	ARMv6Z (32 bit)	1	700 MHz	1	No	No	No
	4 Model A+	Nov-2018	512 MB	ARMv8-A (64/32 bit)	4	1.4 GHz	1	No	Si	Si
Zero	Zero v1.2	Nov-2015	512 MB	ARMv6Z (32 bit)	1	1 GHz	1	No	No	No
	Zero v1.3	May-2016	512 MB	ARMv6Z (32 bit)	1	1 GHz	1	No	No	No
	Zero Wireless	Feb-2017	512 MB	ARMv6Z (32 bit)	1	1 GHz	1	No	Si	Si
Compute Module	Compute Module	Abr-2014	512 MB	ARMv6Z (32 bit)	1	700 MHz	1	No	No	No
	CM3	Ene-2017	1 GB	ARMv8-A (64/32 bit)	4	1.2 GHz	1	No	No	No
	CM3 Lite	Ene-2017	1 GB	ARMv8-A (64/32 bit)	4	1.2 GHz	1	No	No	No
	CM3+	Ene-2019	1 GB	ARMv8-A (64/32 bit)	4	1.2 GHz	1	No	No	No
	CM3+ Lite	Ene-2019	1 GB	ARMv8-A (64/32 bit)	4	1.2 GHz	1	No	No	No

El modelo más relevante es el denominado **Modelo B**. Este modelo ha evolucionado a lo largo de múltiples versiones en los últimos años. El modelo B original con 256 MB de RAM que se convirtieron en 512 MB en la revisión 2, fue reemplazado por el modelo B+ que aumentaba el número de pines GPIO hasta los 40, pasaba a disponer de un slot para tarjetas microSD e incrementaba el número de puertos USB hasta 4. Estos modelos utilizaban el SOC (*System on Chip*) Broadcom BCM2835, que incorporaba en un mismo encapsulado un único núcleo ARM11 a 700 MHz y la GPU que se ha conservado en todos los modelos, la Broadcom VideoCore IV.

La aparición en 2015 de la siguiente versión, denominada **Raspberry Pi 2 Modelo B** (o 2B) supuso una muy importante mejora en la capacidad de cómputo de la placa ya que implementaba el SOC Broadcom BCM2836 con un procesador de 32 bits y cuatro núcleos ARM Cortex A7, que además aumentaba su frecuencia hasta los 1.2 GHz. Además, este SOC mejorado añadía soporte para Ubuntu y para Windows 10 IoT.

El último modelo disponible es el **Raspberry Pi 3 Modelo B+**, cuyo SOC es el Broadcom BCM2837, también de cuatro núcleos, pero en esta ocasión con una arquitectura de 64 bits, aparte de algunas mejoras en conectividad como la incorporación de Wi-Fi y Bluetooth.

En el año 2015 apareció una versión ultra-compacta denominada **Pi Zero** con una fuerte vocación hacia el Internet de las Cosas donde el precio y el tamaño tienen una especial relevancia. Este modelo consigue estos objetivos reemplazando el conector HDMI por un mini-HDMI, eliminando los puertos CSI y DSI (para la cámara y un display respectivamente), decrementando la RAM hasta 512 MB y optando por un SOC single-core ARM 11 a 1 Hz. A cambio reduce su tamaño hasta 65 por 30 mm con un coste de apenas 5€, que aumentan hasta los 10€ en la versión Pi Zero W que incorpora Wi-Fi y el conector CSI para la cámara.

Otra rama de la Raspberry Pi es la denominada **Compute Module**, que, al contrario que las otras ramas, no está orientada al ámbito educativo o *maker*, sino que su objetivo son los entornos empresariales e industriales. Esta rama apareció en el año 2014 y ha sido renovada bianualmente con dos versiones cuya principal diferencia es la inclusión de una memoria eMMC (de hasta 32 GB en el CM3+) integrada en la placa de que no dispone la versión Lite.

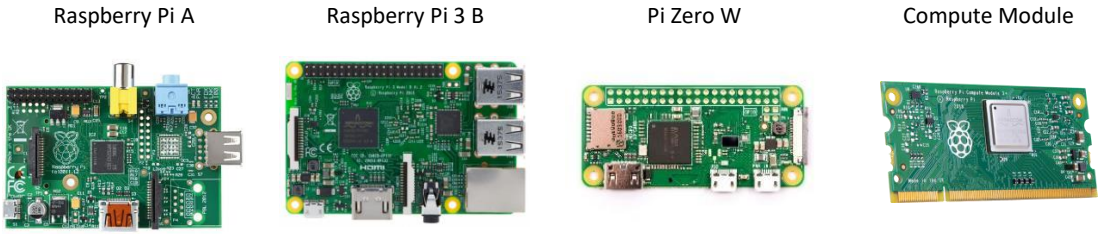


Figura 1: Modelos de Raspberry Pi

La vocación industrial de la rama *Compute Module* se puede ver claramente en la ya mencionada memoria eMMC integrada en la placa, en su ajustado consumo, y, sobre todo, en su factor de forma, compatible con los conectores SODIMM. Estos conectores son disponibles a costes muy contenidos por parte de múltiples vendedores, lo que facilita su inclusión dentro de otro hardware.

Dada la orientación de la Raspberry Pi hacia el Internet de las Cosas, dispone de una serie de puertos destinados a su conexión con diferentes componentes hardware, en concreto los pines GPIO como puertos de entrada y salida para su conexión con sensores y actuadores, el

puerto DSI para la conexión con un display y el componente más interesante para el presente proyecto, el puerto CSI para la conexión de una cámara.

La cámara de la Raspberry Pi original apareció en el año 2013 y disponía de un sensor OmniVision OV5647 de 5 Megapíxeles y con una versión para luz visible y otra para infrarrojos, lo que la hacía muy adecuada para proyectos de videovigilancia.

En el año 2016 apareció una segunda versión, Camera Module v2, que reemplazaba el sensor por un Sony IMX219 con 8 Megapíxeles. Este sensor permite capturar imágenes de 3280x2464 píxeles, o bien vídeo en HD (1920x1080 píxeles) a 30 *frames* por segundo.

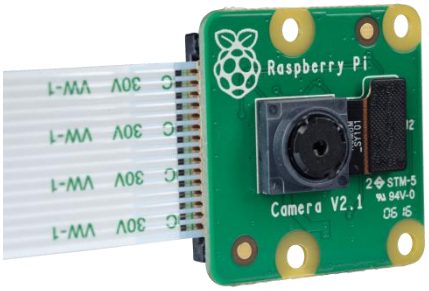


Figura 2: Raspberry Pi Camera v2.1

2.2.- Detección y reconocimiento facial

Hasta hace unos pocos años los ordenadores eran incapaces de interpretar el contenido de una fotografía. Mientras que para los seres humanos es muy fácil extraer información de una imagen, para los ordenadores suponía una secuencia de píxeles sin ningún significado más allá de los colores de cada uno de los píxeles. Sin embargo, en los últimos años ha surgido una nueva área de estudio, la **visión por computador**, cuyo objetivo es que los ordenadores sean capaces de adquirir, procesas, analizar y comprender las imágenes del mundo real.

Esta área de estudio abarca muchos campos, uno de ellos es la **detección de objetos**, es decir, ser capaz de interpretar la información contenida en una imagen para detectar los objetos mostrados en ella.

Aunque se pueden detectar todo tipo de objetos en una imagen, cobran especial importancia algunos relevantes para ciertas aplicaciones. Por ejemplo, hay múltiples desarrollos de sistemas de detección de peatones y coches para su implantación en los vehículos autónomos. Otro objeto sobre el que se ha estudiado ampliamente es la **detección facial**, de especial importancia por ejemplo para sistemas como cámaras fotográficas para enfocar correctamente las caras en una escena.

Una vez detectadas las caras en una imagen, el siguiente paso natural es ser capaz de identificar a la persona a la que corresponda dicha cara. Esto da lugar a los sistemas de **reconocimiento facial**, que son capaces de identificar la persona a la que pertenece una cara. Los sistemas de reconocimiento facial son ampliamente utilizados por las fuerzas de seguridad en los sistemas de video vigilancia (por ejemplo, en aeropuertos), pero también en entornos más domésticos, como las redes sociales, muchas de las cuales son capaces de etiquetar automáticamente a todas las personas que se encuentran en una imagen.

2.2.1.- Historia de la detección facial

La detección y el reconocimiento facial son probablemente unos de los campos que más interés ha despertado en el área de la visión por computador, y, por tanto, en el que más se ha estudiado. Aunque es en los últimos años cuando se han conseguido resultados realmente satisfactorios, ha sido objeto de investigación prácticamente desde la aparición de las primeras

computadoras. Por ejemplo, en (Sakai, Nagao, & Kanade, 1971) ya se propone un método para la detección de caras basado en la localización de puntos característicos, como la nariz, barbilla, …

Aunque se desarrollaron múltiples métodos, especialmente durante los años 90, todos compartían el mismo problema, y era su falta de aplicabilidad en el mundo real, en entornos no controlados con unas características ambientales que distaban mucho de ser las óptimas (mala iluminación, caras no completamente visibles, …) y donde había que detectar las caras en tiempo real.

El mérito de romper esta barrera hay que otorgárselo a Viola y Jones, que en el año 2001 presentaron un artículo (Viola & Jones, 2001) donde exponían un método de detección de caras basado en las cascadas de Haar. Este método permitía la detección de caras en aplicaciones en el mundo real, con una alta tasa de éxito y con un coste computacional bastante bajo, haciendo viable su utilización en una gran gama de dispositivos. A pesar de su antigüedad, este método aún es ampliamente utilizado en multitud de pequeños dispositivos, tales como cámaras de fotos o teléfonos móviles.

Este hecho supuso un hito que ha dado lugar a grandes avances en el campo de la detección facial. Junto con el incremento en la potencia de los ordenadores y la capacidad de almacenamiento, en el artículo *A Survey of Face Detection in the Wild: Past, Present and Future* (Zafeiriou, Zhang, & Zhang, 2015) se identifican cuatro factores clave que han beneficiado las investigaciones en este campo:

1. La introducción de metodologías para la extracción de características robustas, tales como *Scale Invariant Feature Transform (SIFT)*, *Histogram of Oriented Gradients (HOG)*, *Local Binary Patterns (LBP)* o *Speeded Up Robust Features (SURF)*, que han permitido discernir entre la información útil para la detección de una imagen y la información irrelevante que puede ser descartada.
2. Los esfuerzos por parte de la comunidad de investigación para desarrollar bases de datos con imágenes sobre las que trabajar, en la actualidad hay gran número de estos *datasets* accesibles para cualquiera con miles de imágenes correctamente identificadas y etiquetadas.
3. El desarrollo de metodologías tales como el *Boosting*, las *Support Vector Machines (SVM)* o las redes neuronales, que permiten clasificar los descriptores de las imágenes para determinar si pertenecen a un determinado tipo, por ejemplo, una cara, o no.
4. El desarrollo de repositorios con código de alta calidad públicamente disponibles. Un ejemplo claro de esto son las librerías OpenCV o Dlib, que han supuesto una de las piedras angulares para el desarrollo de este proyecto.

Si en 2001 el método de Viola Jones supuso un punto de inflexión en la detección de objetos, en el año 2012 hubo un segundo hito que marcó el camino a seguir en este campo hasta la actualidad, donde las redes neuronales convolucionales (CNNs) son ampliamente utilizadas. Esta primera red neuronal, llamada **AlexNet**, tenía 650.000 neuronas y ganó el primer premio de la Competición ImageNet de ese año, con un error de únicamente un 15,3% en la clasificación de 150.000 fotografías frente a errores superiores al 26% de los métodos tradicionales.

2.2.2.- Historia del reconocimiento facial

Al igual que pasa con los métodos de detección facial, los intentos de desarrollar un sistema de reconocimiento facial se remontan prácticamente a mediados del siglo pasado. Uno de los primeros intentos fue el propuesto por Woodrow Wilson Blesoe (Blesoe, 1963), que desarrolló un sistema denominado *reconocimiento facial hombre-máquina (man-machine facial recognition)* capaz de clasificar fotos de caras seleccionando manualmente mediante un *stylus* los puntos clave de las caras en un dispositivo que llamó *RAND Tablet*. Una vez introducidos dichos puntos se almacenaban sus coordenadas en una base de datos de forma que, cuando se introducía una cara nueva en el sistema, era capaz de extraer la cara más similar de las almacenadas utilizando un algoritmo que clasificaba dos caras como la misma basándose en la distancia entre los puntos clave de esta.

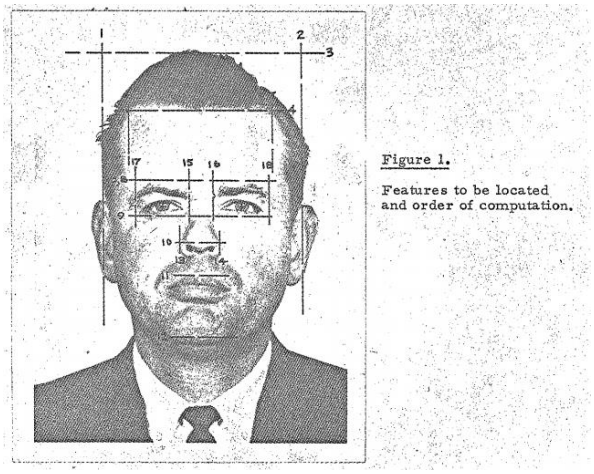


Figura 3: Selección manual de puntos clave con RAND Tablet

Obviamente los resultados no eran muy precisos ya que pequeños cambios en la pose o del entorno podían fácilmente confundir al algoritmo.

En 1986, Sirovich y Kirby (Sirovich & Kirby, 1986) propusieron la aplicación del álgebra lineal al problema del reconocimiento facial con objeto de obtener una representación con menos dimensiones de imágenes faciales. Demostraron que tomando únicamente un conjunto de características básicas de la imagen se podía codificar correctamente una cara normalizada con apenas un centenar de valores. Este método se conoce como Eigenfaces.

Al igual que con los métodos de detección facial, la existencia de amplias bases de datos de imágenes faciales etiquetadas es imprescindible para el desarrollo de sistemas de reconocimiento facial. La primera de estas bases de datos fue publicada conjuntamente por DARPA (*Defense Advanced Research Projects Agency*) y NIST (*National Institute of Standards and Technology*) con el nombre de programa FERET (*Face Recognition Technology*) (NIST, 1993) con el propósito de animar el mercado comercial del reconocimiento facial.

FERET fue actualizado en el año 2003 para incluir versiones en alta resolución con color de 24 bits de las imágenes. La base de datos incluye un total de 2413 imágenes que corresponden a 856 personas diferentes.

En 2010, Facebook comenzó a utilizar el reconocimiento facial para etiquetas las fotos subidas por los usuarios de esta red social. A pesar de la polémica inicial, no tuvo ningún impacto negativo sobre su popularidad. En la actualidad, más de 350 millones de fotos son subidas y etiquetadas diariamente.

En este momento, los sistemas de reconocimiento facial están en un punto en el que las múltiples ventajas de su uso chocan con la pérdida de privacidad argumentada por sus detractores. Usada de forma apropiada, esta tecnología puede tener múltiples usos para facilitar la vida de la gente. En España poco a poco se va implantando en diversos ámbitos: en

algunas oficinas de CaixaBank es posible sacar dinero de un cajero sin necesidad de introducir el código PIN; AENA, junto con Air Europa, ha puesto en marcha un proyecto piloto en el aeropuerto de Menorca para permitir acceder a los pasajeros a la zona de embarque y subirse al avión y seguro que en los próximos años veremos una integración paulatina de reconocimiento facial en nuestras vidas.

En el otro extremo se encuentra el ejemplo de China, un país donde hay 170 millones de cámaras de seguridad con reconocimiento facial repartidas por todo el territorio (y se planea instalar otros 450 millones para el año 2020) (Aldama, 2018). El objetivo de todas estas cámaras es alimentar un sistema denominado Dragonfly Eye, desarrollado por la empresa china Yitu, cuya motivación es la detección de delincuentes, pero indudablemente a costa de una pérdida de privacidad de los ciudadanos.

2.3.- Fundamentos algorítmicos: características

Las imágenes obtenidas de la cámara tienen demasiada información extra que no es necesaria para el posterior proceso de clasificación. Para descartar toda la información irrelevante es necesario extraer las características de la imagen, pudiendo definirse una característica como "*una pieza de información relevante para resolver una tarea computacional relacionada con una determinada aplicación*". Es decir, el proceso de **extracción de características** permite simplificar la imagen eliminando toda la información no relevante y manteniendo únicamente la información que puede ser útil para la posterior clasificación de la imagen.

Las características relevantes pueden variar en función del tipo de objeto que se quiera detectar, ya que diferentes características enfatizan en diferentes cualidades de la imagen. Así, por ejemplo, si se quieren detectar monedas en una imagen lo más adecuado puede ser la utilización de un detector de bordes para simplificar la misma, ya que se sigue manteniendo la información relativa a su forma circular mientras que se descarta información menos relevante como puede ser el color de las monedas.

Figura 4: Ejemplo de un detector de bordes

Las características de una imagen se suelen representar como un vector en un espacio n-dimensional, el cual se denomina **descriptor de la imagen**.

Hay múltiples detectores de características y cada uno enfatiza un determinado aspecto de la imagen por lo que el tipo de objeto que se quiere detectar es un aspecto que hay que tener en cuenta a la hora de elegir un algoritmo.

Algunos de los más destacables, y que se han utilizado en el presente proyecto, son:

- **Características de Haar**. Utilizado en el algoritmo de Viola-Jones, enfatiza en los cambios de intensidad en las imágenes, así como su dirección.
- **Histogram of Oriented Gradients (HOG)**. Utilizado inicialmente para la detección de peatones, se basa en la distribución de los gradientes de intensidad, es decir, en la dirección de los bordes.
- **Scale Invariant Feature Transform (SIFT)**. En este caso las características que se buscan en la imagen son los denominados puntos de interés.
- **Speeded Up Robust Feature (SURF)**. Al igual que SIFT también se basa en los puntos de interés de la imagen.

2.3.1.- Características de Haar (Algoritmo de Viola-Jones)

En el año 2001 Paul Viola y Michael Jones publicaron el artículo *Rapid Object Detection using a Booster Cascade of Simple Features* (Viola & Jones, 2001) lo cual supuso indudablemente un punto de inflexión en el ámbito de la detección facial. Hasta entonces los métodos disponibles de detección facial requerían intensivos consumos de CPU haciéndolos totalmente inviables para su utilización en sistemas en tiempo real. Sin embargo, el algoritmo de Viola y Jones era capaz de detectar caras en imágenes de 384 por 288 píxeles a 15 frames por segundo sobre un Intel Pentium II a 700 MHz., lo que lo hacía apto para ser utilizado en aplicaciones para el mundo real.

Características

El sistema propuesto por Viola y Jones se basa en las **características de Haar**. Estas características fueron propuestas por primera vez para su uso en sistemas de detección de objetos por Papageorgiou en su artículo "*A general framework for object detection*" (Papageorgiou, Oren, & Poggio, 1998) donde proponía la utilización de un conjunto de características basadas en las wavelets de Haar en lugar de las habituales imágenes de intensidades.

En concreto proponen el uso de tres tipos de características:

- Características de dos rectángulos
- Características de tres rectángulos
- Características de cuatro rectángulos

En la figura 5 se muestran cinco ejemplos de estas características. Hay que tener en cuenta que estas que se muestran aquí no son únicas, sino que se puede elegir otras diferentes, por ejemplo, rotaciones de estas que serán más sensibles a bordes en diagonal.

Figura 5: Ejemplos de características de Haar

Todas estas características se basan en la misma idea: se selecciona un área de la imagen y se le asigna como valor de dicha característica el de la diferencia entre las sumas de las intensidades de diferentes partes de dicha área. Los valores de los píxeles que coinciden con la

parte negra de la característica contribuyen de forma negativa al valor final y los valores de los píxeles de la parte blanca contribuyen de forma positiva.

En la figura 6 se puede apreciar el resultado de aplicar tres características diferentes a una imagen. La primera característica calculada está formada por dos rectángulos dispuestos horizontalmente, con un tamaño de 4x1 píxeles. Se puede observar que es sensible a los bordes verticales que hay en la imagen.

La segunda característica mostrada es la misma, pero en este caso con un tamaño de 2x6 píxeles. También refleja los bordes verticales, pero en este caso sobre una mayor extensión. Por último, se muestra la característica de cuatro rectángulos que recoge los bordes en diagonal con un tamaño de 2x2.

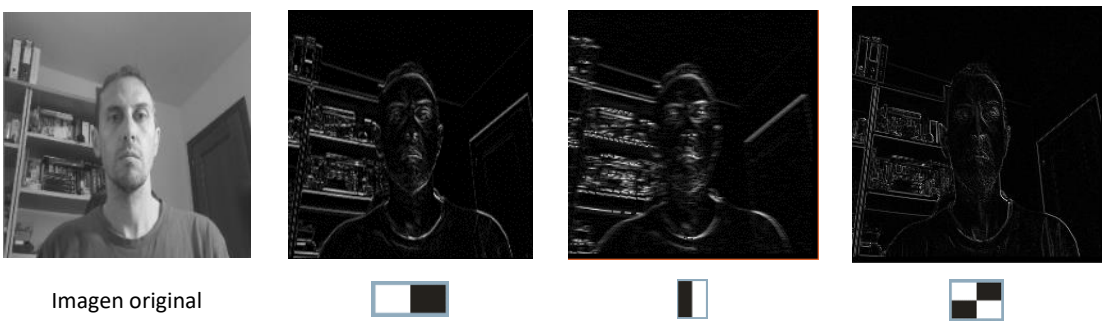


Figura 6: Características de Haar aplicadas a una imagen

Esta operación debe realizarse para todos los tamaños posibles de las características por lo que se debe iterar sobre todos los píxeles de la imagen para cada uno de los posibles tamaños de dicha característica.

Esta iteración con diferentes tamaños de cada característica sobre todos los píxeles de la imagen generará un gran número de características, creciendo estas exponencialmente a medida que crezca el tamaño de la imagen original. Por ello será necesario reducir la imagen hasta un tamaño que genere un número de características aceptables para el algoritmo pero que a la vez permita identificar aún los rasgos propios de una cara.

En el método original de Viola-Jones proponían un tamaño original de 24 por 24 píxeles. De esta forma se comenzaría por la primera característica y se calcularía sobre todos los píxeles de la imagen para un tamaño de 2x1 píxeles. Esto da un total de 23x24=552 posiciones diferentes.

Luego se continuaría con un tamaño de 4x1 píxeles (504 posiciones), 8x1 píxeles (456 posiciones), … Tras alcanzar el tamaño de 24x1 píxeles se pasaría a las características con 2 píxeles de alto y así sucesivamente.

Como se puede ver el número de elementos del vector de características obtenido es muy alto, siendo únicamente para la primera característica de:

$$(23 + 21 + \cdots + 1) * (24 + 23 + \cdots + 1) = 43.200$$

El cálculo de todas las características propuestas por Viola Jones alcanza un número superior a las 180.000 características. Este elevado valor es obviamente un problema, tanto por el coste computacional para calcular cada uno de ellos, como para el posterior proceso de clasificación, que debe trabajar en un entorno n-dimensional con tantas dimensiones como características.

La solución a estos problemas fue el uso de la imagen integral para el cálculo de cada característica, y de AdaBoost como clasificador.

Imagen integral

El cálculo de cada una de las 180.000 características es indudablemente muy costoso computacionalmente ya que para cada una de ellas hay que sumar los valores de todos los píxeles de cada una de las dos partes de la característica. Para soslayar este problema propusieron el uso de una representación intermedia de la imagen denominada **imagen integral**.

La imagen integral tiene el mismo tamaño que la imagen original pero el valor de cada píxel en una posición (x, y) es el valor de la suma de todos los valores de los píxeles que se encuentran sobre y a la izquierda de este.

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

Por ejemplo, en la figura 7 se puede ver como el valor del píxel (4, 3) de la imagen integral se obtiene sumando los valores de todos los píxeles marcados en color más oscuro en la imagen original.

Imagen original

6	2	1	3	6
2	4	1	8	1
1	7	3	2	3
3	5	4	6	2
2	3	5	3	1

Imagen integral

6	8	9	12	18
8	14	16	27	34
9	22	27	**40**	50
12	30	39	58	70
14	35	49	71	84

Figura 7: Creación de la imagen integral

Con la imagen integral es muy fácil obtener el valor de la suma de todos los píxeles de un área de la imagen original $((x_1, y_1), (x_2, y_2))$ ya que simplemente hay que aplicar la fórmula:

$$Suma = (ii(x_2, y_2) - ii(x_2, y_1 - 1)) - (ii(x_1 - 1, y_2) - ii(x_1 - 1, y_1 - 1))$$

En la figura 8 se muestra cómo se calcularía el valor de la suma del área delimitada por $((3, 2), (4, 4))$. Según la fórmula indicada el valor de la suma de todos los píxeles de este rectángulo sería:

$$Suma = \big(ii(4,4) - ii(4,1)\big) - \big(ii(2,4) - ii(2,1)\big) = (58 - 12) - (30 - 8) = 24$$

a) Imagen original

6	2	1	3	6
2	4	1	8	1
1	7	3	2	3
3	5	4	6	2
2	3	5	3	1

1+8+3+2+4+6=24

b) Imagen integral

6	**8**	9	**12**	18
8	14	16	27	34
9	22	27	40	50
12	**30**	39	**58**	70
14	35	49	71	84

58-12-(30-8)=24

Figura 8: Aplicación de la imagen integral

2.3.2.- Histogram of Gradients (HOG)

Aunque HOG se hizo muy popular a raíz del artículo (Dalal & Triggs, 2010), los **Histogramas de Gradientes Orientados** o **HOG** (*Histogram of Oriented Gradients*) ya habían sido formulados mucho antes por Robert K. McConnell (McConnell, 1986).

Gradiente

El concepto clave de HOG es el **gradiente**, que hace referencia a los cambios de intensidad en una imagen. Para ello el gradiente está definido por dos valores, por un lado, la dirección en la que ese cambio de intensidad es máximo y por otro la magnitud del cambio en dicha dirección. Dicho de otra manera, el gradiente va a detectar los contornos de la imagen.

El gradiente de una imagen se puede obtener fácilmente aplicando el filtro de Sobel sobre cada uno de los píxeles de la imagen. En la figura 9 se muestra el resultado de aplicar dicho gradiente a una imagen sobre el eje Y, sobre el eje X y el valor de la magnitud del gradiente.

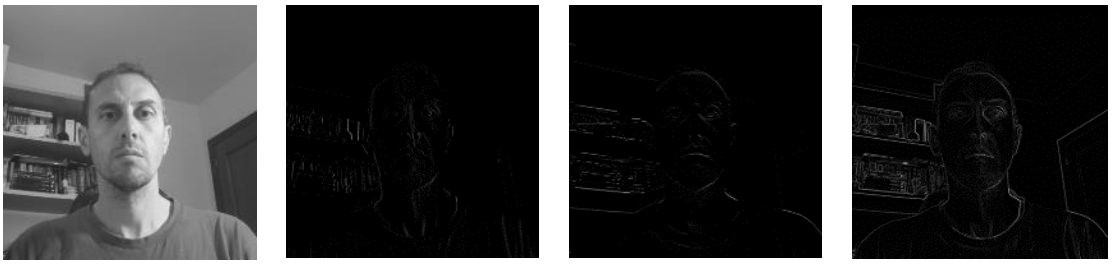


Figura 9: Filtro de Sobel con diferentes gradientes

Como se puede apreciar tiene valores superiores cuando hay cambios bruscos en la intensidad de la imagen, es decir, en los contornos y bordes de esta. En cambio, no refleja ningún valor en las zonas donde hay transiciones suaves. De esta forma estamos eliminando información no relevante de la imagen, como zonas de color constante del fondo, mientras que se mantiene la información de los contornos, que es lo que determina la forma del objeto y va a ayudar a detectarlo.

Histograma de gradientes

En la imagen anterior se calculaba el gradiente para cada uno de los píxeles de la imagen original. Sin embargo, la idea detrás de HOG no es tratar individualmente los gradientes para cada píxel, sino que la información que se extrae es la proporción en que muestra cada dirección posible en los gradientes.

Para ello el primer paso es dividir la imagen en un conjunto de **celdas** de tamaño fijo. Para cada celda se calcula todas las direcciones de los gradientes y se agrupan en un grupo de rangos de orientación.

Para cada celda se calculan los gradientes de todos los píxeles, en concreto la orientación del gradiente en cada píxel, y se agrupan en rangos de direcciones. Habitualmente en las direcciones no se tiene en cuenta el signo, por lo que las orientaciones deben tener un valor entre 0 y 180º. El número de **rangos** es un valor que se debe escoger al implementar el algoritmo. Por ejemplo, si se escogen 9 rangos cada uno de ellos abarcaría los gradientes en un espacio de 20º.

Para cada rango se suman las magnitudes de los gradientes que se encuentran en el mismo. Esto hace que gradientes más pronunciados contribuyan con mayor peso a sus rangos, mientras que el efecto de pequeñas orientaciones aleatorias debidas al ruido será mínimo.

El conjunto de los valores obtenido para cada uno de los rangos de una celda proporciona información sobre la orientación dominante en dicha celda y se denomina **histograma**, siendo este un vector con tantas dimensiones como rangos se hayan escogido.

El descriptor de la imagen es la concatenación de todos los histogramas de todas las celdas de una imagen y contiene una representación de la estructura de dicha imagen, enfatizando en la información relativa a los bordes de la imagen.

Una conclusión a la que llegaron Dalal y Triggs en su artículo, es la necesidad de compensar las variaciones de iluminación y contraste mediante la **normalización**. Los mejores resultados fueron obtenidos cuando se agrupaban varias celdas en un bloque y se realizaba la normalización del contraste de cada bloque por separado. Además, cada bloque se solapa con los bloques adyacentes, por lo que cada celda pertenecerá a varios bloques, contribuyendo por tanto con diferentes descriptores al vector de características, cada uno normalizado con respecto a un bloque diferente.

Para normalizar cada bloque, se divide cada elemento de este por $\sqrt{x_1^2 + x_2^2 + x_3^2 + \cdots + x_n^2}$ donde x_i es cada uno de sus elementos.

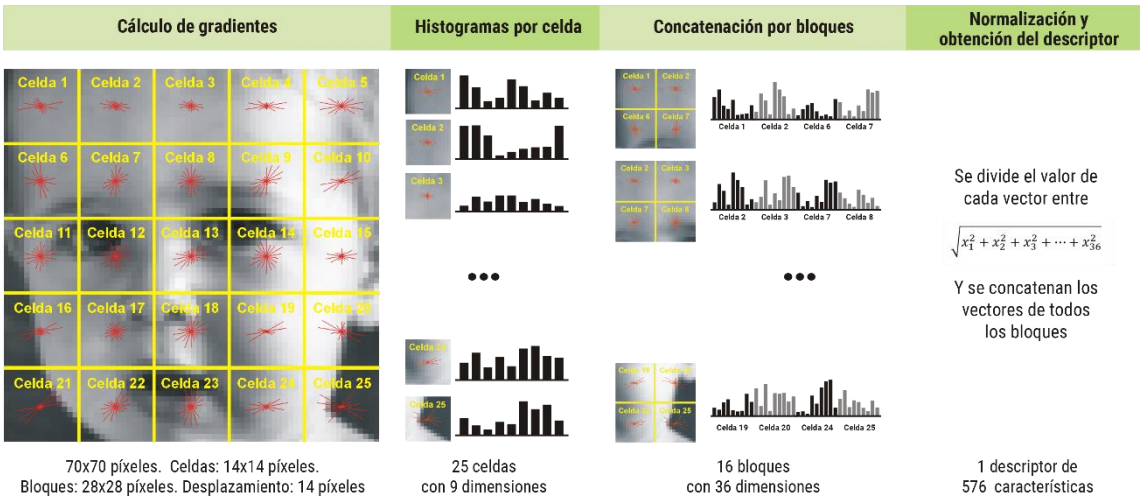


Figura 10: Cálculo del Histograma de Gradientes

Parámetros del histograma de gradientes

Cuando se calcula el histograma de gradientes de una imagen hay que tener en cuenta una serie de parámetros que básicamente determinan cuántos histogramas se calcularán en la imagen y que información contendrán dichos histogramas.

Como se verá más adelante, la correcta elección de estos parámetros influirá notablemente, tanto en la calidad de los resultados obtenidos por el clasificador como el tiempo de cómputo necesario, ya que determinan el número de dimensiones que tendrá el vector de características.

Estos parámetros son:

- **Tamaño de celda**: para el cálculo de HOG la imagen debe dividirse en una serie de celdas del mismo tamaño, y esto es precisamente lo que indica este parámetro. En el caso concreto de este proyecto, dado que se ha trabajado con imágenes cuadradas, las celdas también han de tener el mismo alto que ancho. Su valor ha de ser un divisor del tamaño de la imagen.
- **Tamaño de bloque**: indica el tamaño en píxeles de cada bloque. Dado que cada bloque es un conjunto de celdas, su tamaño deberá ser un múltiplo exacto del tamaño de celda escogido.
- **Desplazamiento de bloque**: este parámetro indicará el grado de solapamiento entre diferentes bloques. De esta forma, el desplazamiento de bloque hace referencia al número de bloques a los que pertenece cada una de las celdas.
- **Número de rangos (nbins)**: indica el número de campos que tendrá el histograma de gradientes. Por tanto, indicará el número de direcciones que se tendrán en cuenta.

2.3.3.- Scale-Invariant Feature Transform (SIFT)

SIFT fue propuesto por David G. Lowe en su artículo *Distinctive Image Features from Scale-Invariant Keypoints* (Lowe, 2004) y propone un método de detección de **puntos de interés** en una imagen. Su principal ventaja frente a otros detectores de puntos de interés en una imagen es que es invariante frente a cambios de escala, orientación e iluminación, lo que le hace bastante útil para su uso en sistemas de reconocimiento de objetos.

Una característica SIFT es una región de la imagen (o punto de interés) con un descriptor asociado y son obtenidos mediante un proceso de cuatro fases:

- **Detección en diferentes escalas**: esta primera fase itera sobre todas las escalas y localizaciones de la imagen buscando potenciales puntos de interés.
- **Localización de puntos de interés**: se analizan los candidatos obtenidos en el punto anterior y se seleccionan los más adecuados.
- **Asignación de orientación**: a cada punto se le asigna una orientación basada en las direcciones del gradiente local. Esta orientación y la escala servirán como punto de referencia para posteriores operaciones, lo que proporciona invariancia frente a estas transformaciones.
- **Descriptor del punto de interés**: para cada punto de interés se calcula un descriptor en base a los gradientes locales de la imagen.

Por lo tanto, SIFT localiza una serie de puntos de interés identificados por una posición, un tamaño y un ángulo, y cada uno de ello tiene asociado un descriptor que es un histograma de los gradientes de la imagen que caracteriza el punto de interés.

En la figura 11 se muestran los puntos de interés localizados en una cara representando gráficamente sus características geométricas. A la derecha, se muestra un ejemplo de la información que asocia SIFT a cada punto de interés: su ubicación, ángulo y tamaño, así como el descriptor obtenido a partir de los gradientes en torno al punto de interés y representado por un vector de 128 dimensiones.

```
EJEMPLO DE INFORMACIÓN ASOCIADA A UN KEYPOINT
==============================================
Punto de interés SIFT
---------------------
Ángulo:   89.69537353515625
Posición: (10.447190284729004, 135.8819122314453)
Tamaño:     3.137603521347046
Descriptor SIFT
---------------------
[  5.   2.   1.   7.   6.   1.   1.   1.  12.  12.   7.  86. 108.  10.
   2.   1. 105.  59.  11.  32.  40.   3.   0.   2.  20.   5.   0.   8.
  80.   3.   0.   0.   1.   2.   2.  22.  31.   0.   0.   0.  37.  15.
   6. 139. 139.   9.   1.   3. 139.  42.   6.  33.  46.   5.   1.  11.
  38.   1.   0.   1. 113.  11.   0.   2.   0.   0.   1.  24.  45.  15.
  16.   3.  29.   3.   0.  27. 139.  92.  24.  25. 139.  11.   1.   3.
  32.  21.  15.  88.  25.   2.   0.   1. 106.  10.   0.   4.   7.   0.
   0.   1.   7.  37. 100.  75.  12.   1.   0.   2.  56. 139.  80.  10.
  76.  49.  39.  12.  30.  48.   8.  12.   9.  20.  20.  17.  55.   2.
   0.   0.]
```

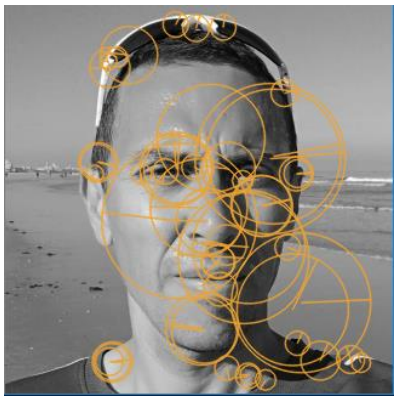
Figura 11: Características SIFT

2.3.4.- Speedup Robust Features (SURF)

Aunque SIFT funciona bastante bien para la identificación de puntos de interés de una imagen, adolece de cierta lentitud. Por ello, en el año 2006 fue propuesta una alternativa más rápida por parte de Herbert Bay, Tinne Tuytelaars y Luc Van Gool en su artículo titulado "*SURF: Speeded Up Robust Features*" (Bay, Tuytelaars, & Van Gool, 2006).

Ambos métodos difieren en el método utilizado para calcular los puntos de interés, pero la información asociada a cada punto sigue siendo la misma: los datos relativos a su ubicación, escala y dirección, así como el descriptor de este.

En la figura 12 se muestran los puntos de interés detectados por SURF en una imagen, así como un ejemplo de los datos de uno de los puntos de interés.



```
EJEMPLO DE INFORMACIÓN ASOCIADA A UN KEYPOINT
==============================================
Punto de interés SURF
---------------------
Ángulo:   272.29278564453125
Posición: (142.05831909179688, 134.82662963867188)
Tamaño:     56.0
Descriptor SURF
---------------------
[ 0.00116367  0.00145976  0.00132681  0.00327048 -0.00783345  0.0123337
  0.01218348  0.02803774 -0.00584916  0.01691778  0.00942851  0.02851159
 -0.00068865  0.00249411  0.00105029  0.006025    0.01345154 -0.01709726
  0.01933888  0.0265591  -0.32867646 -0.15050241  0.33741203  0.17285295
  0.27834767 -0.11433566  0.31538415  0.14967084 -0.00245498 -0.00325099
  0.00843248  0.00644783  0.00948225 -0.00134145  0.02933728  0.03359917
 -0.26416698 -0.00540373  0.29343078  0.36355647  0.21568418  0.03678585
  0.27425972  0.29177094  0.00193791 -0.00569832  0.00583547  0.00981821
  0.00046251 -0.00756162  0.00403252  0.00938872 -0.02188584  0.00314111
  0.03420823  0.05278638  0.04360262 -0.02861866  0.04652838  0.03135816
 -0.0009943  -0.0038088   0.00123949  0.0038088 ]
```

Figura 12: Características SURF

2.4.- Fundamentos algorítmicos: clasificadores

El objetivo de los algoritmos que se han visto hasta ahora es obtener un descriptor de la imagen, el cual contiene la información relevante de la misma descartando aquella que no tenga importancia a la hora de determinar si se trata de una cara o no. Este descriptor se puede representar como un vector n-dimensional con tantas dimensiones como características tenga el descriptor.

Partiendo de ese punto, es razonable pensar que las imágenes similares tengan ubicaciones próximas en este espacio n-dimensional, en concreto, para el caso del presente proyecto, es

de suponer que todas las imágenes que corresponden a caras se encuentren en ubicaciones más o menos próximas dentro de este espacio.

La función de los **algoritmos de clasificación** consiste en determinar si una imagen es o no es una cara en función de su ubicación dentro de ese espacio. Para ello parten de un conjunto de entrenamiento, que incluye cientos o miles de imágenes de caras y de no caras, y, basándose en esos datos, utilizan diferentes métodos para predecir si una nueva imagen se trata de una cara o no.

Los algoritmos utilizados en este proyecto son:

- **AdaBoost**: se basa en crear un clasificador fuerte, que identifique correctamente la clase a la que pertenece una imagen, partiendo de múltiples clasificadores débiles, es decir, que tengan una tasa de acierto solo ligeramente superior a una clasificación aleatoria.
- **Support Vector Machines**: este algoritmo busca dividir el espacio en dos partes, de forma que cada una de las clases (cara o no cara) se encuentre en una parte diferente, así, la ubicación de una nueva imagen determinará si se trata de una cara o no.
- **k-Nearest Neighbours**: el principio detrás de este algoritmo es que una imagen es de la misma clase que las imágenes que se encuentran más próximas a ella en el espacio.

2.4.1.- Algoritmo de Viola-Jones: AdaBoost

El algoritmo **AdaBoost** (*Adaptative Boosting*) fue formulado por Yoav Freund y Robert Schapire en su artículo titulado *Experiments with a New Boosting Algorithm* (Freund & Schapire, 1996) y es ampliamente conocido por ser clasificador utilizado en el algoritmo de Viola-Jones.

La idea detrás de AdaBoost es crear un clasificador fuerte a partir de múltiples clasificadores débiles. Se puede definir un **clasificador fuerte** como aquel que tiene una alta tasa de aciertos, mientras que un **clasificador débil** es simplemente un clasificador que tiene un desempeño bastante pobre, pero que es ligeramente mejor que realizar una elección aleatoria.

AdaBoost tiene algunas similitudes con un algoritmo conocido como bosques aleatorios (*random forests*), pero hay tres diferencias fundamentales que le diferencian:

- Los clasificadores débiles utilizados en AdaBoost son árboles de decisión con un único nodo y dos hojas. Este tipo de árboles, cuya clasificación se basa en una única característica, se denominan **tocones de decisión** (*decision stumps*).
- Mientras que en los bosques aleatorios todos los árboles tienen el mismo peso en la decisión final, en AdaBoost cada tocón de decisión tiene un peso diferente en función de su relevancia respecto a la clasificación.
- Cada uno de los árboles de un bosque aleatorio se calcula de forma independiente al resto de árboles. Sin embargo, en AdaBoost la construcción de cada uno de los tocones de decisión se realiza en función del resultado de los tocones calculados anteriormente.

En la figura 13 se expone un ejemplo de funcionamiento de este algoritmo en un espacio bidimensional.

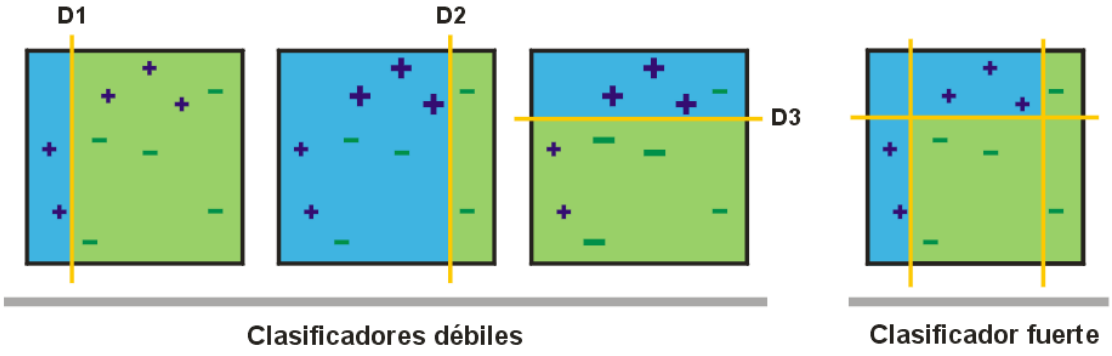


Figura 13: AdaBoost en espacio bidimensional

En este ejemplo, hay una serie de elementos con dos características, y que por tanto se pueden mostrar en un espacio bidimensional. Estos elementos, utilizados para el entrenamiento del clasificador, pueden ser de dos clases, identificadas por los símbolos + y -.

Los pasos que se realizarían para el entrenamiento son:

- Se asigna un mismo peso a todos los puntos y se aplica el tocón de decisión para clasificarlos, generando la línea etiquetada como D1. Claramente se puede ver que esta clasificación no es muy buena, ya que, aunque clasifica correctamente algunos puntos, hay tres puntos que son incorrectamente clasificados.
- En una segunda iteración, se asigna un mayor peso a los puntos incorrectamente clasificados con el clasificador anterior, y se vuelve a calcular el tocón de decisión, generando una nueva línea (D2). Esta nueva clasificación sigue sin ser buena, ya que ahora hay otros tres puntos que no son correctamente clasificados.
- Se repite el proceso anterior, se asigna a los puntos incorrectamente clasificados un mayor peso, y se vuelve a calcular el tocón de decisión, lo que genera la línea D3.
- Los clasificadores que se han obtenido hasta ahora se consideran **clasificadores débiles**, ya que ninguno realiza una buena clasificación. Sin embargo, en el cuarto punto se combinan estos clasificadores débiles para crear un clasificador fuerte que identifica correctamente todos los puntos.

2.4.2.- Support Vector Machines (SVM)

El algoritmo SVM busca realizar la clasificación a través de la búsqueda de un **hiperplano** que separe en dos el espacio n-dimensional, de forma tal que todas las imágenes que correspondan a caras se encuentren a un lado de este, y las imágenes que no sean caras se encuentren al otro lado.

Para encontrar este hiperplano es necesario un proceso de entrenamiento en el que se surte al algoritmo con una gran cantidad de imágenes etiquetadas como caras o como no caras.

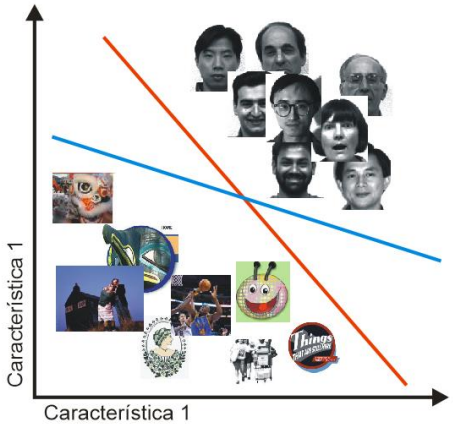


Figura 14: Clasificador SVM

Pero, como se aprecia en la figura 14, este hiperplano no es único ya que puede haber múltiples planos que dividan con éxito el conjunto de entrenamiento. El objetivo será encontrar el hiperplano óptimo, y para eso SVM se apoya en los denominados **vectores de soporte**.

Los vectores de soporte son muestras que se escogen del conjunto de entrenamiento y sirven como referencia para escoger el hiperplano de separación entre clases, que será aquel cuya distancia sea máxima con respecto a los hiperplanos que contienen los vectores de soporte de ambas clases. Esto se resuelve como un problema de optimización que se resuelve mediante los multiplicadores de Lagrange.

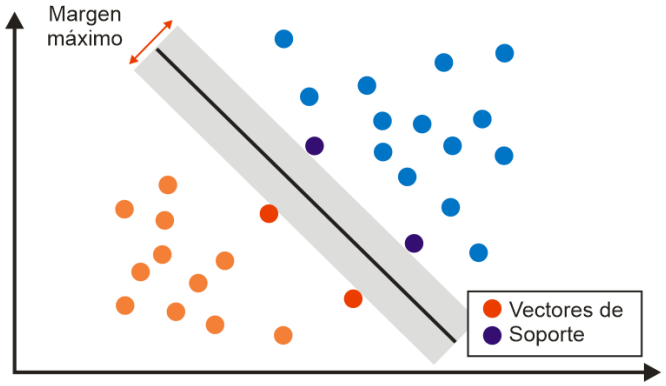


Figura 15: Vectores de soporte en SVM

Un problema que puede surgir con SVM es la posibilidad de que ambos clústeres no sean linealmente separables, es decir, que no hay ningún hiperplano que separe ambos conjuntos. Para solucionar este problema hay dos alternativas:

- Margen suave
- Kernel trick

El **margen suave** consiste en relajar la condición del margen, es decir, se añade una tolerancia a errores que permite que algunos elementos de una clase no estén en la parte del espacio que les corresponde.

La otra posibilidad es el denominado **kernel trick**. Este mecanismo es útil cuando los vectores o puntos que corresponden a una clase están *rodeados* por los de la otra clase, de forma que, aunque se relajen las condiciones de margen sería imposible encontrar un hiperplano que los separe. La solución en este caso es añadir una nueva dimensión al espacio y, para cada punto, asignar un peso en esa nueva dimensión que siga una distribución gaussiana cuyo valor máximo esté en el centro de la clase que se encuentra *rodeada*.

En la figura 16 se puede ver un ejemplo en un espacio unidimensional con una única característica. Como se puede ver se ha añadido una nueva dimensión, pasando a ser un espacio bidimensional y se han distribuido los puntos en esta nueva dimensión acorde a su distancia con respecto al centro de la clase coloreada de color naranja. Una vez hecho esto ya es posible encontrar una línea que

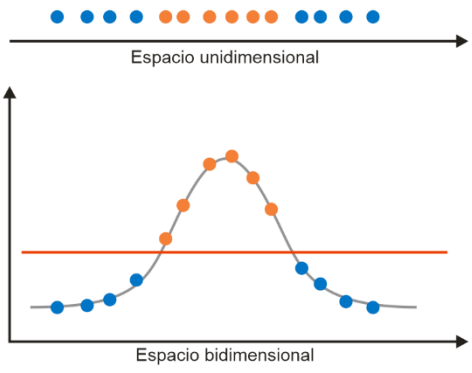


Figura 16: Kernel trick de SVM

separe ambas clases.

Parámetros de SVM

Al utilizar SVM hay una serie de parámetros que necesitan ser ajustados para obtener el mejor rendimiento del clasificador. Estos parámetros son:

- **Tipo de kernel**: determina el tipo de hiperplano que se utilizará para dividir el espacio. Los tipos de kernel que permite la librería OpenCV son: lineal, poligonal, Radial Basis Function (RBF) y sigmoideo. En la figura[1] 17 se muestran algunos de estos tipos.
- **Gamma**: este parámetro se aplica a kernels no lineales, y determina la precisión con que el hiperplano se ajustará al conjunto de datos de entrenamiento. Cuando mayor sea el valor, más se ceñirá el hiperplano a los datos.

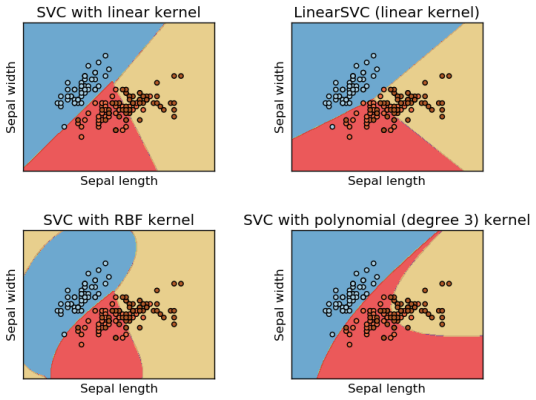


Figura 17: Kernels de SVM

- **C**: determina la penalización por los elementos incorrectamente clasificados. Un valor bajo proporcionará un hiperplano suave aún a pesar de clasificar incorrectamente algunos datos de entrenamiento, un valor alto se ajustará mucho mejor a los datos de entrenamiento, pero puede provocar un sobreajuste del clasificador.
- **Grado**: utilizado en los kernels poligonales, determina el grado del polinomio que representa el hiperplano utilizado.

2.4.2.- k-Nearest Neighbours

KNN (*K-Nearest Neighbours*) es un algoritmo de aprendizaje supervisado que puede ser utilizado tanto para resolver problemas de clasificación como de regresión. Su principal ventaja es que se trata de un algoritmo muy sencillo y fácil de comprender, mientras que el mayor inconveniente es que es computacionalmente costoso y tiene un alto consumo de memoria, especialmente con conjuntos de entrenamiento muy grandes y con muchas dimensiones, ya que necesita almacenar todos los datos de entrenamiento.

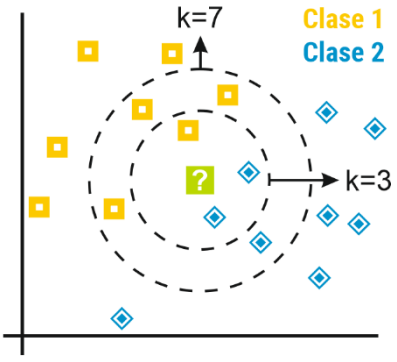


Figura 18: K-Nearest Neighbours

El funcionamiento de kNN es muy sencillo, parte de un espacio n-dimensional donde se ubican espacialmente todos los vectores de características del conjunto de entrenamiento. Estos vectores estarán etiquetados como pertenecientes a caras o no caras. Cuando se quiere

[1] Fuente de la imagen: https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769

determinar si una imagen pertenece a una cara o no, se calcula su vector de características y se buscan los k puntos más cercanos a él. El tipo de categoría al que pertenezcan la mayoría de estos puntos será el que decida si la imagen analizada pertenece a una cara o no.

Un elemento importante es la elección del valor de k, que es el único parámetro que requiere este algoritmo y cuya elección puede tener como consecuencia algunos problemas que se representan en la figura 19.

- Un valor muy bajo tomará pocos puntos como referencia para determinar el tipo al que pertenece. Esto puede suponer un problema si hay un cierto ruido en el sistema y hay algunos elementos de un tipo mezclados en el espacio donde se sitúan los elementos del otro tipo.
- Un valor alto, por el contrario, supondrá que se tienen en cuenta muchos más elementos para determinar el tipo. En este caso, el inconveniente se puede apreciar en la imagen de la derecha, donde los elementos de un tipo están agrupados en una región determinada del espacio, mientras que los del otro tipo están mucho más dispersos.

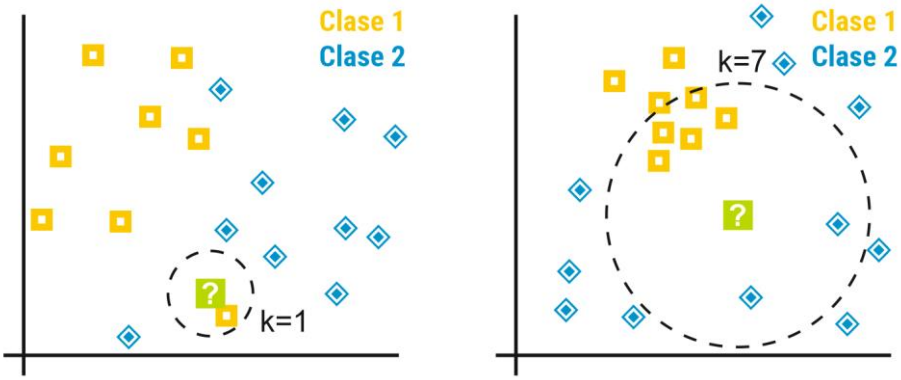


Figura 19: Problemas de kNN

2.5.- Fundamentos algorítmicos: reconocimiento facial

El otro conjunto de algoritmos que es necesario conocer son los utilizados para el proceso de reconocimiento facial. Los algoritmos que se han utilizado en el presente proyecto son:

- **Eigenfaces**: basado en una reducción de la dimensionalidad de las imágenes mediante la técnica denominada *Principal Component Analysis* (PCA).
- **Fisherfaces**: de forma análoga a Eigenfaces, también busca una reducción del número de dimensiones del descriptor, pero en este caso usando *Linear Discriminant Analysis* (LDA).
- **LBP**: un enfoque diferente a los anteriores que obtiene el descriptor de la imagen mediante histogramas de los denominados *Patrones Locales Binarios* (LBP).

2.5.1.- Eigenfaces

El problema que hay con la representación de una imagen, y por tanto su tratamiento, es la elevada dimensionalidad de esta. Por ejemplo, la representación vectorial de una imagen de

480x320 píxeles supone un vector en un espacio de 153.600 dimensiones. Sin embargo, no todas son igualmente relevantes, conteniendo más información aquellas dimensiones que tienen una mayor varianza en los datos, entendiéndose la varianza como una medida de la dispersión de los datos.

Teniendo lo anterior en cuenta, sería posible descartar aquellas dimensiones con una menor varianza, manteniendo únicamente aquellas que contienen información relevante. Esto es precisamente los que persigue la técnica denominada **Principal Component Analysis (PCA)**. Fue desarrollada por Pearson (1901) y Hotelling (1933) y su objetivo es reducir la dimensionalidad de datos multivariable preservando toda la información posible. PCA es una transformación lineal que transforma los datos a un nuevo sistema de coordenadas de forma tal que el nuevo conjunto de coordenadas, denominadas los *componentes principales*, son funciones lineales de las variables originales, son no correlacionadas, y la mayor varianza de los datos se encuentra en la primera coordinada, la segunda mayor varianza en la segunda coordenadas, y así sucesivamente.

El concepto detrás de PCA se puede ver más claramente en la imagen de la figura[2] 20, donde hay una serie de elementos dispuestos en un espacio bidimensional en la que los ejes representan el alto y el ancho de dichos elementos. La idea con PCA es cambiar el eje de coordenadas para que esté alineado en la dirección con mayor varianza, en la imagen representado por la línea azul.
Esta línea determinará ahora la componente principal de la imagen, mientras que la línea azul punteada será la segunda componente principal. Se puede ver fácilmente que se podría descartar esta segunda dimensión y apenas se perdería información relevante de los datos.

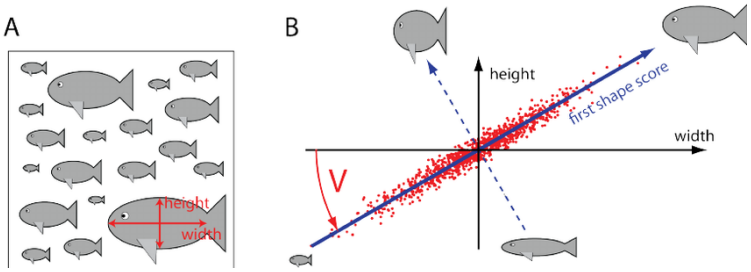


Figura 20: Cambio de espacio dimensional en PCA

2.5.2.- Fisherfaces

El método Principal Component Analysis (PCA) busca un conjunto de características que maximiza la varianza total de los datos, pero tiene el inconveniente de que no toma en consideración todas las clases y puede haber información relevante que descarta.

El fundamento detrás de Fisherfaces es el **Análisis Discriminante Lineal (LDA)**, desarrollado por el estadista Sir R. A. Fisher en 1936, busca la combinación de características que mejor diferencien las clases aumentando la distancia entre ellas y reduciendo la dispersión entre los elementos de una misma clase. Básicamente, la idea es que, dentro del espacio con menos dimensiones, los elementos de una misma clase se encuentren muy próximos, a la vez que se encuentran lo más alejados posible de los elementos de otras clases.

[2] Fuente de la imagen: https://www.researchgate.net/figure/Illustration-of-principal-component-analysis-A-As-a-minimal-example-we-consider-a_fig1_263968032

2.5.3.- Local Binary Patterns (LBP)

Este es uno de los métodos más sencillos y fáciles de comprender que se pueden aplicar al reconocimiento facial. Su origen se remonta al año 1994 y es un tipo de descriptor visual que ha demostrado tener muy buenos resultados para la clasificación de texturas, al margen de su uso para el reconocimiento facial.

Para utilizar LBP para el reconocimiento facial hay que realizar los siguientes pasos:

- **Creación de la imagen con LBP**: la idea que subyace tras LBP es obtener una representación alternativa de la imagen donde el valor de cada píxel está determinado por los valores de los píxeles adyacentes, para ello hay que hacer lo siguiente:
    - Para cada píxel de una celda, se compara con los 8 píxeles vecinos, comenzando siempre en la misma posición (por ejemplo, el píxel que se encuentra encima) y siguiendo siempre el mismo sentido (dextrógiro o levógiro).
    - Se compara el valor del píxel con el del vecino que corresponda. Si el valor del píxel es mayor que el del vecino se le asigna un 0, en caso contrario se asigna un 1.
    - El paso anterior generará una secuencia de 8 dígitos binarios (0 o 1) para cada píxel, uno por cada vecino, que se puede representar por tanto como un número decimal comprendido entre 0 y 255.
- **Cálculo del descriptor de la imagen**: a partir de la imagen anterior se calcula el histograma de la imagen, para ello:
    - Se divide la imagen en celdas.
    - Para cada una de las celdas, se calcula el histograma a partir de los valores de los píxeles que haya en dicha celda. Cada histograma tendrá 256 posiciones representando cada uno de los 256 posibles valores que puede tener una celda. Opcionalmente, se podrían normalizar dichos histogramas.
    - Finalmente, se concatenan todos los histogramas de todas las celdas para obtener el descriptor de la imagen. Por ejemplo, para una división de 8x8 celdas, se obtendría un vector de 16.384 dimensiones (8x8x256).
- **Reconocimiento facial**: tras realizar el entrenamiento donde se ha calculado el descriptor de un conjunto de imágenes ya etiquetadas, se realizan los siguientes pasos:
    - Se calcula el histograma de la imagen que se quiere reconocer.
    - Se compara el histograma de la imagen con los histogramas de las imágenes de entrenamiento. Para compararlos se mira la distancia entre ellos, por ejemplo, la distancia euclídea.
    - La salida del algoritmo será el identificador del histograma más próximo al de la imagen a reconocer, siempre y cuando esta distancia sea inferior a un valor umbral. Si no hay ningún histograma con distancia inferior al valor umbral se etiquetará la imagen como no reconocida.

3.- Diseño de la solución

3.1.- Análisis de las herramientas

3.1.1.- Sistema operativo

Aunque es posible instalar en la Raspberry Pi la versión para ARM utilizada en *tablets* de Windows 10, el sistema operativo natural para este dispositivo es Linux. Hay varias distribuciones preparadas para instalarse en la Raspberry, pero se ha optado por utilizar **Raspbian**, la distribución oficial de esta plataforma y que está basada en Debian.

3.1.2.- Lenguaje de programación

Uno de los puntos de partida en el desarrollo del presente proyecto es la selección del lenguaje de programación utilizado para la implementación de este. La principal librería utilizada, **OpenCV**, está desarrollada en C++, pero puede ser utilizada desde diversos lenguajes de programación: C/C++, Python, Java, JavaScrip, Matlab, …

El lenguaje finalmente escogido ha sido **Python**, los motivos para esta elección han sido los siguientes:

- Es un lenguaje conciso y claro, con una curva de aprendizaje bastante moderada pero que no le quita potencia al lenguaje. Permite una programación muy sencilla y rápida, pero, si es necesario, también dispone de constructores más avanzados permitiendo, por ejemplo, programación concurrente o un fácil acceso al hardware de la máquina subyacente.
- Es el lenguaje recomendado por la *Raspberry Pi Foundation*.
- Dispone de gran número de *frameworks* y librerías, lo que simplifica mucho el desarrollo de las aplicaciones. Ejemplos de estas librerías pueden ser Matplotlib, para la representación de gráficas de datos o la librería *thread* para la programación concurrente con múltiples hilos.
- En los últimos años se ha destacado como uno de los lenguajes más utilizados en el campo del *machine learning*, lo que ha propiciado la aparición de librerías específicamente dedicadas a este campo y que serán de gran utilidad en el presente proyecto. Ejemplos de estas librerías son Numpy para optimizar el tratamiento numérico de grandes cantidades de datos, Scikit-learn o el propio OpenCV, que implementan gran cantidad de algoritmos de *machine learning*.
- A pesar de ser un lenguaje interpretado, no tiene una excesiva penalización de tiempo de ejecución respecto a otros lenguajes compilados. Esto se debe principalmente a que las tareas del presente proyecto que pueden sufrir una penalización por sus excesivos requisitos computacionales recaen sobre librerías, tales como Numpy, que están desarrolladas en C++.

  Además, la posible pérdida de rendimiento por ser un lenguaje interpretado es compensada ampliamente por la mejora en los tiempos de desarrollo, ya que facilita en gran manera la mantenibilidad y calidad del código.

- Es totalmente independiente de la plataforma, el código puede ser portado entre diferentes arquitecturas y plataformas sin necesidad de realizar ninguna modificación en el mismo. Esto es importante porque, aunque el código final se ejecutará sobre la Raspberry Pi 3, no será esta la plataforma más adecuada para la fase de desarrollo. Esto se debe principalmente a sus limitaciones de hardware y rendimiento, por lo que el desarrollo se realizará sobre una máquina con arquitectura Intel y sistema operativo Windows o Linux.
- Por último, Python es uno de los lenguajes de programación que mayor crecimiento está experimentando en los últimos años, no solo en el campo del *machine learning*, sino también en otros ámbitos como el desarrollo Web o como herramienta de *scripting* en administración de sistemas. Esto ha motivado que a nivel personal considerar interesante profundizar en el conocimiento de este lenguaje.

3.1.3.- Librerías utilizadas

La madurez que ha alcanzado el campo de la visión por computador y la creciente proliferación de aplicaciones que se basan en técnicas de visión por computador ha permitido la aparición de múltiples librerías implementando algoritmos para el tratamiento y la manipulación de imágenes. De todas las librerías disponibles en el presente proyecto se han utilizado OpenCV y Dlib.

OpenCV

El proyecto OpenCV fue iniciado en enero del año 1999 como parte de una iniciativa de Intel para proporcionar una infraestructura común para aplicaciones de visión por computador y para acelerar la utilización de sistemas de visión artificial en aplicaciones comerciales.

El promotor de este proyecto fue Gary Bradski, perteneciente al *Intel's Perfomance Library Team*, al que posteriormente se le unió un grupo de expertos en optimización de Intel en Rusia liderados por Vadim Pisarevsky.

Tras la primera versión Alpha, publicada en el año 2000, aparecieron hasta cinco versiones beta que culminaron con la publicación de la versión 1.0 en el año 2006. La versión 2.0 se publicó en octubre de 2009 con importantes cambios centrados en una interfaz más amigable, nuevas funciones y una mejor implementación de las existentes en términos de rendimiento.

En agosto de 2012 se fundó la organización sin ánimo de lucro **OpenCV.org** para el desarrollo y control de la librería. Tres años más tarde, en junio de 2015 se publicó la versión 3.0 centrada en nuevas funciones, más velocidad y una mayor estabilidad.

En la actualidad cuenta con una comunidad de 47.000 usuarios con un número de descargas calculado en torno a los 18 millones. Esto hace que el desarrollo de la librería avance a buen ritmo, con una versión menor aproximadamente cada seis meses. La última versión es la versión 4.1.1 que fue liberada en julio de 2019.

En su última versión, OpenCV tiene implementados aproximadamente 2500 algoritmos de visión por computador y aprendizaje máquina. Estos algoritmos abarcan aspectos tan variados como identificación de objetos, seguimientos de movimientos, extracción de modelos 3D de objetos, eliminación de ojos rojos, reconocimiento de escena para establecer marcadores para superponerlos con realidad aumentada, etc.…

De forma paralela a OpenCV, hay disponible un repositorio denominado **opencv_contrib**, que contiene módulos que proporcionan una funcionalidad extra a OpenCV. Estos módulos no suelen disponer de una API estable y tampoco están tan probados como los del repositorio principal, pero en general proporcionan un rendimiento y estabilidad aceptables, por lo que son aptos para su uso.

Por norma general, todos los módulos nuevos son publicados en primer lugar en este repositorio y, cuando han alcanzado madurez y popularidad, son pasados al repositorio principal. Por ello, si se desea disponer de toda la funcionalidad de OpenCV es conveniente instalar también este repositorio.

Un ejemplo de módulo incluido en el repositorio opencv_contrib es el que implementa el algoritmo SIFT, utilizado en el presente proyecto. Este módulo en concreto fue finalmente retirado debido a que se trata de un algoritmo patentado, siendo la versión 3.4.2.17 del repositorio contrib la última que lo implementa. Por ello, en el este proyecto se utilizará la **versión 3.4.2 de OpenCV**.

Dlib

Dlib es una librería de código abierto escrita en C++ y disponible para múltiples plataformas. Desde su nacimiento en el año 2002 por obra de Davis E. King, ha crecido enormemente para incluir componentes de software relacionados con redes, hilos, interfaces gráficas, álgebra lineal, procesamiento de imágenes, minería de datos, optimización numérica, … En los últimos años gran parte de su desarrollo se ha centrado en el aprendizaje máquina, implementando los algoritmos más populares de esta ciencia.

Al igual que OpenCV está desarrollado en C++, pero dispone de una API para su utilización desde Python, permitiendo un entorno de trabajo en el que se combina la velocidad del código desarrollado en C++ con la versatilidad y agilidad de desarrollo de Python.

Numpy

La librería Numpy fue desarrollada en el año 2005 por Travis Oliphant tomando como punto de partida la librería de operaciones matemáticas Numeric, a la que añadió soporte para grandes matrices.

El elemento principal de Numpy es el `ndarray`, una estructura de datos que representa un array n-dimensional y cuyo objetivo es proporcionar una alternativa a las listas de Python para las operaciones con vectores y matrices de una forma más rápida y optimizada. El motivo de esto se debe a la ineficiencia de acceso a las listas de Python, principalmente debido a su carácter dinámico y a su capacidad de almacenar elementos de diferentes tipos.

Estos problemas son solucionados en los *ndarrays*, cuyo tipado homogéneo permite una organización en memoria con un acceso mucho más rápido.

3.1.4.- Entorno de desarrollo

A pesar de que la arquitectura objetivo para el presente proyecto es la arquitectura ARM de la Raspberry Pi, se ha optado por aprovechar las capacidades multiplataforma de Python para

realizar el proceso de desarrollo del proyecto sobre una máquina Windows con arquitectura Intel, aunque las pruebas se han realizado directamente sobre la Raspberry Pi.

Con idea de sincronizar ambas máquinas y agilizar el proceso de codificación, se ha optado por utilizar un sistema de **control de versiones Git**. Este software, junto con un repositorio remoto alojado en bitbucket.org, permite sincronizar el código prácticamente en tiempo real entre ambas máquinas, lo que ha facilitado mucho el proceso de desarrollo.

En cuanto al entorno local de cada una de las máquinas, en el Anexo I de la presente memoria se indican los pasos necesarios para instalar todo el software necesario en la Raspberry Pi, desde el sistema operativo hasta el intérprete Python y todas las bibliotecas utilizadas en este proyecto. Este proceso, que puede parecer trivial, reviste mayor número de dificultades de las que se podrían esperar. Estas dificultades provienen del hecho de que la librería OpenCV ha de ser compilada a partir del código fuente en C++, al igual que Dlib, que también debe ser compilada.

El problema encontrado en este punto ha sido la escasa memoria RAM de la Raspberry Pi (1 GB), insuficiente para realizar el proceso de compilación. Por ello, tal como se explica en el citado anexo, ha sido necesario realizar una serie de operaciones para maximizar el tamaño de la memoria RAM disponible. Estas operaciones han consistido en aumentar el tamaño asignado a la memoria de intercambio o *swap*, configurar el arranque para no cargar el entorno gráfico o reducir el tamaño asignado para memoria de vídeo, la cual se extrae de la memoria RAM.

Otra cuestión importante en la configuración del entorno de desarrollo ha sido la selección de la versión de las librerías utilizadas, especialmente OpenCV. Periódicamente se publican nuevas versiones que añaden o eliminan funcionalidades, por lo que durante el proceso de implementación del proyecto se han probado diferentes versiones buscando aquella que mejor se adapte a las necesidades de este. Para facilitar esta tarea, se han utilizado los **entornos virtuales** de Python, que permiten tener instaladas diferentes conjuntos de bibliotecas en distintos entornos aislados sin crear ningún tipo de conflicto.

Por otro lado, en la máquina Windows, la creación del entorno ha sido bastante más sencilla, ya que se ha recurrido a **Anaconda**, una suite de código abierto orientada a la ciencia de datos con Python que incluye prácticamente todo lo necesario para configurar el entorno de trabajo. Además, dispone gestión de diferentes entornos, equivalentes a los entornos virtuales de Python, para trabajar con diferentes conjuntos de versiones de bibliotecas.

3.1.5.- Hardware

En el desarrollo de este proyecto se han utilizado dos placas diferentes, la Raspberry Pi 3 Modelo B y la Modelo B+. Como se puede ver en el apartado 2.1, ambas son prácticamente idénticas en cuanto a características, siendo la diferencia más destacable el ligero aumento de la frecuencia del procesador del último modelo, en concreto pasa de 1.2 GHz a 1.4 GHz. En teoría, este aumento de frecuencia proporciona un aumento del rendimiento del 16% a costa de un pequeño aumento en el consumo energético y un mayor calentamiento general de la placa.

Para la recepción de imágenes de vídeo, se ha utilizado la cámara oficial de Raspberry Pi, cuyas características ya se han expuesto en el apartado 2.1.

Figura 21: Medidor UM24C de Kkmoon

Para el análisis del consumo energético sobre la Raspberry Pi se ha optado por utilizar un medidor de corriente USB de la casa Kkmoon, concretamente el modelo **UM24C**. Este dispositivo, conectado entre la fuente de alimentación USB y la propia Raspberry Pi, proporciona los valores de la intensidad, voltaje y potencia de la corriente suministrada al dispositivo.

La diferencia de este dispositivo con otros disponibles en el mercado es que, mientras que la mayoría de los medidores de corriente proporcionan únicamente información instantánea, el UM24C almacena los datos y permite exportarlos a un teléfono móvil o PC mediante su conexión Bluetooth, facilitando así la generación de gráficas con la evolución de estos valores en el tiempo.

3.2.- Arquitectura software

Dado que el objetivo del presente proyecto es probar diferentes métodos y algoritmos para implementar un sistema de reconocimiento facial viable para la Raspberry Pi, se ha considerado que lo deseable sería implementar un sistema modular, maximizando la cohesión y minimizando el acoplamiento entre los diferentes módulos, a la vez que se favorece la reutilización del código. Para ello, se ha optado por la programación orientada a objetos, encapsulando en un objeto cada uno de los bloques funcionales de un sistema de reconocimiento facial.

A continuación, se exponen las principales clases implementadas, con una somera descripción de su funcionamiento. En el Anexo III se incluye el diagrama de clases completo.

3.2.1.- La clase Provider

Establece una interfaz común para todas las clases encargadas de alimentar con imágenes el detector de caras. Independientemente de cuál sea el origen de las imágenes, todas las clases hijo implementan un iterador que en cada iteración devuelve una imagen obtenida de la fuente correspondiente.

Las clases hija que se han implementado son:

- `ProviderWebcam`: las imágenes son obtenidas de la webcam del ordenador o bien de la cámara de la Raspberry Pi.
- `ProviderVideo`: las imágenes se obtienen de un vídeo previamente grabado.
- `ProviderDataset`: el origen son los repositorios de imágenes indicados en el *Anexo II*.
- `ProviderDatasetTrainTest`: el origen de las imágenes es el mismo que en la clase anterior. La diferencia radica en que permite dividir el *dataset* en dos conjuntos: uno

para entrenamiento y otro para test. Cuando se crea el objeto hay que especificar cuál de los dos conjuntos va a generar, así como un valor específico o semilla.

- `ProviderDatasetMetadata`: permite obtener las imágenes de un *dataset* pero leyendo también los metadatos que incluya el mismo. Los metadatos dependen del *dataset* seleccionado, pero pueden el identificador el sujeto que hay en cada imagen (Caltech) o las coordenadas de los puntos de interés en la cara (Helen y BioID).
- `ProviderFolder`: recorre una carpeta devolviendo en cada iteración cada una de las imágenes contenidas en la misma.

Inicialización

Los parámetros de inicialización de cada una de las clases son:

- `ProviderWebcam ()`
- `ProviderVideo (filename)`
  - *filename*: nombre del fichero que contiene el vídeo de origen de los datos.
- `ProviderDataset (dataset, num_images)`
  - *dataset*: nombre de la base de datos de la que se van a extraer las imágenes. Los valores disponibles para este parámetro son:
    - *rasprec.BIOID*: base de datos de caras de BioID.
    - *rasprec.YALE*: base de datos de imágenes de caras de la Yale Face Database.
    - *rasprec.HELEN*: Helen Dataset
    - *rasprec.MIT*: banco de imágenes de caras del MIT
    - *rasprec.CALTECH_BACK*: imágenes de fondos de la Universidad de Caltech.
    - *rasprec.STANFORD_BG*: imágenes de fondos de la Universidad de Stanford.
    - *rasprec.TRAINING_FACES*: imágenes de caras incluidas en la librería Sklearn.
    - *rasprec.TRAINING_NFACES*: imágenes de no caras para el entrenamiento. Estas imágenes no provienen directamente de una base de datos pública, sino que se han generado extrayendo ventanas de diversas imágenes sin caras.
  - *num_images*: número de imágenes de la base de datos que se van a devolver.
- `ProviderDatasetTrainTest (dataset, seed, train_percent, num_images, train)`
  - *dataset*: igual que en la clase anterior.
  - *seed*: este valor numérico es la semilla para el generador de números pseudoaleatorios. Diferentes invocaciones con el mismo valor en este campo garantizan que se obtendrán las mismas imágenes para entrenamiento y para test de la base de datos.
  - *num_images*: número de imágenes de la base de datos que se van a utilizar.
  - *train_percent*: porcentaje del conjunto de imágenes seleccionadas del *dataset* que serán utilizadas para entrenamiento.
  - *train*: si el valor de este parámetro es True, se devolverán las imágenes seleccionadas para entrenamiento, si en cambio, es False, las imágenes devueltas serán para test.

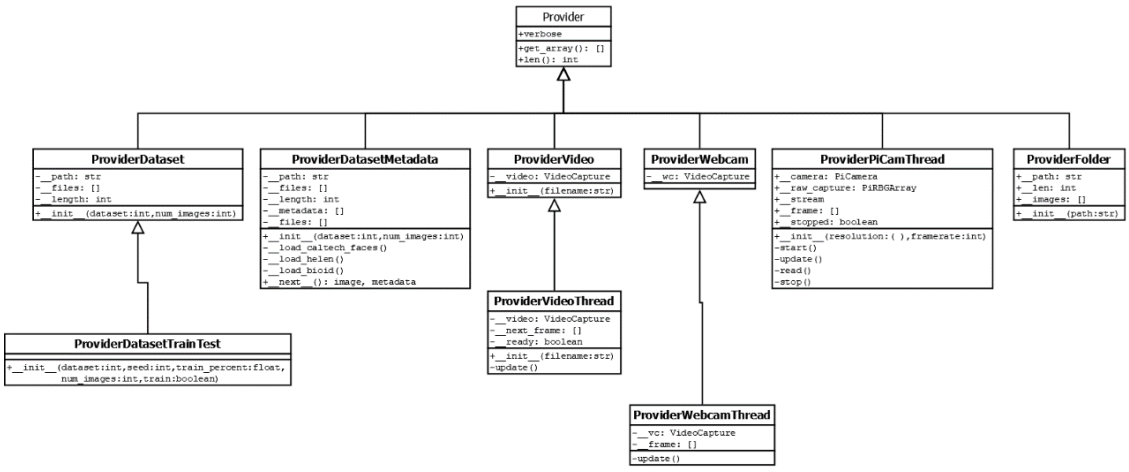
- `ProviderDatasetMetadata (dataset)`:
  - *dataset*: igual que en las clases anteriores. Únicamente están soportadas las bases de datos de Caltech, BioID y Helen, que son las que incluyen metadatos.
- `ProviderFolder (path)`
  - *path*: ruta de la carpeta que contiene las imágenes.

Funciones

Una vez inicializado el origen de los datos, todas las clases que heredan de la clase `Provider` tienen el mismo modo de funcionamiento. Todas ellas implementan un iterador que devuelve la siguiente imagen de dicho origen en cada iteración.

La única excepción es la clase `ProviderDatasetMetadata`, que devuelve dos elementos: la imagen y un diccionario con los metadatos que corresponden a dicha imagen.

Diagrama de la clase



3.2.2.- La clase Preprocessor

Todas las imágenes requieren de un procesamiento previo a su análisis, y ese es precisamente el cometido de esta clase, que realiza las operaciones necesarias sobre las imágenes de entrada con objeto de conseguir una detección óptima. A partir de la imagen original obtenida de la cámara o cualquier otro medio, genera una imagen en escala de grises, de las dimensiones solicitadas y con los ajustes más adecuados de intensidades.

Inicialización

Los parámetros del constructor de esta clase son:

- `Preprocessor (width, gray, norm, width)`
  - *width*: ancho al que se redimensionará la imagen de entrada, el alto será proporcional en función de las dimensiones de la imagen original.
  - *gray*: controla el modo en que se convertirá la imagen a escala de grises. Los modos disponibles son:

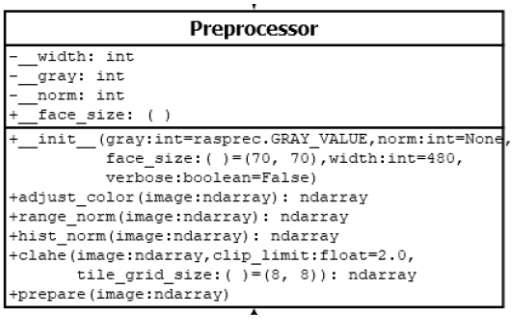
        - *rasprec.GRAY_EQUAL*: cada uno de los canales RGB tiene el mismo peso al calcular el valor de gris correspondiente.
        - *rasprec.GRAY_WEIGHTED*: cada canal RGB influye con un peso diferente al calcular el valor de gris, proporcionando mayor importancia a los colores a los que el ojo humano es más sensible.
        - *rasprec.GRAU_VALUE*: se convierte la imagen al modo de color HSV y se descartan los canales de matiz y saturación, que son los que contienen la información del color, manteniendo únicamente el canal de valor que contiene las intensidades de la imagen.
    - *norm*: si se estable algún valor a este parámetro, se aplica una normalización a la imagen. Los posibles valores que puede tener son:
        - *rasprec.NORM_HISTO*: se aplica la ecualización del histograma sobre la imagen de entrada.
        - *rasprec.NORM_RANGE*: se aplica la normalización de rango, de forma que los valores de intensidades de gris cubran toda la posible escala de valores.
        - *rasprec.CLAHE*: la normalización aplicada es la ecualización de histograma adaptativo con limitación de contraste (CLAHE)

Funciones

Esta clase tiene una única función que recibe una imagen de entrada y devuelve la imagen procesada en función de los valores establecidos en el constructor.

- `prepare(`*image*`)`
    - *image*: imagen de entrada.

Diagrama de clase



3.2.3.- La clase Detector

Esta clase contiene los métodos virtuales y métodos reutilizables por todas las clases que hereden de ella. Pretende establecer la estructura común que deberán tener todos los detectores de caras.

El objetivo final de esta clase es recoger una imagen de entrada y proporcionar un listado de las coordenadas de todos los rectángulos que contengan caras en ella.

Hay dos clases que heredan de esta:

- `DetectorViola`: implementa un detector facial basado en el algoritmo de Viola-Jones. Hay que tener en cuenta que este algoritmo está directamente implementado en la librería OpenCV, por lo que el funcionamiento de esta clase es relativamente sencillo.
- `DetectorCustom`: esta clase establece el marco para crear un detector facial personalizado. Para ello hay que facilitarle un generador de características y un clasificador, objetos heredados de las clases `Features` y `Classifier` respectivamente.

Inicialización

Los parámetros de inicialización de estas clases son:

- `DetectorViola` (*`classifier_file, scale_factor, min_neighbors, min_size, params`*)
  - *`classifier_file`*: el algoritmo de Viola-Jones requiere una etapa de entrenamiento bastante costosa computacionalmente. Para evitar este paso, la librería OpenCV facilita una serie de ficheros con diferentes clasificadores ya entrenados. En este parámetro se debe indicar la ruta del fichero de entrenamiento que se utilizará.
  - *`scale_factor`*: el algoritmo de Viola-Jones recorre la imagen original utilizando el método de las ventanas deslizantes, reduciendo el tamaño de las ventanas en cada iteración. Este parámetro establece el ratio de reducción de las ventanas en cada iteración.
  - *`min_neighbors`*: este parámetro sirve para indicar cuántos vecinos ha de tener un rectángulo para ser considerado una cara.
  - *`min_size`*: tamaño mínimo de las ventanas al aplicar el método de las ventanas deslizantes.
  - *`params`*: como alternativa a los parámetros anteriores, es posible indicar todos los parámetros mediante un diccionario.
- `DetectorCustom` (*`classifier, train_file, concurrent, fusion_mode, debug`*)
  - *`classifier`*: objeto de la clase `Classifier` que implementa el clasificador utilizado. Como se verá posteriormente, este objeto deberá incluir también el generador de características.
  - *`train_file`*: para evitar entrenar el clasificador, se puede indicar la ruta de un fichero de entrenamiento del mismo.
  - *`concurrent`*: parámetro booleano que indica si se hace uso o no del multiprocesamiento.
  - *`fusion_mode`*: lo habitual es que el clasificador proporcione diversos rectángulos solapados sobre una misma cara, por lo que es necesario algún método para eliminar esta redundancia. Los métodos disponibles son:
    - *`rasprec.FUSION_HEATMAP`*: utiliza mapas de calor.
    - *`rasprec.FUSION_NMS`*: utiliza el algoritmo denominado Non Maximum Suppression.
  - *`debug`*: este parámetro se introdujo con objeto de facilitar la depuración de los algoritmos utilizados. Su función es mostrar por pantalla diversas imágenes intermedias en el proceso de detección de las caras. Su valor es una lista con uno o varios de los siguientes valores:

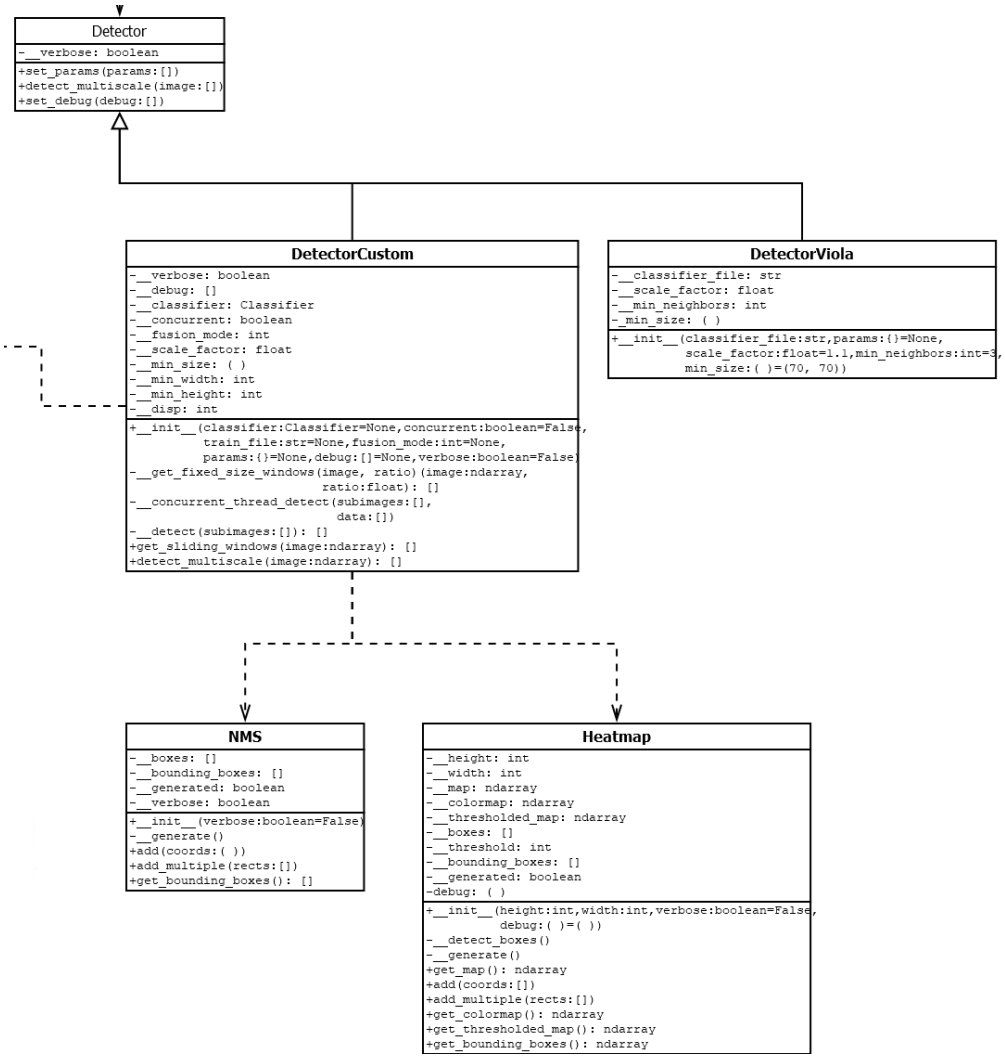
- *rasprec*.DEBUG_BOUNDING_BOXES: muestra todas las detecciones positivas de la imagen, antes de aplicar el método de fusión.
- *rasprec*.DEBUG_HEATMAP: muestra visualmente el mapa de calor.

Funciones

Las funciones más relevantes de estas clases son:

- `get_sliding_windows (image)`
  - *image*: imagen original a la que aplica el método de las ventanas deslizantes.
- `detect_multiscale (image)`
  - *image*: imagen de entrada.

Diagrama de la clase



3.2.4.- La clase Features

Esta clase contiene los métodos virtuales y reutilizables por todas las clases que hereden de ella para establecer una estructura común para todos los generadores de características. Los generadores de características que se han implementado son:

- `FeaturesHOG`: un tipo de descriptor bastante utilizado en el área de la visión por computador es el Histograma de Gradientes. Esta clase proporciona los métodos necesarios para obtener las características HOG de una imagen.
- `FeaturesBoWSIFT`: implementa un generador de características basado en el modelo de bolsas de palabras visuales (*Visual Bag of Words*) con SIFT (*Scale Invariant Feature Transform*).
- `FeaturesBoWSURF`: similar a la clase anterior, pero en este caso combina la bolsa de palabras visuales con SURF (*Speedup Robust Features*).

Inicialización

Los constructores de estas clases son:

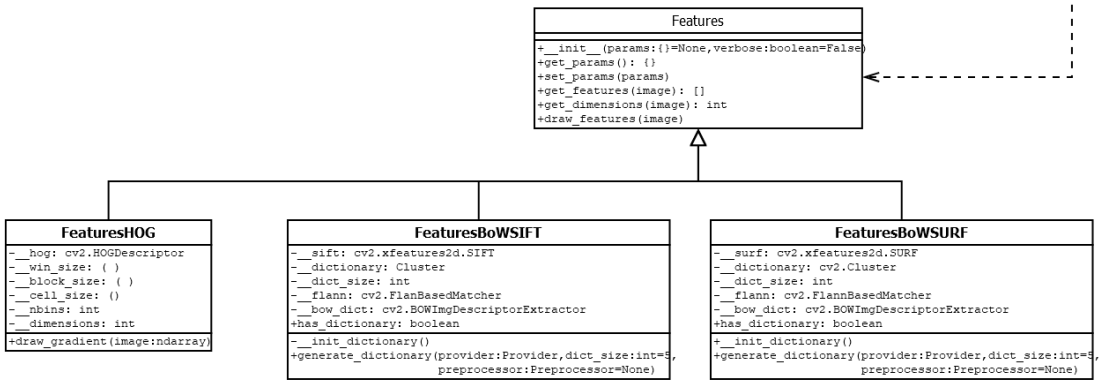
- `FeaturesHOG (win_size, cell_size, block_size, block_stride, nbins, params)`
  - *win_size*: tamaño de la ventana en la que se van a generar las características.
  - *cell_size*: tamaño de las celdas en que se descompone la imagen.
  - *block_size*: tamaño de los bloques en que se agrupan las celdas.
  - *block_stride*: nivel de superposición entre bloques.
  - *nbins*: número de rangos en el histograma.
  - *params*: como alternativa se pueden pasar todos los parámetros anteriores en un único diccionario mediante este parámetro.
- `FeaturesBoWSIFT (dict_size)`
  - *dict_size*: número de palabras en el diccionario.

Funciones

Las principales funciones de estas clases son:

- `get_features(image)`: devuelve el vector de características de la imagen pasada como parámetro.
  - *image*: imagen de entrada.
- *draw_features(image)*: en los métodos en los que sea posible, muestra por pantalla una representación visual de las características de la imagen.

Diagrama de la clase



3.2.5.- La clase Classifier

Esta clase establece el marco general para todos los clasificadores, proporcionando así una interfaz común a todas las clases que heredan de ella. Los diferentes clasificadores que se han implementado son:

- `ClassifierkNN`: implementa el método de los k vecinos más próximos (*k-Nearest Neighbours*)
- `ClassifierSVM`: implementa el clasificador de máquinas de vectores de soporte o SVM.

Inicialización

Los constructores de cada una de estas clases esperan los siguientes parámetros:

- `ClassifierkNN (features, neighs)`
  - *features*: objeto heredado de la clase `Features` que implementa el generador de características que se utilizará con el clasificador.
  - *neighs*: número de vecinos que se calcularán para el algoritmo kNN.
- `ClassifierSVM (features, params)`
  - *features*: objeto heredado de la clase `Features` que implementa el generador de características que se utilizará con el clasificador.
  - *params*: diccionario que puede contener los parámetros del algoritmo SVM.

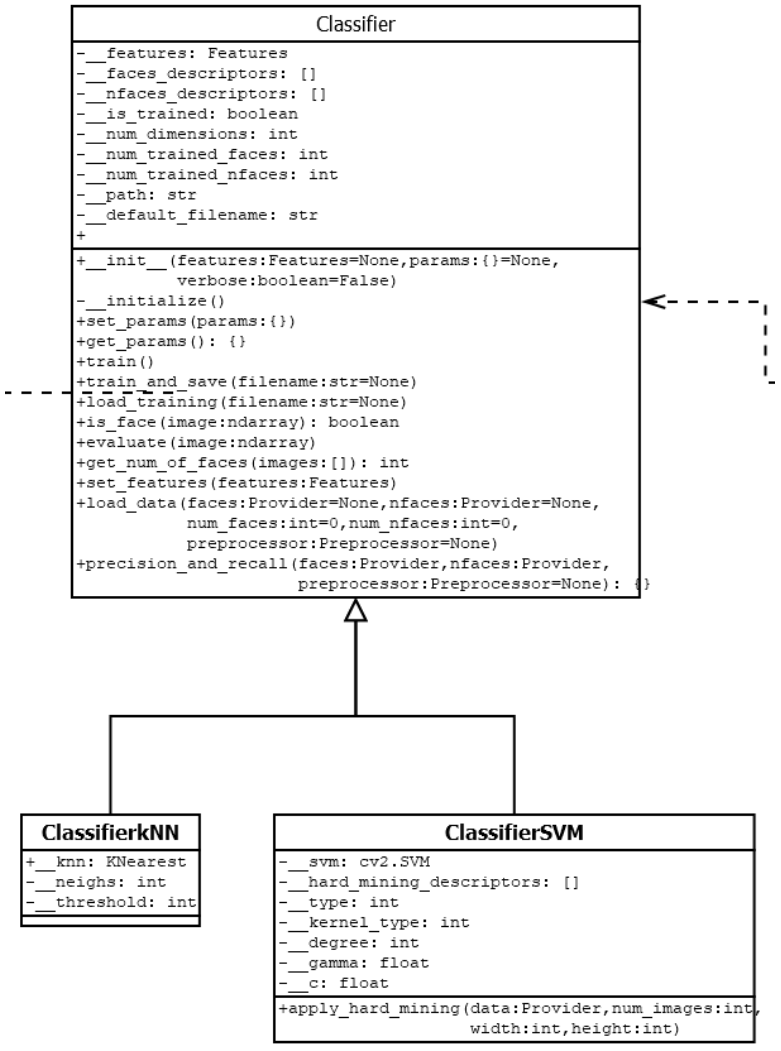
Funciones

Las funciones más relevantes de estas clases son:

- `load_data(faces, nfaces, num_faces, num_nfaces, preprocessor)`: todos los clasificadores requieren de un entrenamiento previo cuya entrada es un conjunto de imágenes correctamente etiquetadas. Esta función permite cargar las imágenes que se utilizarán para el entrenamiento. Los parámetros son:
  - *faces*: imágenes para entrenamiento que contienen caras. Se puede pasar un objeto de una clase heredada de la clase `Provider` o simplemente una lista donde los elementos son rutas a las imágenes de las caras de entrenamiento.
  - *nfaces*: análogo al parámetro anterior, pero con imágenes que no contienen caras.
  - *num_faces*: por defecto se utilizarán todas las caras facilitadas por el proveedor, pero se puede limitar el número de caras a utilizar en el entrenamiento mediante este parámetro.
  - *num_nfaces*: para limitar el número de imágenes de no caras para el entrenamiento.
  - *preprocessor*: si se quiere realizar algún tipo de preprocesamiento previo al entrenamiento del clasificador se puede pasar a esta función un objeto de la clase `Preprocessor`.
- `train()`: realiza en entrenamiento del clasificador utilizando las imágenes que se han cargado previamente.

- `train_and_save(filename)`: el tiempo requerido para el entrenamiento de algunos clasificadores puede ser largo, por lo que es posible guardar el clasificador ya entrenado en un fichero para evitar volver a tener que realizar el proceso de entrenamiento.
    - *filename*: nombre del fichero.
- `load_training(filename)`: recupera el entrenamiento del clasificador a partir de un fichero.
- `is_face(image)`: evalúa si una imagen que se le ha pasado corresponde a una cara o no.
- `are_faces(images)`: análoga a la función anterior, pero tiene como entrada una lista de imágenes y devuelve otra lista que indica cuáles corresponden a caras y cuáles no.
- `apply_hard_mining(data, num_images)`: solo disponible en la clase `ClassifierSVM`, aplica el método del *hard mining* para mejorar los resultados del clasificador.
    - *data*: objeto de la clase Provider con imágenes de no caras.
    - *num_images*: número de imágenes que se utilizarán.

Diagrama de la clase



3.2.6.- La clase Processor

El reconocimiento de las caras requiere un procesamiento de las caras detectadas con objeto de homogeneizar en lo posible todas las caras. Esta es la clase que se encarga de este procesamiento.

Inicialización

El constructor de esta clase tiene la siguiente estructura:

- `Processor (eye_detect_method, debug)`
  - *eye_detect_method*: un paso muy importante para el correcto desempeño de esta clase es la correcta localización de las posiciones de los ojos en la cara. Este parámetro especifica el método que se utilizará para detectarlos, los posibles valores son:
    - *rasprec.EYES_VIOLA*: utiliza el método de Viola-Jones.
    - *rasprec.EYES_FACIAL_LANDMARKS_68*: utiliza el método de detección de *facial landmarks* de 68 puntos.
    - *rasprec.EYES_FACIAL_LANDMARKS_5*: utiliza el método de detección de *facial landmarks* de 5 puntos.
  - *debug*: a efectos de depuración, la clase permite mostrar una salida por pantalla de las diferentes etapas intermedias del procesamiento. Este parámetro debe ser una lista que puede contener uno o varios de los siguientes valores:
    - *rasprec.DEBUG_EYES_DETECTION*: muestra la posición en la que se han detectado los ojos.
    - *rasprec.DEBUG_GEOMETRIC*: muestra la imagen de entrada tras aplicar las transformaciones geométricas (rotación, escalado y recorte).
    - *rasprec.DEBUG_HISTOGRAM*: muestra la imagen tras aplicar el histograma por partes.
    - *rasprec.DEBUG_SMOOTHING*: muestra la imagen tras aplicar el suavizado.
    - *rasprec.DEBUG_ELLIPTICAL*: muestra la imagen tras aplicar la máscara elíptica.
    - *rasprec.DEBUG_FULL_PROCESS*: muestra combinadas todas las imágenes anteriores.

Funciones

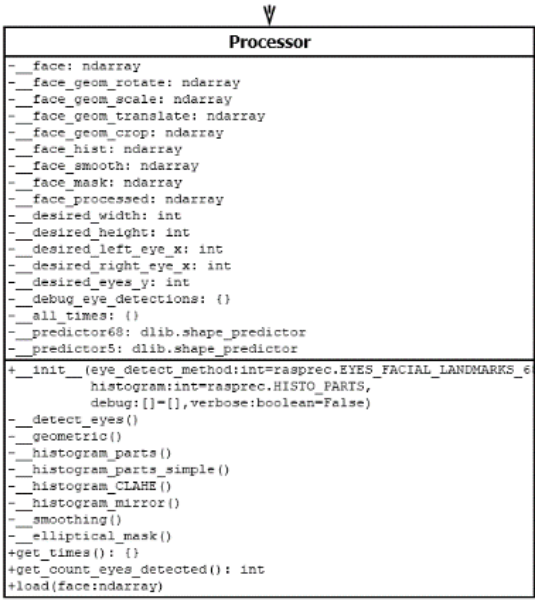
Las funciones más importantes de esta clase son:

- `load (face)`: carga una imagen de una cara y realiza todos los pasos del procesamiento sobre ella. Devuelve la imagen tras aplicar todos los pasos del procesamiento. El único parámetro que requiere es:
  - *face*: imagen de la cara que se debe procesar.
- `get_count_eyes_detected()`: devuelve el número de ojos que se detectaron en la última imagen cargada. Esta función solo es aplicable cuando el algoritmo de

detección de ojos utilizado es el de Viola Jones, ya que el algoritmo de *facial landmarks* siempre devuelve la posición de ambos ojos, aunque sea incorrecta.

- `get_times()`: devuelve un diccionario con los tiempos de procesamiento empleados en cada una de las fases.

Diagrama de la clase



3.2.7.- La clase Recognizer

La clase Recognizer es la encarga de implementar el reconocedor facial. Tras realizar el entrenamiento, requiere como entrada una imagen a la que se le ha aplicado el procesamiento y devuelve el identificador del sujeto al que pertenece.

Los reconocedores implementados son:

- `RecognizerEigenfaces`: implementa el algoritmo de Eigenfaces.
- `RecognizerFisherfaces`: implementa el algoritmo de Fisherfaces.
- `RecognizerLBP`: en este caso el algoritmo utilizado es Local Binary Patterns (LBP).

Inicialización

Los constructores de cada una de estas clases esperan los siguientes parámetros:

- `RecognizerFisherfaces (num_components, threshold)`
  - *`num_components`*: número de componentes que se calcularán en el algoritmo de Fisherfaces.
  - *`threshold`*: valor umbral para determinar si dos imágenes corresponden a un mismo sujeto. Un valor muy bajo solo identificará caras muy similares a las del entrenamiento, etiquetando muchas caras como *desconocidas*. Un valor alto siempre asignará una etiqueta a las caras

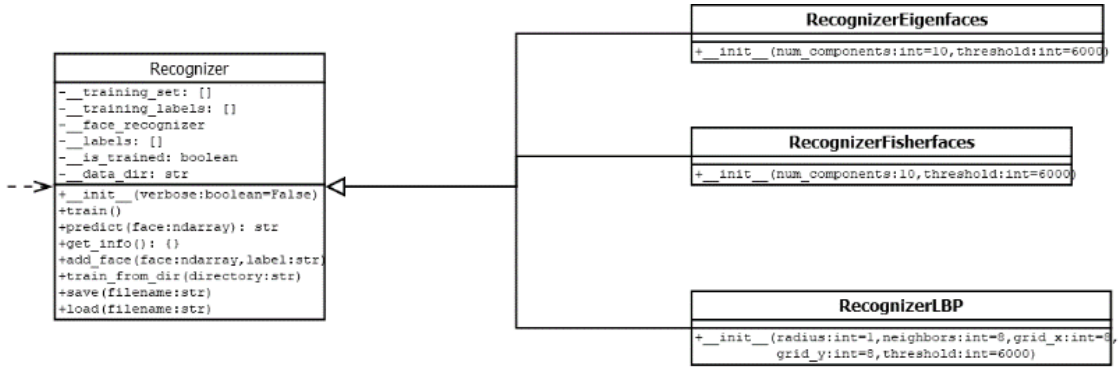
- `RecognizerEigenfaces (num_components, threshold)`: equivalente a la clase anterior.
- `RecognizerLBP(radius, neighbors, grid_x, grid_y, threshold)`:
  - `radius`: el radio hace referencia a la distancia de los puntos que se toman en consideración para calcular el valor de cada píxel.
  - `neighbors`: número de vecinos que se utilizan para calcular LBP. El valor más habitual es de 8 vecinos.
  - `grid_x/grid_y`: determinan el número de celdas en que se va a descomponer la imagen para calcular los histogramas.
  - `threshold`: igual que las clases anteriores.

Funciones

Las funciones más destacables de estas clases son:

- `add_face(face, label)`: carga una cara etiquetada para alimentar el algoritmo como paso previo al entrenamiento. Los parámetros de esta función son:
  - `face`: imagen de la cara, debe estar ya procesada.
  - `label`: cadena de texto con el identificador de la cara que se ha cargado.
- `train()`: tras haber cargado las caras se puede invocar a esta función para realizar el entrenamiento del reconocedor.
- `save(filename)`: una vez entrenado el reconocedor se puede almacenar el entrenamiento en un fichero mediante esta clase. El único parámetro es `filename` que indica el nombre del fichero donde se almacenará el entrenamiento.
- `load(filename)`: análogo a la función anterior, permite cargar el entrenamiento desde el fichero que se indique como parámetro.
- `train_from_dir(directory)`: esta función ofrece una alternativa a la carga manual de cada una de las imágenes para entrenamiento. Requiere la ruta de un directorio como parámetro y buscará en él las imágenes de entrenamiento. Este directorio debe contener un subdirectorio por cada uno de los sujetos conocidos, tomando el nombre del subdirectorio como la etiqueta de la persona a la que pertenecen las imágenes contenidas dentro de dicho subdirectorio.
- `predict(face)`: se le pasa una cara como parámetro y devuelve la cadena con el identificador de la persona a la que pertenece, o bien la cadena *desconocido* en caso de no ser una cara próxima a ninguno de los sujetos con los que ha sido entrenada.

Diagrama de la clase



3.2.8.- La clase Pipeline

La clase principal que articula todas las demás es la clase `Pipeline`, que toma una serie de imágenes de entrada y las guía a través de una serie de etapas para finalmente proporcionar las coordenadas de las caras encontradas y el identificador de estas. Las diferentes fases que incluye este proceso son:

- Obtención de los datos, mediante la clase `Provider`.
- Preprocesamiento de las imágenes para prepararlas para el detector de caras, funcionalidad ofrecida por la clase `Preprocessor`.
- Detección de las imágenes en cada una de las imágenes, con la clase `Detector`.
- Procesamiento de las caras detectadas para prepararlas para el reconocedor, mediante la clase `Processor`.
- Y finalmente, reconocimiento de las caras, implementado en la clase `Recognizer`.

4.- Implementación

4.1.- Introducción

El primer paso para afrontar la implementación de la aplicación ha sido determinar la estructura de directorios a utilizar. En la imagen de la derecha se puede ver el árbol de directorios del proyecto, cuya utilidad se expone a continuación:

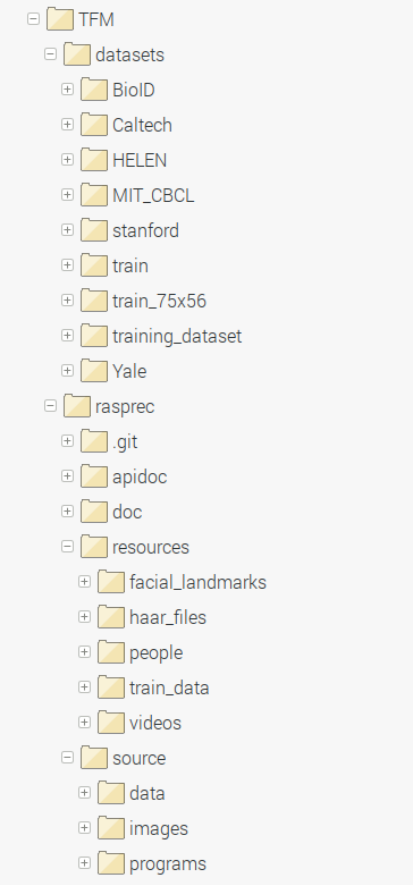


Figura 22: Esquema de directorios del proyecto

- **datasets**: este directorio contiene un subdirectorio por cada una de las bases de datos de imágenes utilizadas en el proyecto. El hecho de ubicar estas bases de datos fuera del directorio del proyecto se debe al gran tamaño de las mismas, por lo que se ha considerado conveniente situarlas fuera del directorio sincronizado con Git.
- **rasprec**: contiene los ficheros del proyecto propiamente dicho.
- **doc**: documentación del proyecto, incluyendo la presente memoria.
- **resources**: este directorio contiene todos los recursos utilizados. En concreto, contiene los siguientes subdirectorios:
    - **facial_landmarks**: ficheros de entrenamiento utilizados en el algoritmo *facial landmarks*.
    - **haar_files**: el algoritmo de Viola Jones incluido en OpenCV dispone de varios ficheros de entrenamiento que se han ubicado en este fichero.
    - **people**: una de las posibilidades de la clase `Recognizer` es cargar las imágenes de entrenamiento a partir de un directorio. Utilizará por defecto este directorio si no se indica ninguno al realizar el entrenamiento.
    - **train_data**: directorio por defecto utilizado para guardar los ficheros de entrenamiento de los diferentes algoritmos.
    - **videos**: muchas de las pruebas se han realizado sobre vídeos grabados que se almacenan en este directorio.
- **source**: directorio que almacena el código fuente de la aplicación.

4.2.- Preprocesamiento

Cuando las imágenes son obtenidas desde la cámara, las condiciones lumínicas y ambientales pueden variar enormemente, afectando de forma negativa al proceso de detección. Las variaciones entre imágenes de caras diferentes son generalmente menores que las variaciones en una misma cara obtenida en diferentes entornos. Por lo tanto, el objetivo de este preprocesamiento es normalizar en cierto modo los efectos de brillo y contraste.

El determinar qué tipo de operaciones pueden suponer una mejora en el posterior proceso de detección puede ser una decisión difícil y que muy probablemente haya que determinar de forma empírica realizando diversas pruebas. Esto se debe a que la mejora estará condicionada a muchos factores: tanto propios de la imagen de entrada como de los algoritmos utilizados.

Algunas de las operaciones que se deben realizar son de redimensionado y recorte, de cambios en los espacios de color utilizados y de manipulación de los niveles de intensidad de la imagen.

En la figura 23 se resumen los pasos realizados en esta etapa y las posibilidades que se han explorado.

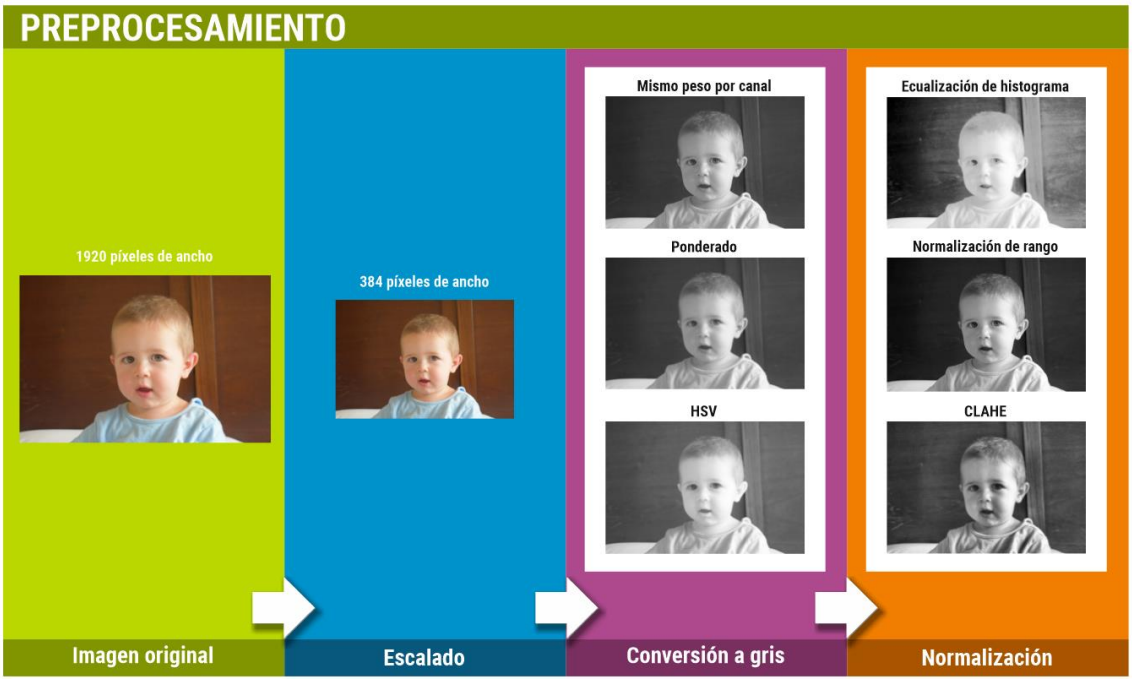


Figura 23: Etapas del procesamiento

4.2.1.- Recorte y redimensionado

El primer paso que se debe realizar sobre las imágenes obtenidas de la cámara es el **recorte y redimensionado** de las mismas. La cámara de la Raspberry Pi puede obtener imágenes en vídeo a 30 fotogramas por segundo de hasta 1920x1080 píxeles. Este elevado valor puede parecer algo positivo ya que, cuanta más resolución, mejor calidad de imagen, y, por tanto, podría parecer que mejor reconocimiento de las caras. Sin embargo, su efecto es todo lo contrario. Utilizar resoluciones tan altas incrementa exponencialmente los tiempos necesarios de computación y harían totalmente inviable el proceso.

Por otro lado, es necesario que las imágenes de las que se extraerán las características deben tener un tamaño fijo, ya que el tamaño de la imagen determina el número de dimensiones del

vector de características, y los clasificadores requieren que todas las muestras tengan igual número de dimensiones. Si la fuente de las imágenes es una cámara de vídeo esto no supone ningún problema ya que todos los fotogramas tienen el mismo tamaño.

En cambio, si la fuente son ficheros de imágenes es muy probable que tengan diferentes tamaños y diferentes ratios por lo que será necesario redimensionar y recortar las imágenes para que el tamaño final sea el mismo en todas.

4.2.2.- Cambios en el espacio de color

Las imágenes originales se encuentran en RGB, donde cada píxel contiene los valores correspondientes a los colores básicos rojo, verde y azul. Sin embargo, todos los sistemas de detección de objetos se basan en las variaciones de los niveles de intensidad de las imágenes por lo que por norma general es irrelevante la información relativa a los colores.

Hay múltiples formas de convertir una imagen en RGB en escala de grises. La más obvia es que el valor de cada píxel en gris sea igual a la media de los valores de cada uno de los tres componentes RGB.

$$G_{xy} = R_{xy} * 0{,}333 + G_{xy} * 0{,}333 + B_{xy} * 0{,}333$$

Este método se puede considerar un poco burdo desde la perspectiva de que no contempla las características propias del ojo humano, que no percibe de la misma forma cada uno de los tres colores básicos. Es mucho más sensible a las variaciones de color de los tonos verdes, mientras que tiene una sensibilidad más baja al percibir las diferentes tonalidades de azul. La función de OpenCV para convertir de RGB a escala de grises tiene en cuenta estas particularidades y asigna diferentes pesos a cada canal RGB en función del color de que se trate. La forma que utiliza en este caso es la siguiente:

$$G_{xy} = R_{xy} * 0{,}299 + G_{xy} * 0{,}587 + B_{xy} * 0{,}114$$

Por último, la otra alternativa contemplada es la posibilidad de cambiar la imagen a otro espacio de color, en concreto a **HSV** (Matiz-Saturación-Valor). En este espacio de color, los canales de matiz y saturación almacenan la información de color, reservando el tercer canal para almacenar la información relativa a la intensidad de la imagen. Por tanto, es razonable descartar los dos primeros para trabajar únicamente con el canal de valor.

En la figura 24 se pueden apreciar los resultados obtenidos utilizando los tres métodos propuestos.

Imagen original | Mismo peso en todos los canales | Pesos ponderados en canales RGB | Canal *valor* en modo HSV

Figura 24: Modos de conversión a escala de grises

4.2.3.- Normalización de la imagen

Un problema importante para la detección facial es la gran variabilidad en cuanto a las condiciones lumínicas y ambientales que pueden dificultar el proceso de detección. Entornos poco luminosos o que tienen fuertes cambios en la dirección e intensidad de la luz introducirán unas variaciones excesivas en las imágenes lo que afectará negativamente a los algoritmos de detección.

Por ejemplo, en la figura 25 se puede apreciar una cierta pobreza en los niveles de intensidad de la imagen, con muy poco contraste en general. Este hecho se puede constatar en el histograma de la derecha, donde se puede apreciar que la mayoría de los píxeles se encuentran en el rango central de intensidades (con valores entre 100 y 175 aproximadamente), mientras que no hay píxeles que tengan valores en los extremos, es decir, no hay píxeles de color blanco ni negro, sino que son todos de color gris.

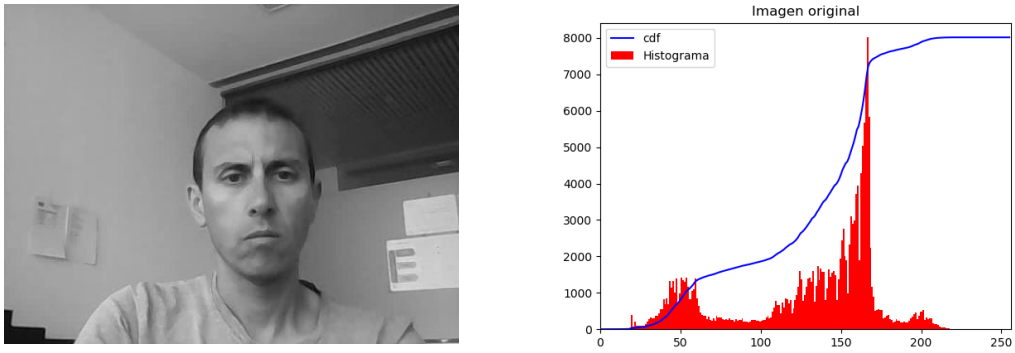


Figura 25: Histograma

La idea detrás de la normalización de la imagen es obtener un histograma donde los valores de intensidad están más repartidos, redundando por tanto en imágenes con mayor contraste. Se han implementado tres métodos diferentes de normalización.

Normalización de rango

Este primer método consiste en *estirar* el histograma, reajustándolo para que las intensidades máxima y mínima de la imagen se sitúen en los valores máximo y mínimo del rango de valores posibles de intensidad de la imagen. Para ello se utilizará la fórmula:

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

donde $x = (x_1, x_2, \dots, x_n)$ es la imagen original de n píxeles y $z = (z_1, z_2, .., z_n)$ la nueva imagen normalizada.

En la figura 26 se muestra el resultado de aplicar dicha normalización a la imagen anterior.

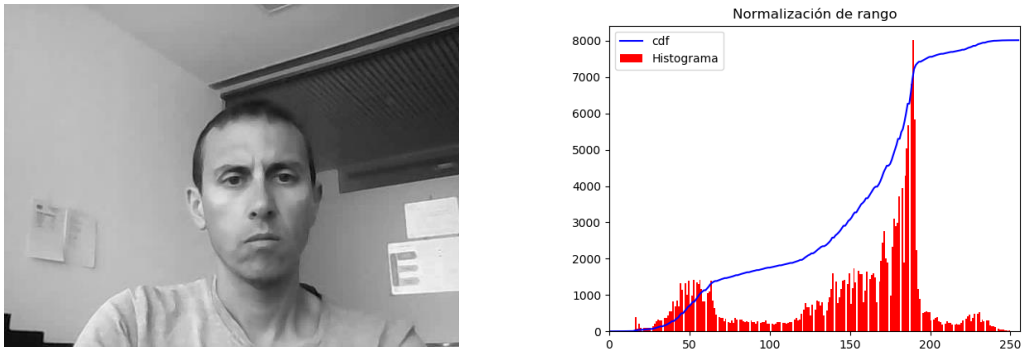


Figura 26: Histograma tras aplicar normalización de rango

Ecualización de histograma

La ecualización del histograma es una técnica bastante utilizada para mejorar el contraste general de una imagen. Se basa en la distribución de las intensidades de la imagen uniformemente a lo largo de todo el rango de valores disponibles.

En la figura 27 se puede apreciar que los resultados obtenidos son mejores que en el caso de la normalización de rango, teniendo la imagen en general un mejor contraste.

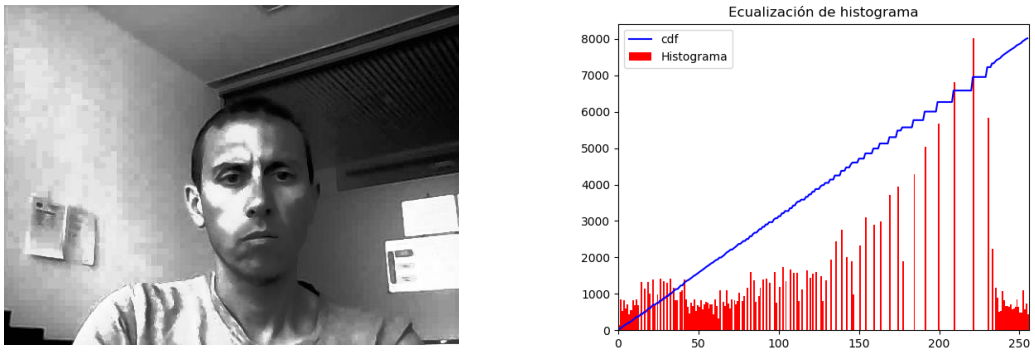


Figura 27: Histograma tras aplicar ecualización

CLAHE (Contrast Limited Adaptive Histogram Equalization)

La ecualización del histograma funciona bien cuando la distribución de los valores de los píxeles es similar a lo largo de toda la imagen. Sin embargo, cuando la imagen contiene regiones que son significativamente más claras u oscuras que el resto de la imagen, la mejora en el contraste en estas regiones no será suficiente.

La ecualización del histograma adaptativo (AHE) intenta solucionar este inconveniente transformando cada píxel en función del histograma del área que rodea a dicho píxel. Esto implica un mayor costo computacional, ya que hay que calcular un histograma para cada píxel, aunque hay métodos para optimizar este proceso.

El tamaño del área que se tendrá en cuenta influirá en el resultado final, un área pequeña generará zonas de mayor contraste, mientras que áreas mayores implicarán una reducción del contraste.

Un inconveniente de AHE es que puede sobre amplificar el ruido en la imagen. Para evitar esto hay una variante de este denominada Contrast Limited AHE (CLAHE), en el cual se limita la amplificación del contraste, y, por tanto, la posible amplificación del ruido.

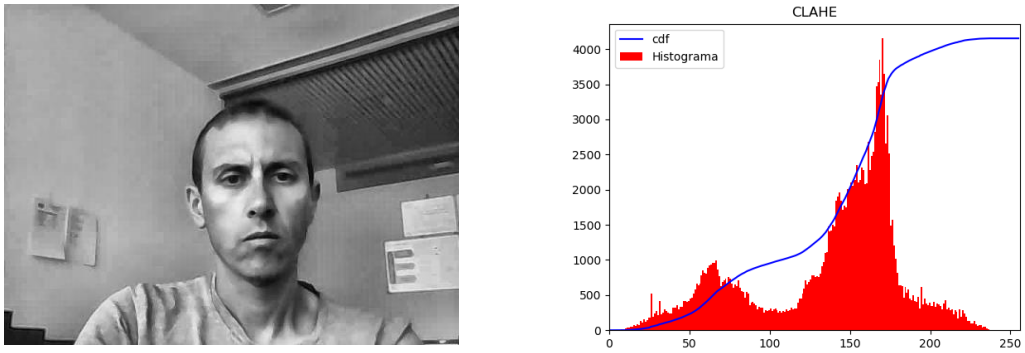


Figura 28: Histograma tras aplicar CLAHE

Para aplicar CLAHE hay dos parámetros que permiten ajustar su funcionamiento:

- **Clip limit**: por norma general, CLAHE tiende a sobre amplificar el contraste en zonas de contraste constante de la imagen, pudiendo generar ruido en estas zonas. Este parámetro permite limitar este efecto tal y como se puede ver en la figura 29.

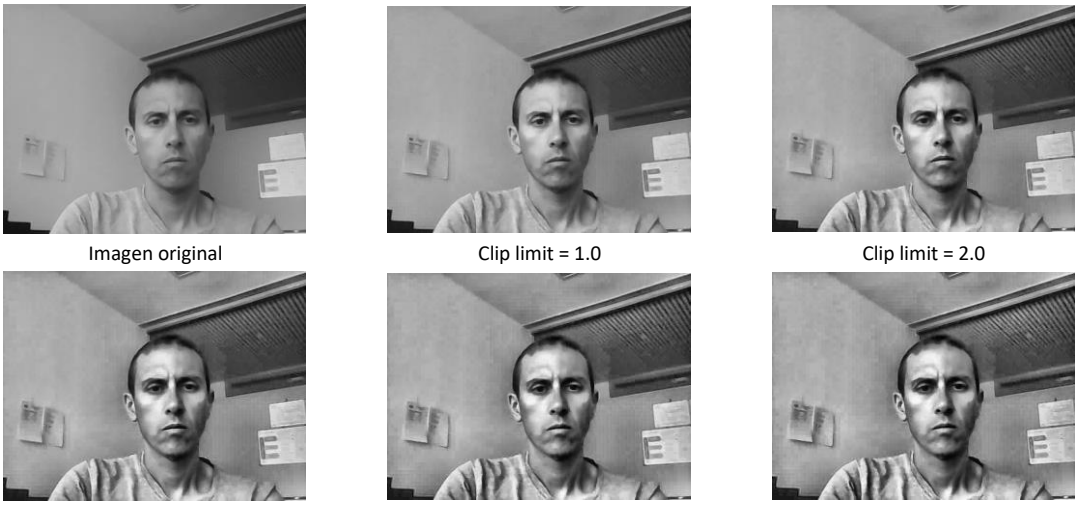


Figura 29: Efectos de CLAHE con diferentes valores de clip limit

- **Tile grid size**: este otro parámetro determina el número de casillas en que se dividirá la imagen original para calcular el histograma. Un valor alto implica más casillas y por tanto más pequeñas, generando así un área de influencia menor para calcular el valor de cada punto, lo que provoca un mayor contraste. En la figura 30 se pueden ver los efectos de diferentes tamaños sobre una misma imagen.

Figura 30: Efectos de CLAHE con diferentes tamaños de tile grid

Los tiempos empleados para realizar los pasos anteriores en la Raspberry Pi sobre una imagen en color con 480 píxeles de ancho se exponen en la figura 31. Lo más razonable, teniendo en cuenta los resultados esperados, es utilizar la conversión a escala de grises mediante pesos ponderados y usar CLAHE para realzar el contraste de la imagen.

```
(cv3) pi@raspberrypi:~/TFM/rasprec/source/programs $ python 10_measure_preprocessor_times.py
---------- TIEMPOS DE CONVERSIÓN A GRIS ----------
Mismo peso: 13.744420632000015 ms.
Ponderado : 1.0031567859999768 ms.
HSV       : 4.721661486999892 ms.

---------- TIEMPOS DE NORMALIZACIÓN ----------
Histograma: 1.339490359000024 ms.
Rango     : 13.685950322999929 ms.
CLAHE     : 2.5452011940000148 ms.
```

Figura 31: Tiempos empleados para preprocesamiento

4.3.- Detección

La implementación del algoritmo de Viola Jones proporcionada en la librería OpenCV incluye toda la funcionalidad necesaria para localizar las caras dentro de una imagen. Esa funcionalidad, sin embargo, no está disponible para otros algoritmos, tales como HOG o BoW SIFT. En esos casos, aunque OpenCV sí que proporciona la implementación de los algoritmos propiamente dichos, es necesario crear la estructura necesaria para aplicarlos sobre una imagen y determinar en qué posiciones se encuentran las caras dentro de la misma.

Por ello, en el proceso de detección se pueden identificar los siguientes pasos:

- Descomponer la imagen en subimágenes o ventanas que incluyan todas las posiciones y escalas, esto se consigue mediante el método de las **ventanas deslizantes**.
- **Obtener el descriptor** de cada una de esas ventanas según las características escogidas.
- Utilizar un **algoritmo de clasificación** para determinar si el descriptor de cada ventana corresponde a una cara o no.

- Es habitual que haya diferentes identificaciones positivas solapadas que corresponden a una misma cara, o bien que haya ventanas erróneamente clasificadas como cara. Para solucionar estos problemas es preciso aplicar algún **método de fusión**, como pueden ser los mapas de calor o NMS.

En la figura 32 se muestra un esquema de la estructura seguida en el proceso de detección.

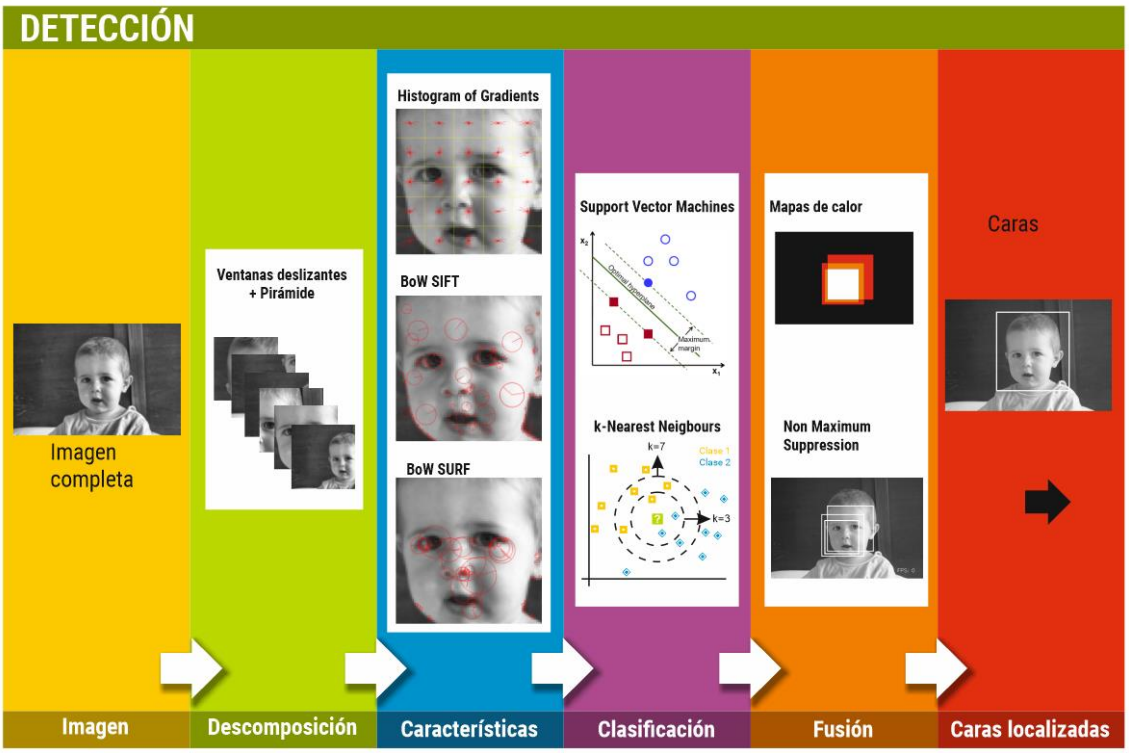


Figura 32: Etapas de la detección

4.3.1.- Ventanas deslizantes

El clasificador permitirá, a partir del descriptor de una imagen, determinar si dicha imagen corresponde a una cara o no. Sin embargo, esto solo es aplicable para el caso de que la cara ocupe prácticamente toda la imagen. Pero lo habitual es que las caras puedan estar en diferentes posiciones de la imagen o incluso que se encuentren a diferentes escalas. Esto implica que será necesario calcular el descriptor en diferentes partes de la imagen y también a diferentes escalas.

Figura 33: Ventanas deslizantes

Hay dos mecanismos para conseguir estos objetivos: el método de las **ventanas deslizantes** permitirá analizar todas las posiciones de la imagen y las **pirámides** ayudarán a detectar caras de diferentes tamaños.

Al calcular el descriptor de la imagen, ya se vio que había que indicar un **tamaño de ventana**, es decir,

el tamaño de imagen sobre la que se van a calcular el descriptor. Esa ventana es la que hay que iterar sobre toda la imagen original, permitiendo así detectar caras en cualquier posición de la imagen.

Con el método de las **ventanas deslizantes** se comienza obteniendo el descriptor correspondiente al fragmento de la imagen del tamaño de la ventana en la esquina superior izquierda y se va desplazando la ventana hacia la derecha calculando los descriptores en cada posición. A continuación, se desplaza la ventana verticalmente y se vuelve a iterar sobre todas las posiciones. Y así sucesivamente.

El factor clave en este método es el **desplazamiento** entre una ventana y la siguiente. Si el desplazamiento tiene un valor muy bajo el resultado final será mucho más preciso ya que se obtendrán descriptores para cualquier posición de la imagen, pero a costa de un mayor tiempo de cómputo ya que se tendrá que calcular un mayor número de descriptores. Un valor de desplazamiento más alto requerirá menos tiempo de cómputo, pero también implicará una menor precisión de detección.

Con el método de las ventanas deslizantes se puede encontrar una cara independientemente de la posición de la imagen original en que se encuentre, pero siempre y cuando el tamaño de la cara coincida con el tamaño de ventana que utilicemos. Si el tamaño de la cara fuera mayor la ventana únicamente detectaría un fragmento de ésta y no la detectaría como cara.

Este problema se soluciona con las **pirámides**. Este es un método iterativo en el cual se aplican las ventanas deslizantes para diferentes tamaños de la imagen. La primera iteración se aplica sobre el tamaño original de la imagen. Para la segunda iteración se escala la imagen original, reduciéndola ligeramente, y se vuelven a aplicar las ventanas deslizantes para este nuevo tamaño. Como el tamaño de ventana sigue siendo el mismo esto permitirá detectar caras que en la imagen de partida tenían un tamaño algo mayor que el tamaño de ventana.

Figura 34: Método de pirámides

En la siguiente iteración se reduce aún más la imagen, lo que permitirá detectar caras aún mayores. Y así sucesivamente hasta que se haya alcanzado un tamaño mínimo, que habitualmente suele coincidir con el tamaño de ventana.

Nuevamente hay un parámetro que es determinante en el desempeño de este método, que es la **escala** que se aplica en cada nueva iteración. Un valor muy bajo haría que las imágenes se reduzcan muy poco en cada iteración, y por tanto habría muchas iteraciones con el consiguiente consumo de CPU ya que en cada iteración hay que calcular todos los descriptores

de la imagen. Un valor alto reduciría el tiempo de procesador requeridos, pero será menos preciso ya que se probarán menos tamaños de cara.

4.3.2.- Generación del descriptor

Los algoritmos de clasificación trabajan en entornos n-dimensionales, es decir, asocian cada imagen con un vector en un espacio de n dimensiones y en función de su ubicación en dicho espacio determinan si corresponde a una cara o no. Por ello, un paso muy importante es eliminar toda la información no relevante de la imagen con objeto de reducir el número de dimensiones del espacio en que se va a trabajar.

Por ejemplo, en el caso concreto de este proyecto se ha escogido un tamaño de ventana de 70x70 píxeles en escala de grises. Si se asocia cada píxel con una dimensión donde el valor del píxel determina la posición dentro de esa dimensión, habría un espacio con 4900 dimensiones, un valor muy alto, sobre todo teniendo en cuenta que hay mucha información que no es relevante para el proceso de detección. Calcular el descriptor de una imagen basándose en unas características determinadas implica quedarse con la información más relevante para la imagen descartando lo que sea superfluo.

Las diferentes características enfatizan en un tipo de información y otro. Por ejemplo, las Cascadas de Haar o el Histograma de Gradientes se centran en la información relativa a los cambios de intensidad de la imagen, mientras que SIFT o SURF le dan más importancia a los puntos de interés de la imagen, como pueden ser las esquinas.

Como se ha dicho, el algoritmo de Viola Jones, que es el que utiliza las Cascadas de Haar, está completamente implementado en OpenCV. Omitiendo las cascadas de Haar del algoritmo de Viola Jones, por estar plenamente implementadas en OpenCV, se exponen a continuación los aspectos más relevantes de las características utilizadas.

Histograma de Gradientes (HOG)

En el apartado 2.3.2 ya se han explicado los fundamentos de este algoritmo. La implementación de OpenCV genera directamente el descriptor de la imagen a partir de la misma y según los parámetros escogidos, siendo el tamaño de la imagen y el valor de los parámetros lo que determina el número de dimensiones que tendrá el descriptor.

SIFT y SURF con Bag of Words (BoW)

Más interesante es el caso de los algoritmos SIFT y SURF, ya explicados en los apartados 2.3.3 y 2.3.4. Como se vio, estos algoritmos identifican dentro de la imagen una serie de puntos de interés, cada uno de ellos con su descriptor. El problema aquí surge del hecho de que todos los algoritmos de clasificación requieren que los descriptores de todas las imágenes tengan el mismo número de dimensiones, y eso no es posible con estos dos algoritmos, ya que el número de puntos de interés en la imagen es variable.

La solución a este problema es usar el modelo **Bolsa de Palabras Visuales** (*Bag of Visual Words*), un método ampliamente utilizado en clasificación y que permite representar las

características obtenidas con algoritmos como SIFT o SURF a un espacio con un número fijo de dimensiones.

La idea detrás de este método es representar una imagen como un conjunto de características que consisten en puntos de interés y sus descriptores. La idea general es identificar diferentes tipos de puntos de interés en función de su descriptor, que es lo que se conoce como el vocabulario. A continuación, se representa cada imagen como un histograma de frecuencias que refleja el número de apariciones de cada tipo de punto de interés dentro de la imagen. Esto permite identificar imágenes similares a una determinada en función de la similitud de sus histogramas de frecuencias.

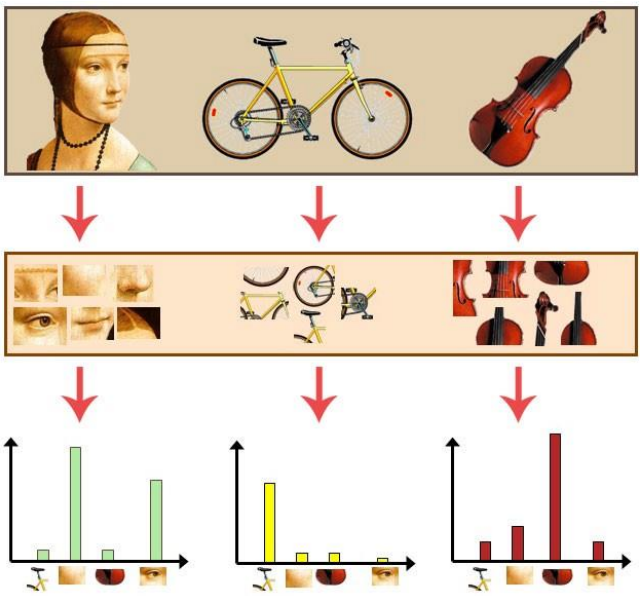


Figura 35: Esquema de BoVW

El descriptor de la imagen será el histograma de frecuencias, por lo que el número de dimensiones de dicho descriptor siempre será la misma, ya que estará determinado por el tamaño del vocabulario y por tanto será independiente del número de puntos de interés que se encuentren en la imagen. En la figura 35[3] se puede ver una simplificación de cómo funcionaría el método de BoW para generar los histogramas de las imágenes.

De la explicación anterior se puede deducir que la aplicación de este método requiere de dos pasos:

Creación del diccionario de palabras visuales

Este paso debe ser realizado una única vez y su objetivo es determinar cuál será el vocabulario que se utilizará. Por tanto, es un paso que se realiza una única vez y los datos obtenidos en él se aplican a todas las imágenes.

Los pasos que involucra son:

- Se escoge una serie de imágenes de entrenamiento y se calculan los puntos de interés de cada una de ellas según el algoritmo deseado (SIFT, SURF, ORB, …).
- Cada punto de interés tiene dos tipos de información asociadas: la localización del punto con información como posición, escala o ángulo y el descriptor asociado. La localización no es relevante para este caso, interesando únicamente el descriptor.
- Los descriptores son puntos en un espacio n-dimensional, por ejemplo, en el caso de SIFT los descriptores son vectores de 128 dimensiones. Se ubican en el espacio todos

[3] Fuente de la imagen: https://towardsdatascience.com/bag-of-visual-words-in-a-nutshell-9ceea97ce0fb

los descriptores calculados, que pueden ser varios miles en función del tamaño del *dataset* de entrenamiento utilizado.

- Finalmente se determina el tamaño deseado para el vocabulario y se clusterizan los descriptores en tantos clústeres como palabras se desea que tenga el vocabulario.

Cálculo del descriptor de una imagen

Una vez calculado el diccionario es utilizado para obtener el descriptor de cada imagen. Básicamente, los pasos a seguir son:

- Se calculan los puntos de interés de la imagen.
- Para cada uno de los puntos de interés se calcula a qué clúster pertenece su descriptor, es decir, se asocia a una palabra visual.
- Una vez identificada la palabra visual que corresponde a cada punto de interés se genera el histograma de frecuencias que será el descriptor de la imagen, con tantas dimensiones como palabras tenga el vocabulario.

4.3.3.- Clasificación

El proceso de clasificación es bastante sencillo ya que los algoritmos utilizados (SVM y kNN) están implementados en OpenCV y únicamente requieren de la carga previa de los descriptores de las imágenes de entrenamiento, y una vez entrenados, a partir del descriptor de una imagen la clasifican en un tipo u otro.

4.3.4.- Refinamiento de los resultados

Como se ha visto, cada una de las partes de la imagen obtenidas mediante las ventanas deslizantes es evaluada con el clasificador para detectar si contiene una cara o no. Además, este proceso se realiza con diferentes tamaños de ventana. Esto quiere decir que muchos de los fragmentos analizados se solapan y, por tanto, si hay una cara en la imagen original, ésta se encontrará en múltiples ventanas teniendo como consecuencia una alta probabilidad de que haya múltiples ventanas que den positivo para la misma cara de la imagen.

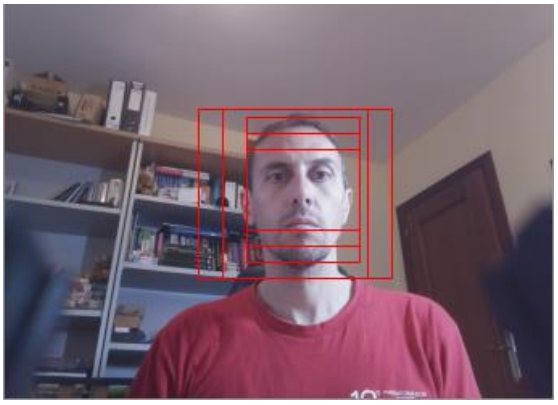


Figura 36: Múltiples detecciones positivas superpuestas

Igualmente, también es probable que haya falsos positivos, es decir, ventanas que el clasificador identifique como cara, aunque no lo sean.

Por tanto, será necesario realizar un proceso de refinamiento sobre los resultados obtenidos para, por un lado, eliminar los falsos positivos y por otro, unificar las diferentes detecciones obtenidas sobre una misma cara.

Básicamente el proceso consistirá en analizar todos los resultados positivos obtenidos, cada uno definido por la caja que contiene dicho fragmento de la imagen y evaluar el nivel de

solapamiento entre dichas cajas. Si hay múltiples cajas que se solapan indicará que con toda probabilidad se trata de una cara, por lo que se deberán unificar todas esas cajas en una única caja. Por el contrario, si hay una caja que no se solapa con ninguna otra es bastante probable que se trate de un falso positivo, por lo que debería descartarse.

Se han propuesto diferentes soluciones a este problema, de las que he optado por implementar las siguientes:

- Mapas de calor
- Non-Maximum Suppression (NMS)

Mapas de calor

El primer mecanismo implementado para unificar detecciones solapadas y eliminar falsos positivos son los mapas de calor. En esencia, un **mapa de calor** es una imagen en la que cada píxel tiene un valor proporcional al número de rectángulos positivos en los que esté incluido. Así, un píxel que esté dentro de una cara es muy probable que esté ubicado dentro de diversos rectángulos, por lo que tendrá un valor alto, mientras que un píxel que no pertenezca a una cara tendrá un valor nulo, o en su defecto, muy bajo si está incluido dentro de un falso positivo.

Los pasos que se realizan cuando se aplica este método son:

- Crear una matriz del mismo tamaño que la imagen original inicializando todos sus valores a cero. A continuación, se recorren todos los rectángulos de las caras detectadas por el clasificador y se incrementa en uno el valor de los píxeles incluidos dentro de cada uno de los rectángulos.
  En las siguientes imágenes se puede ver la imagen original, con los rectángulos detectados, junto al mapa de calor que refleja la probabilidad de que cada píxel pertenezca a una cara.

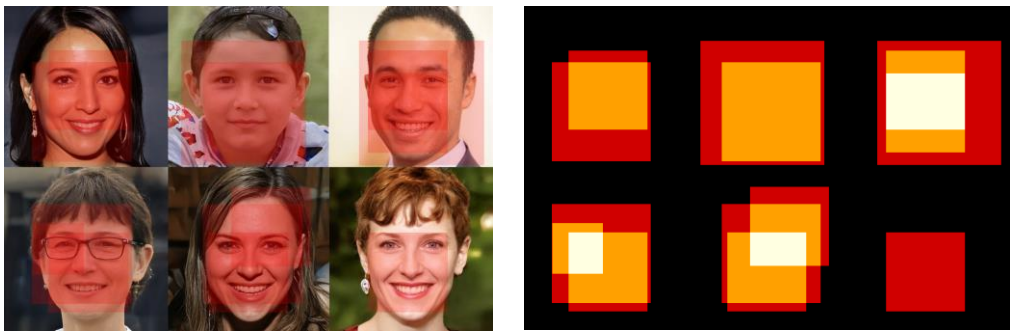


Figura 37: Creación de mapas de calor

- El siguiente paso es seleccionar un **valor umbral** y eliminar todos los píxeles cuyo valor sea inferior al de dicho umbral. Este valor es muy importante, y deberá ser determinado empíricamente, ya que un valor muy bajo implicaría que se colarán muchos falsos positivos, mientras que un valor muy alto puede que elimine algunas caras bien detectadas.
- Una vez eliminados los puntos que no superen el valor umbral hay que determinar si hay una única cara en la imagen o hay más. Para ello, hay que tener en cuenta que en este punto hay una matriz donde cada cara es un conjunto contiguo de puntos con

valores diferentes de cero y cada una de las caras está rodeada por puntos cuyo valor es cero. Teniendo esto en cuenta, se puede utilizar la función `label` de la librería Scipy que busca los conjuntos de puntos contiguos de una imagen y los etiqueta.
De esta forma se obtiene una matriz donde todos los puntos que corresponden a una misma cara tienen el mismo valor numérico.

- El último paso que queda es iterar sobre cada uno de los números etiquetados en el punto anterior y localizar el mayor y menor valor de estos sobre cada uno de los ejes, y que corresponderán a las coordenadas de la caja que contiene la cara.

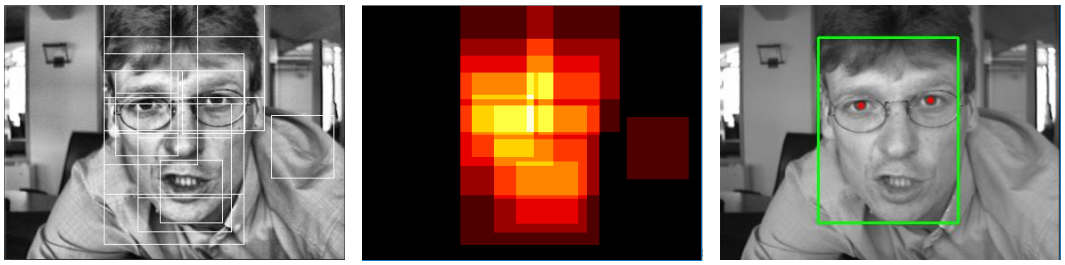


Figura 38: Aplicación de mapas de calor

Non maximum suppression

De forma análoga a los mapas de calor, el método denominado **Non Maximum Suppression** permite combinar todas las detecciones que haya solapadas sobre un mismo objeto en el proceso de detección.

Su implementación difiere en función del tipo de clasificador utilizado. Esto se debe a que algunos clasificadores asocian una probabilidad a cada una de las detecciones positivas de la imagen, por ejemplo, el Bayesiano ingenuo. En cambio, otros clasificadores como SVM indican si un elemento pertenece a una clase u otra, pero como valor absoluto, sin indicar ninguna probabilidad.

En caso de que cada detección tenga asociada una probabilidad se siguen los siguientes pasos:

1. Se descartan todas las detecciones con una probabilidad inferior a un valor determinado, por ejemplo, menos que 0,6. Esto debería eliminar todos los falsos positivos de la imagen.
2. Se busca la detección con un valor de probabilidad más alto.
3. Se eliminan todas las detecciones que tengan solapada más de un 50% de su superficie con el recuadro seleccionado en el punto anterior.
4. Se busca la detección con el siguiente valor más alto y se vuelve al paso 2.

Cuando las detecciones no tienen probabilidad asociada el método arroja peores resultados ya que el rectángulo del paso 2 se escoge más o menos aleatoriamente. Por ejemplo, la detección de mayor tamaño o la que se encuentra más a la izquierda. Esto impide eliminar los falsos positivos y en general proporciona unos resultados relativamente malos.

4.4.- Procesamiento de las imágenes

El reconocimiento facial es un método que requiere una gran uniformidad en la estructura de los datos de entrada. Es imprescindible que las caras que se introducen como entrada al algoritmo sean lo más uniformes posibles de forma que el algoritmo se centre en las

características fisiológicas de cada persona y no se vea alterado por factores de iluminación, ambientales, …

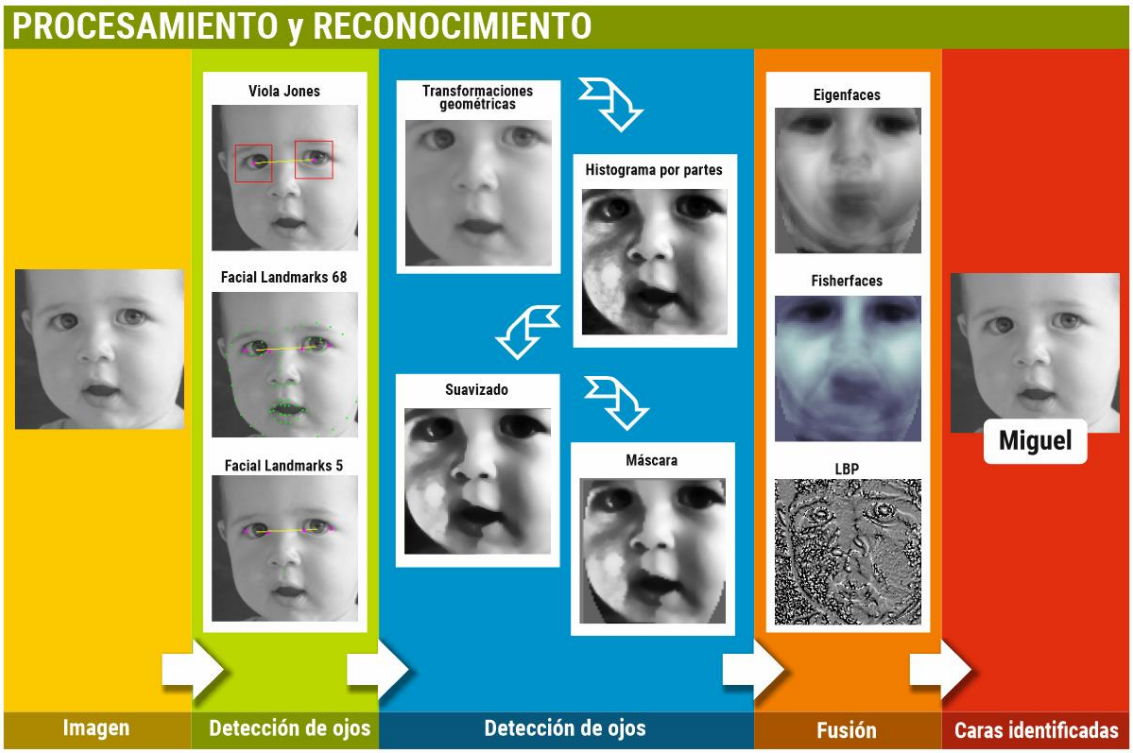


Figura 39: Etapas del preprocesamiento y el reconocimiento

Este proceso de unificación de las imágenes se va a realizar mediante una serie de etapas con las que se pretende conseguir los siguientes objetivos:

- Misma posición respecto a la imagen y misma escala en todas las caras.
- Eliminar factores ambientales, principalmente derivadas de la iluminación y el ruido de la imagen.
- Eliminar partes no relevantes de la cara como puede ser el pelo o en general todas las partes periféricas de la misma.

Para alcanzar estos objetivos se procesarán las caras a través de las siguientes etapas:

- Transformaciones geométricas
- Ecualización del histograma por partes
- Suavizado
- Máscara elíptica

4.4.1.- Transformaciones geométricas

Los algoritmos de detección de imágenes proporcionan como salida el área de la imagen en que se encuentra la cara, pero esta área no tiene ni un tamaño ni una proporción fija. Tampoco está garantizado que la cara esté exactamente centrada en el área detectada. Otra posibilidad es que la cara no se encuentre en una posición totalmente vertical, sino que se encuentre ligeramente ladeada.

De alguna forma es necesario gestionar esta heterogeneidad en la entrada para que todas las caras que se pasen al algoritmo de reconocimiento estén exactamente en la misma posición y tengan la misma escala. Para ello se deben aplicar una serie de transformaciones geométricas:

- **Rotación**, para corregir caras que se encuentren ladeadas.
- **Traslación**, con objeto de que los elementos distintivos de la cara se encuentren en la misma posición respecto a la imagen.
- **Escalado**, ya que puede variar la escala de las caras proporcionadas por el detector.

Como complemento a estas transformaciones geométricas también será necesario realizar una operación de **recorte** para que el tamaño en que se encuentren todas las caras sea el mismo.

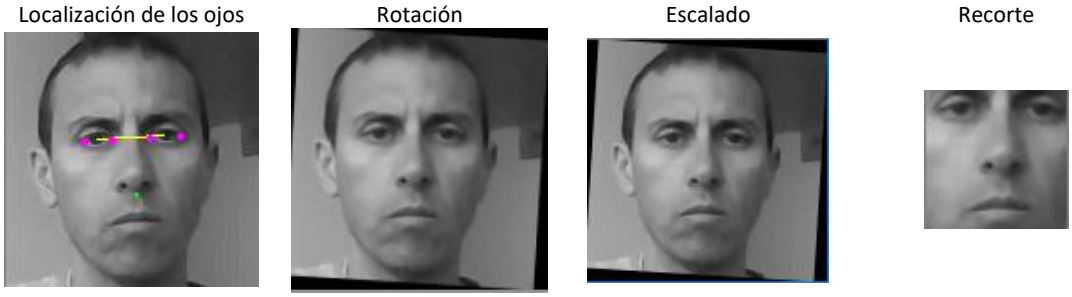


Figura 40: Pasos para aplicar las transformaciones geométricas

Un aspecto clave para conseguir todos estos objetivos es la necesidad de tomar un punto de referencia que sea reconocible en todas las caras y que permita establecer la posición de la cara con respecto a este elemento. Obviamente el candidato perfecto son los ojos ya que, si se obtienen las coordenadas de los centros de ambos se podrá:

- Escalar la imagen de forma que la distancia entre los centros de ambos ojos sea siempre la misma.
- Rotar la imagen, una cara ladeada se traduce en una falta de alineación vertical de los ojos. Por tanto, conociendo la distancia vertical entre los centros de ambos ojos se puede determinar el grado de rotación de la cara.
- Recortar la imagen ya que se puede establecer el centro de los ojos como punto de referencia.

Tras detectar la posición de ambos ojos se aplican las transformaciones para conseguir los siguientes objetivos:

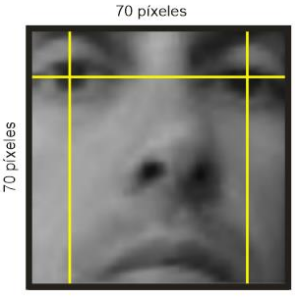


Figura 41: Posición relativa de los ojos

- El tamaño final ha de ser de **70x70 píxeles**, un tamaño que en las diversas pruebas ha demostrado ser adecuado.
- En cuanto a la posición de los ojos respecto a la imagen final, y normalizando los valores de ancho y alto para que se encuentren entre 0 y 1:
    - El centro del ojo izquierdo se debe encontrar en 0,16
    - El centro del ojo derecho se debe encontrar en 0,84
    - La distancia entre ambos centros debe ser 0,68
    - El punto central de los ojos se debe encontrar a 0,14 del borde superior

4.4.2.- Detección de los ojos

Como se puede deducir de lo explicado en el apartado anterior, la fortaleza del proceso de preparación de la cara previamente a su reconocimiento se encuentra en la correcta detección de los ojos en la imagen. Se han utilizado dos métodos diferentes para localizarlos, con objeto de evaluar las ventajas e inconvenientes de cada uno de ellos: el algoritmo de Viola-Jones y el algoritmo para localizar los *facial landmarks*.

Detección de ojos con Viola-Jones

El primer método probado para la detección de los ojos en la cara es el mismo que se ha utilizado para la detección de las caras en la imagen inicial, el algoritmo de Viola Jones. Ya se ha visto que este algoritmo tiene una alta tasa de detección para las caras, pero también tiene una tasa de acierto relativamente aceptable para la detección de los ojos. La propia librería OpenCV incluye ficheros de configuración ya entrenados para la detección de ojos en diferentes situaciones: ojo izquierdo, derecho, con gafas, ….

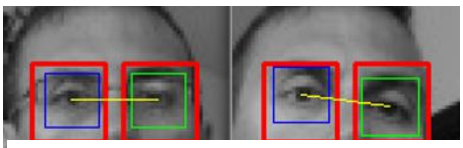


Figura 42: Ejemplos de detección de ojos con cascadas de Haar

Sin embargo, la detección con cascadas de Haar en un proceso relativamente lento que puede suponer un fuerte impacto en el proceso de reconocimiento por lo que es necesario tomar algunas medidas para minimizar este problema. La más relevante consiste en buscar los ojos únicamente en las posiciones en las que es de esperar que estén los ojos. Es decir, en la ventana detectada, la posición de la cara puede variar ligeramente, pero esta variación no es muy grande, por lo que es de suponer que los ojos siempre se encuentren aproximadamente en la misma zona.

En la imagen de la derecha se muestra en color rojo las zonas en las que es de esperar que se ubiquen los ojos y por tanto las únicas áreas donde serán buscados. En ambas imágenes se muestran en color azul y verde las detecciones positivas de los ojos. Reducir el tamaño del área de búsqueda tiene un fuerte impacto en el rendimiento, ya que al aplicar las ventanas deslizantes se obtendrán menos ventanas y por tanto un menor tiempo de procesamiento.

Tal y como se verá posteriormente en el análisis de los resultados, hay dos problemas importantes en la utilización del método de Viola Jones para detectar la posición de los ojos:

- El primero es que, a pesar de minimizar el número de ventanas que se analizan, el tiempo empleado supone un importante impacto en todo el proceso.
- Este elevado consumo de CPU supone un inconveniente importante cuando el objetivo es acercar el tiempo de reconocimiento al tiempo real. Pero aún más importante que la demora introducida es que la tasa de detecciones dista bastante de ser la ideal. Por término medio, mientras que la detección de las caras tiene una tasa de detección con éxito superior al 90%, la tasa de detección con éxito de los ojos desciende hasta un 75%.

  A esta baja tasa de éxito se añade el hecho de que es necesario detectar ambos ojos de la cara para que se puedan aplicar correctamente las transformaciones

geométricas. Se han realizado diversas pruebas[4] para contabilizar el número de detecciones de ojos en una cara y los resultados obtenidos se pueden considerar bastante insatisfactorios.

- En un 7,93% de las caras no se ha detectado la posición de ninguno de los dos ojos.
- En un 45,11% de las caras únicamente se ha podido detectar uno de los dos ojos.
- En únicamente un 46,96% de las caras analizadas se han detectado con éxito ambos ojos.

Esto quiere decir que hay un 53,04% de las caras que no se pueden preparar adecuadamente para el reconocimiento, por lo que, o bien se descartan tras haber empleado un importante tiempo de computo que no llevará a ningún resultado, o bien se envían al reconocedor sin estar correctamente procesadas, lo que provocará que muy probablemente no sea reconocida.

Buscando una alternativa a este problema se ha empleado el método denominado *facial landmarks*, un método que, aunque en principio intuitivamente podría parecer más lento, ya que detecta todos los rasgos característicos de la cara, ha demostrado ser muy rápido y con una altísima tasa de éxito.

Detección de los ojos con facial landmarks

La detección de *facial landmarks*, expresión que podría traducirse como *rasgos faciales*, es un método propuesto por Vahid Kazemi y Josephine Sullivan (Kazemi & Sullivan, 2014) en su artículo titulado *One Millisecond Face Alignment with an Ensemble of Regression Trees*. En este artículo exponen un método que, partiendo de una imagen que contiene el objeto de interés, por ejemplo, una cara obtenida con Viola Jones intenta localizar los puntos clave de esa cara.

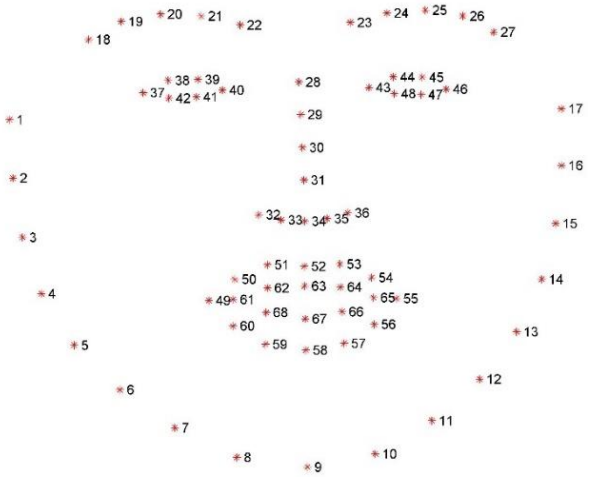


Figura 43: Ubicación de los puntos de facial landmarks 68

Lo que hace especialmente reseñable este método es que, aunque el entrenamiento puede requerir varias horas, una vez realizado éste es muy rápido localizar los rasgos faciales de una cara concreta. En el artículo original habla de un tiempo de detección de 1 milisegundo, y, aunque en la implementación en la Raspberry Pi 3 este tiempo se incremente, indudablemente estará muy por debajo del tiempo requerido por el método propuesto en el apartado anterior donde se aplicaba el método de Viola Jones para detectar ambos ojos.

[4] Las pruebas se han realizado sobre 3 vídeos grabados en diferentes entornos con un total de 1255 fotogramas en los que en todo momento hay una cara. Únicamente se cuentan los ojos buscados en caras positivamente detectadas.

Hay una gran variedad de detectores de rasgos faciales basados en este método que difieren principalmente en el número de puntos que detectan, pero todos tratan de acotar los principales rasgos de una cara: boca, pestañas, ojos, nariz, barbilla. En un principio, en este proyecto se utilizó la implementación disponible en la librería Dlib, que detecta 68 puntos clave de la cara, tal como se puede ver en la figura 43.

El predictor de rasgos faciales de Dlib ha sido entrenado con la base de datos iBUG 300-W[5], cuyas caras tienen previamente anotados cada uno de estos 68 puntos. Sin embargo, conviene destacar que hay otras implementaciones con diferente número de puntos, por ejemplo, la versión original propuesta en el artículo contempla un modelo de 194 puntos entrenado sobre la base de datos HELEN[6].

De todos los puntos obtenidos en la implementación de Dlib, solamente se utilizan los puntos numerados como 37, 40, 43 y 46, que corresponden a los extremos de ambos ojos. Con estas 4 coordenadas es fácil obtener el centro de cada ojo, asumiendo que éste se encuentra en el punto medio entre los extremos laterales del mismo.

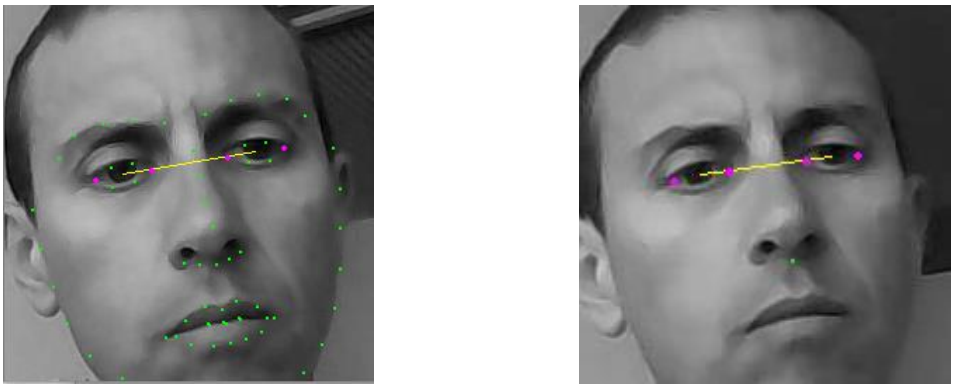


Figura 44: Puntos detectados con los predictores de 68 y 5 puntos

Durante el desarrollo del presente proyecto, se incorporó a la librería Dlib un segundo conjunto de entrenamiento que localiza únicamente 5 puntos clave en la cara: los extremos de ambos ojos y la parte inferior de la nariz. Aunque mucho más limitada, esta información es suficiente para determinar la rotación de la cara, ya que únicamente es necesario conocer la ubicación de los ojos.

Dado que ambos métodos son igualmente válidos para el objetivo, que es calcular la rotación de la cara, se compararon en cuanto a costo computacional, llegando a las siguientes conclusiones:

- El tamaño del fichero de entrenamiento difiere bastante, ocupando el fichero del predictor de 5 puntos apenas 9MB frente a los 97MB del predictor de 68 puntos.
- En cuanto al consumo de CPU también hay una gran diferencia. Las pruebas han demostrado que el tiempo

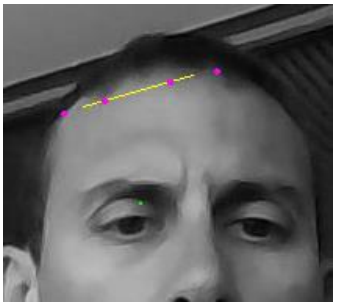


Figura 45: Ejemplo de detección incorrecta en un falso positivo

[5] https://ibug.doc.ic.ac.uk/resources/facial-point-annotations/
[6] http://www.ifp.illinois.edu/~vuongle2/helen/

medio empleado por el predictor de 5 puntos en analizar una cara de 250x250 píxeles en la Raspberry Pi es de 11,3 ms, frente a los 24.4 ms requeridos por el predictor de 68 puntos[7]. Una diferencia del 215%. Hay que destacar que, aunque computacionalmente hay una gran diferencia entre ambos métodos, ambos son igual de eficaces para el objetivo de obtener el ángulo de rotación de la cara. Por ello, se ha determinado utilizado el predictor de 5 puntos.

El único inconveniente que se ha encontrado con la utilización del método de *facial landmarks* es que este método **siempre proporciona un resultado**, incluso aunque la imagen procesada no corresponda a una cara. De esta forma, al analizar un falso positivo proporcionaría unas coordenadas para unos ojos inexistentes pero que validaría el continuar con el proceso de reconocimiento. También puede ocurrir que realice una detección errónea, por lo que las transformaciones geométricas serán incorrectas y por tanto la cara resultante no válida para el reconocimiento.

4.4.3.- Ecualización del histograma por partes

Ya se ha mencionado más veces que las condiciones de iluminación del entorno pueden afectar al rendimiento de los algoritmos de detección y reconocimiento. En el paso previo a la detección se realizaba un preprocesamiento sobre la imagen que pretendía minimizar los efectos de la iluminación aplicando técnicas como la ecualización del histograma o el ajuste gamma.

Estas técnicas ayudan al proceso de detección, pero pueden ser insuficientes si se quiere una preparación óptima de la cara para su reconocimiento. Hay un factor lumínico en concreto que es necesario compensar y es el hecho de que habitualmente, en el mundo real, las fuentes de luz es muy probable que incidan de forma lateral sobre la cara. Esto provoca que la mitad de la cara tenga sombras muy acentuadas mientras que la otra mitad estará mucho más iluminada.

Para solucionar este problema la solución consiste en aplicar la ecualización del histograma. Si se aplica sobre toda la cara simplemente se conseguirá acentuar aún más el contraste entre ambos lados. La solución está en aplicarlo de forma independiente sobre ambas mitades de la cara. De esta forma se consigue estandarizar el brillo y el contraste en cada una de las mitades de la cara.

Si simplemente se aplican los histogramas en ambas mitades y luego se juntan se conseguirá el objetivo de eliminar diferencias entre las mitades, pero habrá un marcado borde entre ambas mitades. Para evitarlo, será necesario aplicar la ecualización del histograma a ambas mitades por separado y también a toda la cara. Tras ello se combinarán las tres imágenes fusionándolas con mayor peso para la imagen completa a medida que se aproxime al centro de la imagen.

Original | Histograma completo | Histograma por partes

Figura 46: Diferencias entre histograma completo y por partes

[7] Resultados generados por el programa facial_landmarks_profiling.py, que realiza 100 iteraciones sobre 6 caras diferentes.

Como puede verse en la figura 46, el resultado cuando se aplica el histograma por partes es muy superior a aplicar la ecualización del histograma sobre toda la cara, eliminando las diferencias de iluminación entre ambas mitades.

4.4.4.- Suavizado y máscara elíptica

Es frecuente que la ecualización del histograma incremente el ruido a nivel de píxel en la imagen por lo que es necesario realizar un **suavizado** para eliminar este ruido. Para ello se aplica un **filtro bilateral** ya que con él se consigue una reducción de ruido y suavizado de la imagen, pero preservando los bordes.

En el filtrado bilateral el valor de intensidad de cada píxel de la imagen es reemplazado por una media ponderada de los valores de intensidad de los píxeles cercanos basándose en una distribución de Gauss que no solamente tiene en cuenta la distancia sino también aspectos como las diferencias de rango o la distancia de profundidad.

Al recortar la cara en el primer paso del procesamiento se eliminaron todas las partes no relevantes para el proceso de reconocimiento como el pelo, el cuello o el fondo de la imagen. Pero es posible que hayan quedado algunos restos de estos elementos periféricos, especialmente en las esquinas de la imagen.

Para eliminarlos se aplicará una **máscara elíptica** sobre la cara. Esta máscara, suponiendo las medidas normalizadas entre 0 y 1, tendrá su centro en las coordenadas (0,5, 0,4), con un radio horizontal de 0,5 y vertical de 0,8. La mascará tendrá un color gris neutro para no crear demasiado contraste con la imagen.

4.4.5.- Resultado final

La imagen obtenida de aplicar todos los pasos anteriores ya estará preparada para ser analizada por el reconocedor.

En la figura 47 se muestran los resultados obtenidos al realizar todas las etapas de procesamiento sobre un conjunto de caras.

Figura 47: Preparación para el reconocimiento sobre conjunto de caras

4.6.- Reconocimiento facial

La implementación del reconocimiento facial es muy sencilla, ya que la implementación de OpenCV incluye toda la lógica para crear un reconocedor.

Al margen de la funcionalidad proporcionada por OpenCV solamente se ha añadido la posibilidad de facilitar los datos de entrenamiento a partir de un directorio en el disco, el cual contiene un subdirectorio para cada uno de los sujetos que el sistema debe reconocer y donde la etiqueta del mismo es el nombre del subdirectorio.

5.- Resultados

El objetivo de este proyecto es evaluar diferentes métodos de detección y reconocimiento facial para determinar su viabilidad de utilización en el procesamiento de vídeo en la Raspberry Pi. Los múltiples pasos necesarios desde la obtención de las imágenes hasta la identificación de las caras en las mismas hacen que sea necesario establecer una metodología que permita evaluar objetivamente los algoritmos utilizados.

Para ello, se realizará un análisis de los resultados en tres puntos clave del proceso de reconocimiento facial:

- **Clasificación de una imagen como cara o no cara**: en el presente proyecto se proponen diferentes combinaciones de características con clasificadores para identificar si una imagen corresponde a una cara o no. El método habitual para evaluar el rendimiento de estos algoritmos es dividir la base de datos de caras y no caras en dos grupos: uno para el entrenamiento del clasificador y otro que servirá para probar el clasificador. Y esto es lo que se hará en esta fase de análisis de resultados, utilizando la métrica denominada *Precision and recall*, que se explicará más adelante, para poder comparar los resultados de los diferentes algoritmos.
- **Detección de las caras en una imagen**: en el punto anterior se evaluará la capacidad del clasificador para identificar una imagen como cara o no cara, pero la detección de caras en imágenes es más compleja, ya que requiere que la imagen original se descomponga en fragmentos mediante el algoritmo denominado ventanas deslizantes y evaluar si cada uno de esos fragmentos es cara o no. Además, será necesario fusionar diferentes detecciones que pueda haber habido sobre una misma cara o eliminar los falsos positivos que pueda haber habido.
  Por tanto, en este apartado se evaluará la capacidad de detectar caras en imágenes más complejas.
- **Reconocimiento de las caras detectadas**: por último, se evaluará la capacidad del sistema para etiquetar correctamente las caras una vez detectadas.

5.1.- Precisión y exhaustividad

Para determinar la calidad de los resultados obtenidos por un algoritmo, es necesario recurrir a una métrica que proporcione una medida del rendimiento de estos. En concreto, se utilizarán la métrica conocida como **precisión y exhaustividad** (*precission and recall*).

Antes de definir estos conceptos, es necesario aclarar cuáles son los posibles resultados que se pueden obtener en un sistema de clasificación para la detección de caras:

- **Verdaderos positivos (VP):** son las imágenes que contienen una cara y que es reconocida como tal por el clasificador.
- **Falsos positivos (FP):** son las imágenes que, a pesar de no contener una cara, son identificadas por el clasificador como si las tuviera.
- **Verdaderos negativos (VN):** en este caso se trata de imágenes que no contienen una cara y que son clasificadas correctamente por el clasificador.

- **Falsos negativos (FN):** son las imágenes que, aunque contienen una cara, son identificadas por el clasificador como si no fuera tal.

Para una visualización más representativa de estos términos es frecuente utilizar la denominada **matriz de confusión**. En una matriz de confusión, cada fila representa las instancias predichas de una clase, mientras que cada columna representa las instancias reales de la clase.

Por lo tanto, para un sistema de clasificación binario como el de detección de caras de este proyecto, la matriz de confusión sería de la forma:

		Realidad	
		Cara	**No cara**
Estimado	**Cara**	Verdadero positivo	Falso positivo
	No cara	Falso negativo	Verdadero negativo

Aunque estos cuatro valores proporcionan información sobre el clasificador, la información realmente útil se obtiene de otras métricas generadas mediante diferentes combinaciones de estos valores. Las métricas que se han utilizado son:

- **Precisión**: es la ratio que indica, de entre todos los resultados identificados como positivos, cuantos lo son realmente. En el caso de la detección de caras, indicaría, de todas las imágenes clasificadas como caras, cuántas corresponden realmente a una cara.
  Un valor muy alto, próximo a uno, indica que hay muy pocas imágenes que han sido reconocidas como caras cuando realmente no lo son, es decir, que hay muy pocos falsos positivos. Por otro lado, un valor bajo indica que hay un número de imágenes que el clasificador ha identificado como caras a pesar de no serlo.
  Se calcula según la fórmula:

$$Precisión = \frac{VP}{VP + FP}$$

- **Exhaustividad**: nos da una medida de cuantos elementos ha identificado como positivos de todos los positivos que se han analizado. En el caso de la detección de caras, indicaría el porcentaje de caras que han sido positivamente reconocidas.
  Un valor alto, próximo a 1, quiere decir que hay muy pocas imágenes de caras cuya clasificación es errónea, es decir, que hay muy pocos falsos negativos.
  Se calcula según la fórmula:

$$Exhaustividad = \frac{VP}{VP + FN}$$

- **Exactitud**: esta métrica proporciona una medida de la fracción de predicciones realizadas correctamente por parte del clasificador.

Un valor alto, próximo a 1, indica que el clasificador tiene una alta tasa de aciertos, es decir, que ha identificado correctamente tanto las imágenes de caras como las imágenes de no caras.
Se calcula según la fórmula:

$$Exactitud = \frac{VP + VN}{VP + VN + FP + FN}$$

5.2.- Resultados del proceso de clasificación

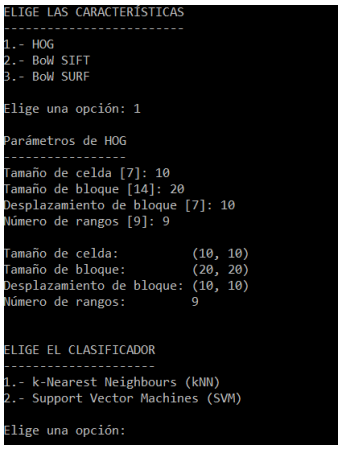
Para evaluar la eficacia de los algoritmos implementados es necesario establecer un marco común que permita establecer comparaciones objetivas a todos ellos. El programa `detector_results.py` es el que contiene toda la lógica para entrenar los algoritmos, almacenar el entrenamiento en un fichero y obtener una medida del rendimiento de estos según la métrica de precisión y exhaustividad.

Para mantener la objetividad en lo posible, se utilizan las mismas imágenes de caras y no caras para entrenamiento en cada algoritmo, y se realizan las comprobaciones sobre las mismas imágenes de prueba. En la figura 48 se puede ver un ejemplo de estas imágenes extraídas de las bases de datos utilizadas.

Figura 48: Ejemplos de imágenes de entrenamiento

Todas las imágenes tienen un tamaño de 70x70 píxeles y se les aplica el mismo preprocesamiento: conversión ponderada a escala de grises y normalización mediante Ecualización de Histograma Adaptativo (CLAHE).

El *dataset* utilizado contiene 2000 imágenes de caras y 2000 imágenes de no caras, de las que se utilizarán 400 imágenes de cada tipo para el entrenamiento y las 1600 imágenes restantes para prueba.



El programa encargado de realizar todos los cálculos se ubica en la carpeta `rasprec/source/programs` y se denomina `trainer.py`. Es un programa interactivo muy sencillo que permite al usuario seleccionar entre diferentes tipos de características (HOG, BoW SIFT y BoW SURF) y también diferentes clasificadores (kNN y SVM), así como seleccionar los parámetros tanto para las características como para el clasificador. Una vez realizada esta selección por parte del usuario entrena el clasificador utilizando el conjunto de imágenes de entrenamiento y guarda el entrenamiento en un fichero. A continuación, evalúa el rendimiento del algoritmo realizando el análisis del conjunto de imágenes de prueba. Finalmente, calcula los valores de la métrica *Precision and Recall* y los muestra por pantalla.

Para cada tipo de características se van a utilizar los siguientes clasificadores:

- **kNN con 1 vecino**: la variante más simple del algoritmo kNN que clasifica un elemento en función del tipo del elemento más próximo a él.
- **kNN con 7 vecinos**: evalúa el tipo de los 7 vecinos más próximos y clasifica el elemento según el tipo de aquellos que sean mayoría.
- **kNN con 35 vecinos**: en este caso se evalúan los 35 vecinos más próximos, determinando el tipo según aquellos que sean mayoría.
- **SVM con kernel lineal, polinómico, RBF y sigmoideo**

5.2.1.- Histogram of Gradients (HOG)

Las primeras pruebas se han realizado utilizando un descriptor de la imagen basado en el Histograma de Gradientes (HOG). Los parámetros que recibe este algoritmo se explican en el apartado 3.2.2, y su selección influye directamente en los resultados obtenidos. En este apartado hay que tener en cuenta que el tamaño de la imagen original condiciona los valores de los parámetros, ya que el tamaño de celda ha de ser un divisor del tamaño total de la imagen y el tamaño de bloque ha de ser un múltiplo del tamaño de celda. Las imágenes del *dataset* que se ha utilizado para el entrenamiento es de 70x70 píxeles.

Tamaño celda	Tamaño bloque	Desp. bloque	Num. bins	Dims.	Verdadero Positivo	Falso Positivo	Verdadero Negativo	Falso Negativo	Exactitud	Precisión	Exhaust.
10	**20**	**10**	**9**	**1296**	**693**	**7**	**699**	**1**	**0,994**	**0,990**	**0,999**
10	20	10	7	1008	692	8	700	0	0,994	0,989	1,000
10	20	5	5	2420	692	8	700	0	0,994	0,989	1,000
10	20	5	7	3388	692	8	699	1	0,994	0,989	0,999
14	28	7	7	1372	691	9	699	1	0,993	0,987	0,999
10	20	10	5	720	691	9	699	1	0,993	0,987	0,999
10	20	5	9	4356	691	9	699	1	0,993	0,987	0,999
14	28	7	5	980	691	9	698	2	0,992	0,987	0,997
7	7	7	9	900	689	11	700	0	0,992	0,984	1,000
7	14	7	5	1620	689	11	700	0	0,992	0,984	1,000
7	14	7	9	2916	689	11	700	0	0,992	0,984	1,000
14	28	14	9	576	691	9	697	3	0,991	0,987	0,996
14	28	14	5	320	690	10	698	2	0,991	0,986	0,997
14	28	7	9	1764	690	10	698	2	0,991	0,986	0,997
7	14	7	7	2268	688	12	700	0	0,991	0,983	1,000
14	28	14	7	448	689	11	698	2	0,991	0,984	0,997
5	10	10	9	1764	688	12	699	1	0,991	0,983	0,999
5	10	5	7	4732	687	13	700	0	0,991	0,981	1,000
10	10	10	7	343	688	12	698	2	0,990	0,983	0,997
7	7	7	7	700	687	13	699	1	0,990	0,981	0,999
5	10	5	5	3380	686	14	700	0	0,990	0,980	1,000
5	10	5	9	6084	686	14	700	0	0,990	0,980	1,000
10	10	10	9	441	686	14	699	1	0,989	0,980	0,999
5	5	5	9	1764	685	15	700	0	0,989	0,979	1,000
5	10	10	7	1372	688	12	696	4	0,989	0,983	0,994
5	5	5	7	1372	684	16	699	1	0,988	0,977	0,999
5	10	10	5	980	690	10	692	8	0,987	0,986	0,989
5	5	5	5	980	681	19	699	1	0,986	0,973	0,999
7	7	7	5	500	681	19	698	2	0,985	0,973	0,997
14	14	14	9	225	687	13	691	9	0,984	0,981	0,987
10	10	10	5	245	684	16	693	7	0,984	0,977	0,990
7	14	14	5	500	677	23	698	2	0,982	0,967	0,997
7	14	14	7	700	675	25	700	0	0,982	0,964	1,000
7	14	14	9	900	675	25	700	0	0,982	0,964	1,000
14	14	14	7	175	680	20	691	9	0,979	0,971	0,987
14	14	14	5	125	673	27	694	6	0,976	0,961	0,991
35	35	35	9	36	614	86	590	110	0,860	0,877	0,848
35	35	35	7	28	642	58	491	209	0,809	0,917	0,754
35	35	35	5	20	675	25	376	324	0,751	0,964	0,676

Figura 49: Resultados obtenidos con diferentes parámetros en HOG+SVM

También hay que tener en cuenta que los parámetros del clasificador determinarán el número de dimensiones que tendrá el descriptor de la imagen, considerando desde este punto de vista que un elevado número de dimensiones puede potencialmente afectar al desempeño del clasificador.

Se han realizado pruebas con diferentes combinaciones de valores en los parámetros de HOG. En la figura 49 se pueden ver los resultados utilizando SVM como clasificador, aunque estos pueden ser extrapolados a los obtenidos con el clasificador k-NN, cuyos resultados han sido muy similares.

Como se puede apreciar en esta figura, los mejores resultados se obtienen para tamaños de celda de 10x10 píxeles, un tamaño de bloque de 20x20 píxeles y un desplazamiento de 10 píxeles, utilizando 9 rangos en el histograma. Por tanto, todas las pruebas se han realizado utilizando estos parámetros de HOG, los cuales, como se puede ver en la tabla, generan descriptores de la imagen de **1296 dimensiones**.

El caso más sencillo es el del clasificador 1NN, que como se puede ver en su matriz de confusión, a pesar de su sencillez arroja buenos resultados.

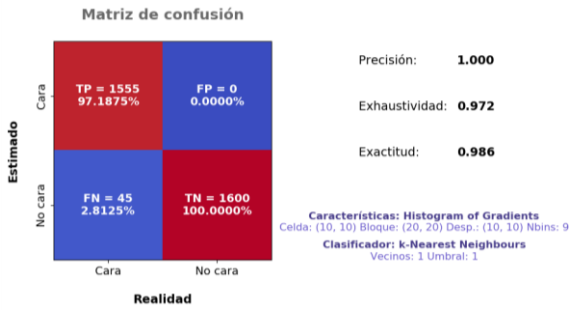


Figura 50: Matriz de confusión HOG+1NN

Si se tienen en cuenta un mayor número de vecinos, se puede ver que los resultados son muy similares, no significando ninguna mejora el aumentar el número de vecinos a tomar en consideración.

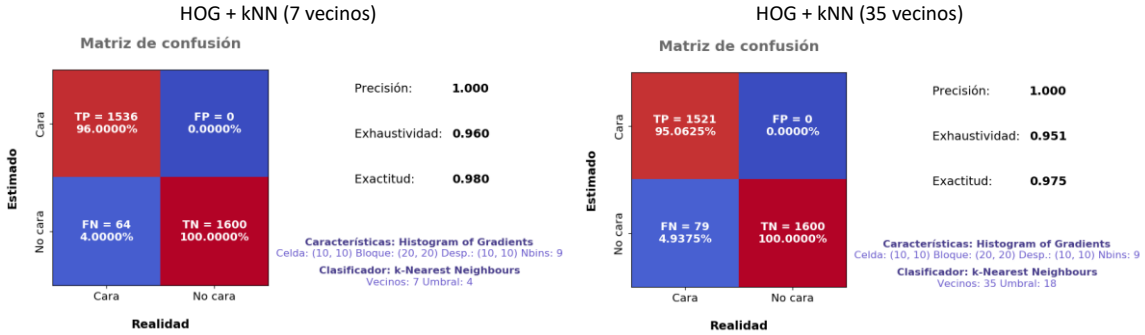


Figura 51: Matrices de confusión HOG+kNN

A continuación, en la figura 52, se muestran los resultados utilizando el clasificador Support Vector Machines (SVM). Hay pocas diferencias con respecto al uso de kNN. Hay algo menos de un 1% de falsos positivos frente a ninguno que había cuando se utilizaba kNN. Por el contrario, el número de falsos negativos no supera el 3% en función del tipo de kernel utilizado, mientras

que en el caso de kNN había aproximadamente un 5% de falsos negativos. Esto indica que en general, el clasificador SVM tiende a clasificar como no cara las imágenes más dudosas, evitando así los falsos positivos, pero en detrimento de un mayor número de falsos negativos.

Algo que llama la atención es que la utilización de un kernel sigmoideo proporciona unos resultados totalmente inválidos, clasificando prácticamente todas las imágenes como caras, tanto aquellas que lo son como las que no lo son.

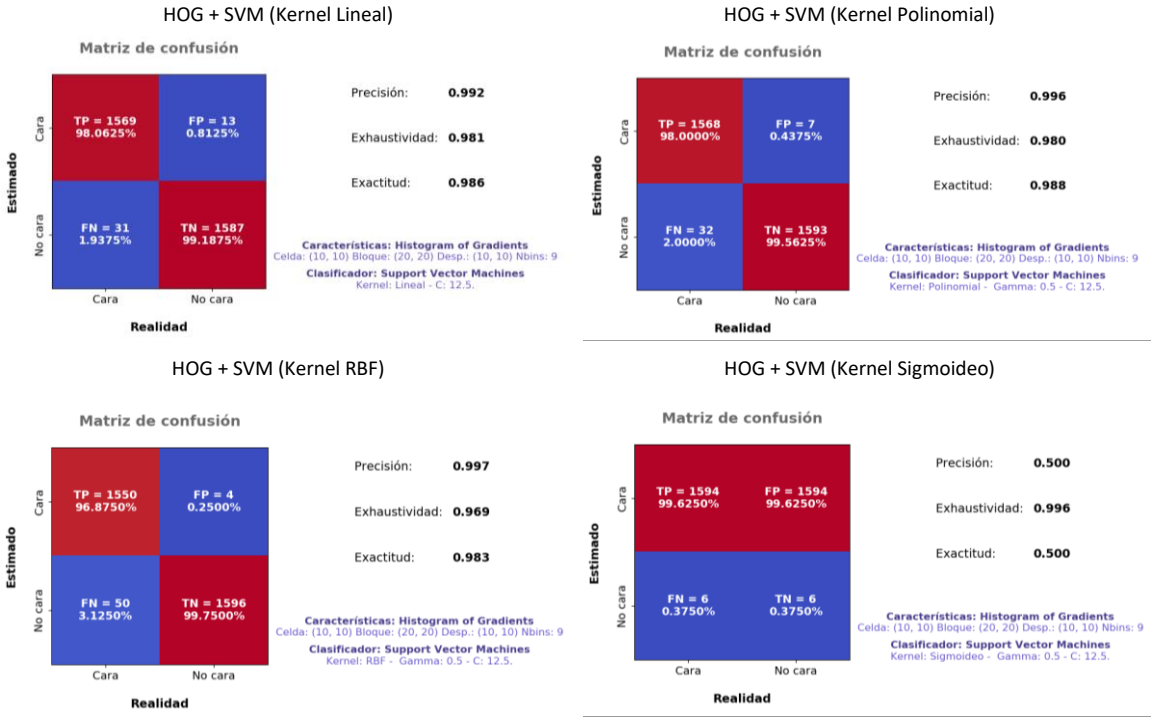


Figura 52: Matrices de confusión HOG+SVM

5.2.2.- BoW SIFT

El siguiente tipo de características es **Scale-Invariant Feature Transform (SIFT)**, que debido a sus características debe ser utilizado en conjunto con el algoritmo **Bolsa de Palabras Visuales (BoW)** para poder ser procesador por un clasificador. En este caso, el número de dimensiones del espacio sobre el que se trabajará está determinado por el número de palabras que tendrá el diccionario de BoW. El primer paso, por tanto, ha sido determinar el número de palabras que mejores resultados proporciona, por lo que se han realizado pruebas con diferentes números de palabras en el diccionario. Los resultados se pueden ver en la figura 53.

Se puede apreciar claramente que los resultados mejoran según se aumente el número de palabras del diccionario hasta alcanzar aproximadamente las 128 palabras. Por encima de este valor, empiezan a empeorar rápidamente los resultados, siendo totalmente inválidos para valores por encima de 256 palabras. Por tanto, en todas las pruebas se utilizarán diccionarios de **100 palabras**, trabajando por tanto en un espacio 100-dimensional.

Tamaño Diccionario	Verdadero Positivo	Falso Positivo	Verdadero Negativo	Falso Negativo	Exactitud	Precisión	Exhaust.
16	91,063	21,563	78,438	8,938	0,848	0,081	0,091
32	90,750	20,750	79,250	9,250	0,850	0,814	0,907
48	91,750	16,375	83,625	8,250	0,877	0,917	0,849
64	88,375	12,063	87,938	11,625	0,882	0,884	0,880
96	88,438	10,500	89,500	11,563	0,890	0,884	0,894
128	76,250	8,750	91,250	23,750	0,838	0,897	0,762
160	65,938	7,250	92,750	34,063	0,793	0,901	0,659
192	30,438	4,125	95,875	69,563	0,632	0,304	0,881
240	4,063	2,625	97,375	95,938	0,507	0,607	0,041
256	2,063	3,188	96,813	97,938	0,494	0,393	0,021
384	0,000	0,750	99,250	100,000	0,496	0,000	0,000
512	0,000	1,000	99,000	100,000	0,495	0,000	0,000

Figura 53: Resultados obtenidos con diferentes tamaños de diccionario en BoW SIFT

A continuación, en las figuras 54 y 55, se muestran las matrices de confusión para el clasificador kNN con 1, 7 y 35 vecinos respectivamente.

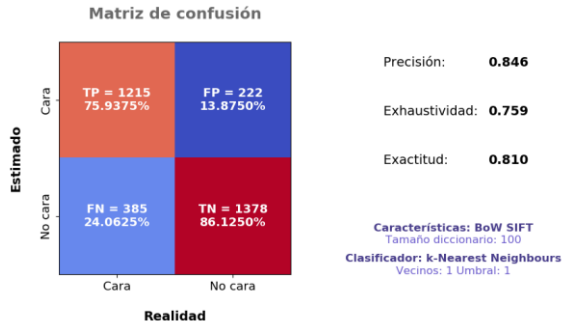
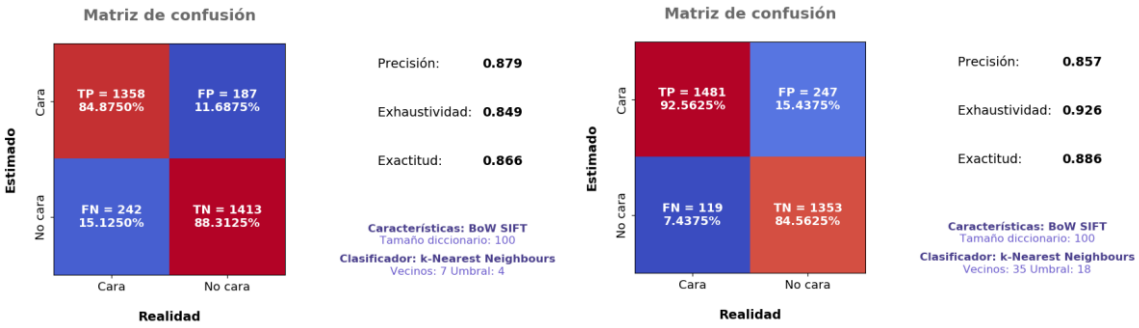


Figura 54: Matriz de confusión BoW SIFT + 1NN



BoW SIFT + kNN (7 vecinos) BoW SIFT + kNN (35 vecinos)

Figura 55: Matrices de confusión BoW SIFT + kNN

Es fácil constatar que las características basadas en SIFT son peores para el reconocimiento facial que HOG ya que las diferentes configuraciones de kNN proporcionan de media entre un 10% y un 15% de falsos positivos, mientras que los falsos negativos llegan incluso hasta un 25% en el caso de 1NN.

Los resultados usando el clasificador SVM son incluso peores como se ve en la figura 56. Obviando el kernel sigmoideo, que sigue proporcionando resultados muy alejados de lo razonable, el resto de kernels tienen unas respuestas bastantes alejadas de los resultados que se habían obtenido con HOG, con tasas de falsos resultados que llegan incluso al 30%.

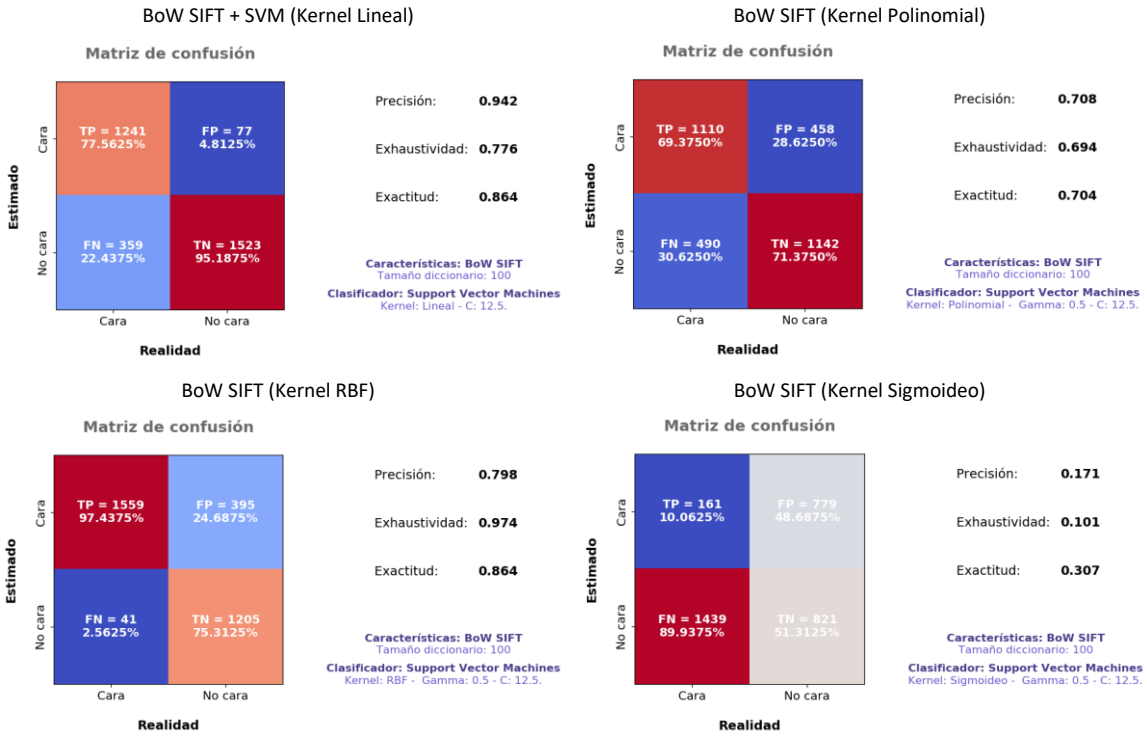


Figura 56: Matrices de confusión BoW SIFT + SVM

5.2.3.- BoW SURF

Las últimas características que se han evaluado son las basadas en **Speeded Up Robust Features (SURF)**, que al estar basadas también en la localización de puntos de interés requieren ser utilizadas junto con **Bag of Visual Words (BoW)**.

Como se ha hecho en los apartados anteriores, el primer paso será determinar el número de dimensiones óptimo, por lo que se realizan pruebas sobre el conjunto de imágenes de entrenamiento con diferentes tamaños de diccionario. Estas pruebas se han realizado con el algoritmo kNN con 7 vecinos y los resultados se muestran en la figura 57.

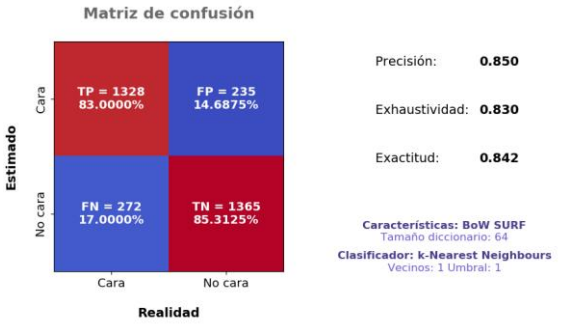
Tamaño Diccionario	Verdadero Positivo	Falso Positivo	Verdadero Negativo	Falso Negativo	Exactitud	Precisión	Exhaust.
16	93,313	24,813	75,188	6,688	0,790	0,933	0,843
32	94,000	14,313	85,688	6,000	0,898	0,868	0,940
48	92,875	15,688	84,313	7,125	0,886	0,855	0,929
64	89,125	14,428	85,563	10,875	0,873	0,861	0,891
96	29,750	5,938	94,063	70,250	0,619	0,834	0,297
128	7,750	3,250	96,750	92,250	0,522	0,077	0,705
160	2,313	2,500	97,500	97,688	0,499	0,481	0,250
192	0,563	2,188	97,813	97,813	0,492	0,205	0,006
240	0,250	1,563	98,438	99,750	0,493	0,138	0,003
256	0,375	1,625	98,375	99,625	0,494	0,188	0,004

Figura 57: Resultados obtenidos con diferentes tamños de diccionario en BoW SURF

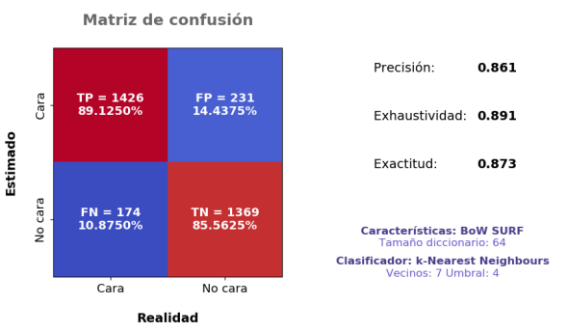
Aquí llama la atención que el número de falsos negativos crece en gran medida por encima de las **64 dimensiones**. Por ello, este será el tamaño de diccionario utilizado en las siguientes pruebas.

Las primeras matrices de confusión reflejan los resultados obtenidos para el clasificador kNN con 1, 7 y 35 vecinos respectivamente.

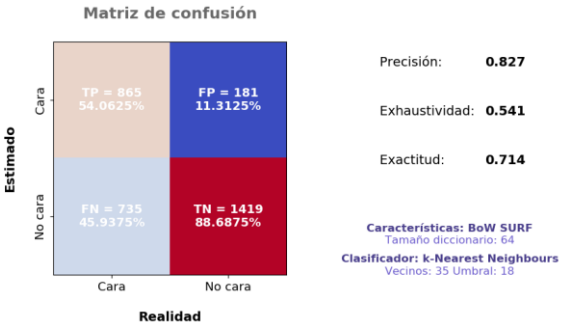
BoW SURF + 1NN



BoW SURF + kNN (7 vecinos)



BoW SURF + kNN (35 vecinos)



BoW SURF + kNN (35 vecinos, umbral=11)

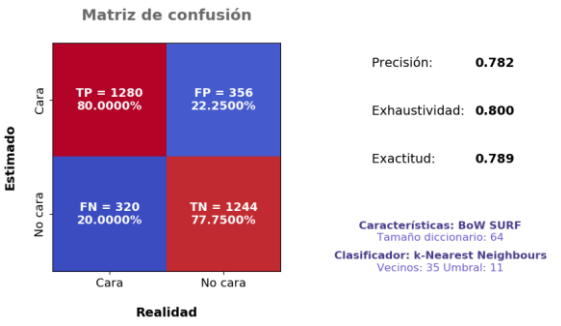
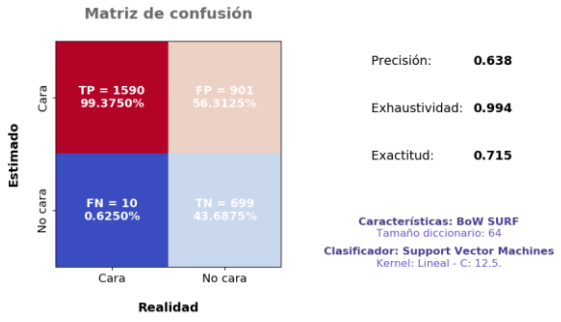


Figura 58: Matrices de confusión BoW SURF + kNN

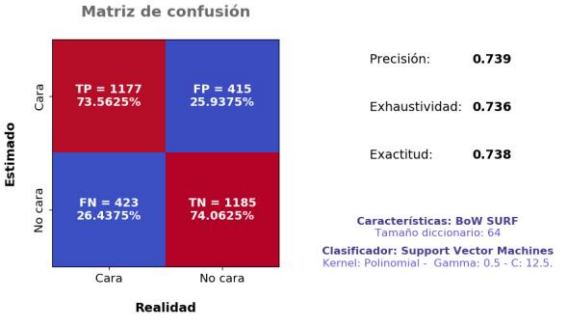
Los resultados obtenidos son muy similares a los que se habían obtenido con SIFT, con la excepción del caso en que se toman en consideración los 35 vecinos más próximos. Aquí se aprecia una excesiva tendencia a clasificar imágenes como no caras. Una forma de corregir esta tendencia es rebajar el umbral, es decir, el número de vecinos que tienen que ser cara para que la imagen analizada sea cara. Si se reduce este valor a 11, es decir, que se considere que la imagen analizada es una cara si por lo menos 11 de los 35 elementos más próximos son caras obtendremos unos valores relativamente más aceptables.

Por último, se muestran las matrices de confusión cuando se utiliza el clasificador SVM. Los mejores resultados se obtienen al utilizar un kernel RBF, aunque siguen siendo peores que los obtenidos con las características HOG.

BoW SURF + SVM (Kernel Lineal)



BoW SURF (Kernel Polinomial)



BoW SURF (Kernel RBF)

BoW SURF (Kernel Sigmoideo)

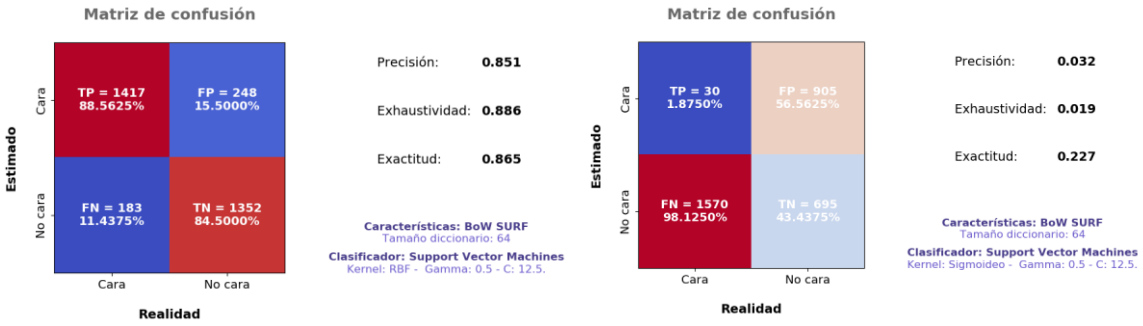


Figura 59: Matrices de confusión BoW SURF + SVM

5.2.4.- Comparativa

En las figuras 60, 61 y 62 se muestra un resumen de los resultados obtenidos para cada uno de los tipos de características utilizados.

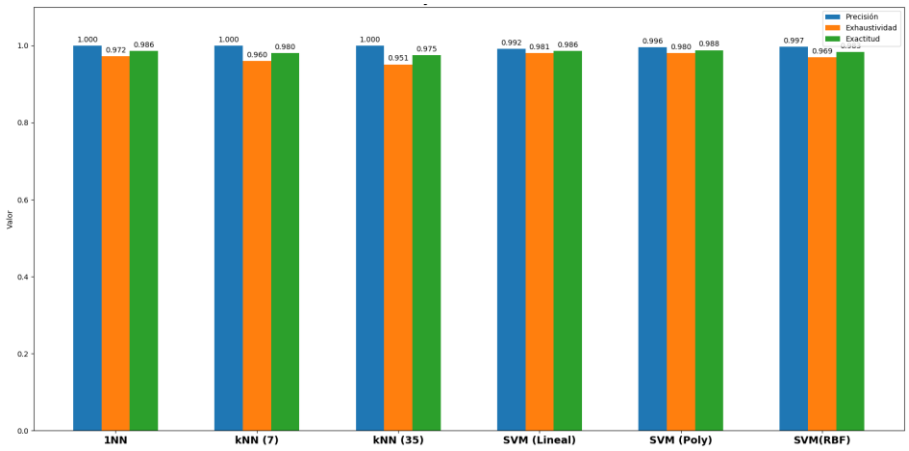


Figura 60: Resultados Precision and Recall HOG

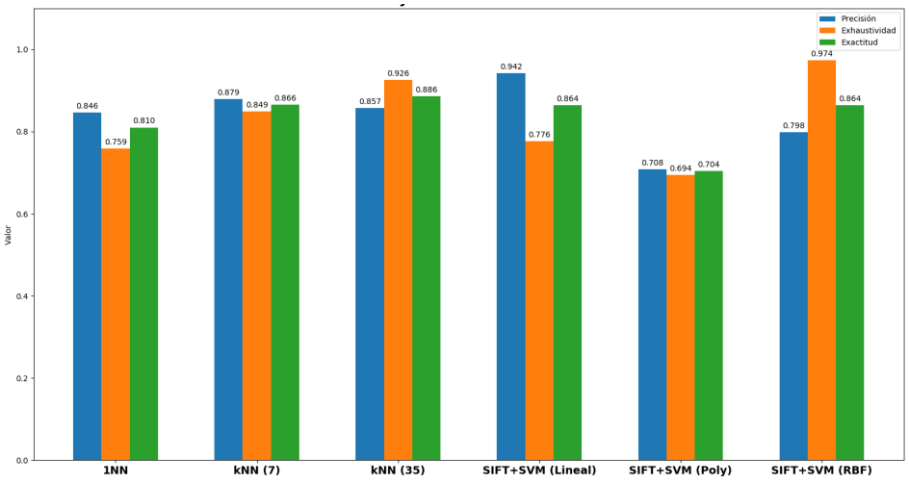


Figura 61: Resultados Precision and Recall BoW SIFT

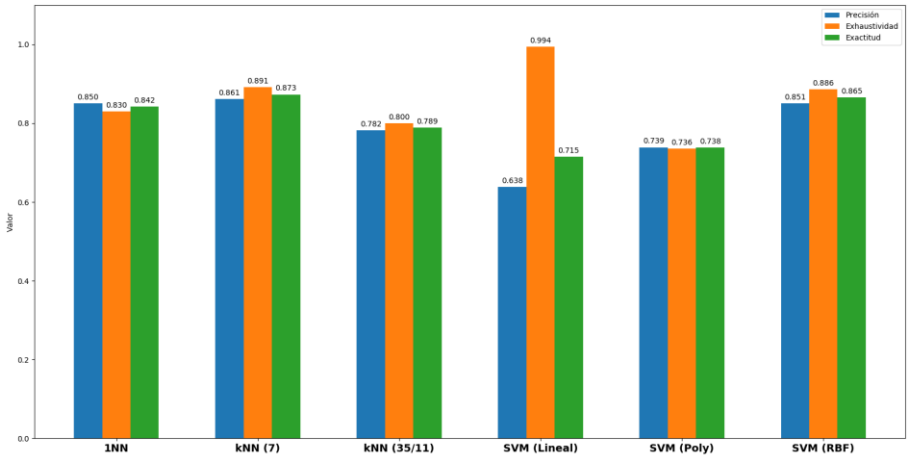


Figura 62: Resultados Precision and Recall BoW SURF

Como ya se reseñó anteriormente, aquí se constata que los mejores resultados son los obtenidos cuando se utiliza HOG como descriptor de la imagen.

Otro factor para tener en cuenta para determinar el algoritmo más adecuado es el tiempo requerido para analizar cada imagen. En la siguiente imagen se muestran los tiempos medios de evaluación de cada una de las imágenes de prueba, medidos en milisegundos.

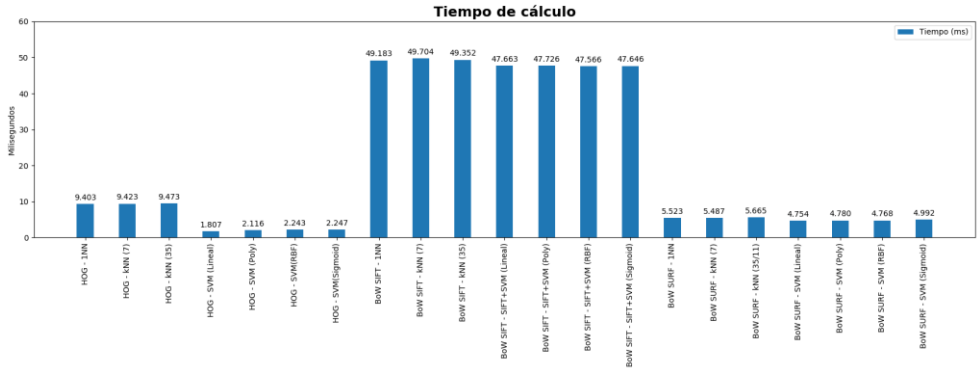


Figura 63: Tiempos de cálculo de los algoritmos de clasificación

Atendiendo a estos datos, se pueden extraer las siguientes conclusiones:

- El impacto en el tiempo de computación depende exclusivamente del tipo de características utilizadas, siendo muy pequeña la diferencia entre diferentes clasificadores.
- La excepción al punto anterior se encuentra en las características de HOG. Muy probablemente, la ventaja de SVM frente a kNN se deba a que es más rápido para espacios con un gran número de dimensiones. Hay que tener en cuenta que las pruebas se han realizado en un espacio con 1296 dimensiones. En el caso de BoW SURF, como el espacio es de únicamente 64 dimensiones, las diferencias entre kNN y SVM son prácticamente inapreciables.
- El elevado tiempo requerido para calcular las características de BoW SIFT hace inviable este método para su uso para procesar vídeo.

Por tanto, y teniendo en cuenta tanto los resultados obtenidos con la métrica *Precision and recall* como los tiempos de computación requeridos, el método más adecuado sería el que combina las características de HOG con el clasificador SVM, utilizando un kernel poligonal.

5.3.- Resultados de la detección facial

El objetivo de esta fase de análisis es evaluar la detección de la posición de las caras dentro de una imagen si las hubiera. Por tanto, involucra la descomposición de la imagen en partes mediante el algoritmo de las ventanas deslizantes y el tratamiento de las detecciones múltiples, así como la eliminación de falsos positivos.

Para mantener la objetividad al comparar los diversos detectores, se tendrán en cuenta los siguientes aspectos:

- Como es un procedimiento supervisado, será necesaria una serie de imágenes que incluyan caras y cuya posición sea conocida. La base de datos que mejor se adapta a estos requisitos es BioID, ya que las imágenes incluyen parte de fondo además de la cara y dispone de metadatos con la ubicación de los ojos. En la figura 64 se puede ver un ejemplo de imagen de esta base de datos con las coordenadas de los ojos que se indican en los metadatos.

Figura 64: Posición de los ojos según los metadatos de BioID

- Se analizará el comportamiento del detector que mejores resultados obtuvo en el apartado anterior, HOG + SVM, así como el detector de Viola Jones, que está implementado en OpenCV.
- Se utilizará nuevamente la métrica *Precision and recall*, aunque hay que tener en cuenta algunas cuestiones:
    - Se considerarán **verdaderos positivos** a los rectángulos detectados en cuyo interior se encuentran las coordenadas de los ojos almacenadas en los metadatos del *dataset*. Esto implica que no se valora completamente la calidad de la detección realizada, considerándose como válida, por ejemplo, una detección que incluya la cara y también parte del fondo.
    - Los **verdaderos negativos** no se evaluarán, ya que todas las imágenes de la base de datos incluyen una cara.
    - Los **falsos positivos** son detecciones que no incluyan ambos ojos dentro del rectángulo detectado. Aunque se indican mediante un porcentaje respecto al total de imágenes procesadas, hay que tener en cuenta que en una misma imagen se puede detectar más de un falso positivo, por lo que hipotéticamente el número de falsos positivos podría ser superior al número de imágenes analizadas.
    - Los **falsos negativos** son las imágenes en las que no hay ninguna detección que incluya en su interior las coordenadas de ambos ojos.

- El parámetro más relevante del detector es el **factor de escala** del algoritmo de las ventanas deslizantes, por lo que se realizarán varias iteraciones con diferentes valores.

Para realizar las pruebas y generar las gráficas se ha desarrollado el programa `detector_results.py`, ubicado en la carpeta `rasprec/source/programs` que permite seleccionar diferentes parámetros para realizar las pruebas.

5.3.1.- Viola Jones

El primer algoritmo analizado es el de Viola Jones, que hay que recordar que está implementado directamente en OpenCV y que ya incluye sus propios ficheros de entrenamiento. Estos ficheros están entrenados para detectar diversos objetos tales como caras, caras con gafas, ojos, cuerpo completo, …. Para las pruebas se ha utilizado el fichero de entrenamiento `haarcascade_frontalface_default.xml` que es uno de los ficheros con el entrenamiento para detectar caras desde una posición frontal.

Como se puede apreciar en la figura 65, los resultados responden a lo que se podría esperar intuitivamente. A medida que aumenta el factor de escala, y por tanto que disminuye el número de fragmentos en que se descompone la imagen, se reducen las caras detectadas positivamente, aunque también se reducen los falsos positivos.

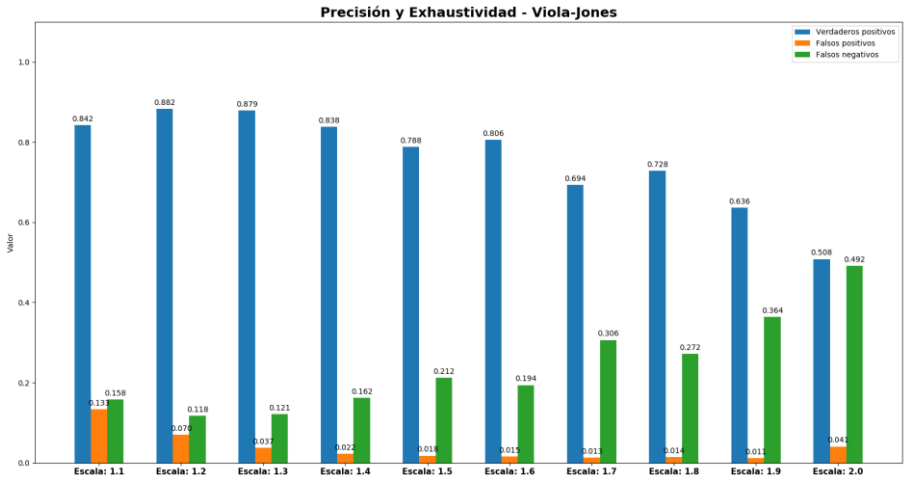


Figura 65: Resultados Precision and Recall de Viola Jones

En la figura 66 se analizan los tiempos empleados en la detección por imagen. Como es de esperar, los tiempos de procesamiento son inversamente proporcionales al factor de escala.

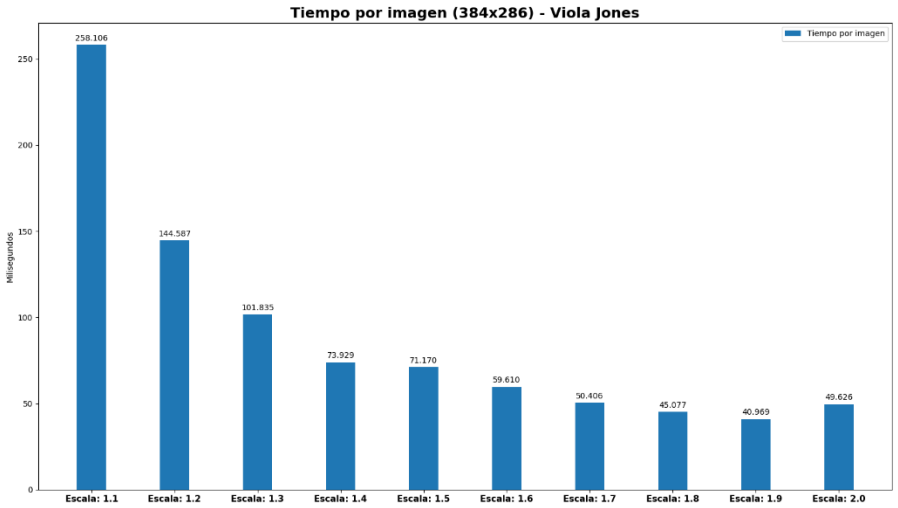


Figura 66: Tiempo de ejecución por imagen de Viola Jones

Una conclusión que se puede extraer directamente a la vista de estos datos es la imposibilidad de implementar un sistema de reconocimiento facial en tiempo real en la Raspberry Pi utilizando este método ya que una tasa de 24 *frames* por segundo requiere que el tiempo dedicado a cada fotograma no exceda de 42 ms.

Combinando los resultados de ambas métricas, un factor de escala que supone un compromiso medio entre tiempo de procesamiento y fiabilidad de la detección sería en torno a 1.4, que proporciona unas tasas de detección bastante aceptables a una tasa aproximada de 14 fps, aunque hay que tener en cuenta que aún habría añadir el tiempo requerido para el proceso de reconocimiento facial.

5.3.2.- HOG+SVM

El otro algoritmo analizado es el que mejores resultados proporcionó en el análisis de los resultados de los clasificadores, que era la combinación de las características de HOG con el clasificador SVM. En este caso, el algoritmo de detección está implementado en todos sus procesos, desde la descomposición de la imagen original mediante ventanas deslizantes hasta la fusión de todas las detecciones. El algoritmo de fusión que se ha utilizado es el basado en mapas de calor, que es el que mejores resultados ha proporcionado en las pruebas. Al ser una implementación propia, hay un nuevo parámetro no disponible en la implementación del algoritmo de Viola Jones de OpenCV, que es el desplazamiento de las ventanas deslizantes al descomponer la imagen original en subimágenes.

Las primeras pruebas se realizaron con un desplazamiento equivalente al 50% del tamaño de la ventana, es decir, de 35 píxeles. Los resultados obtenidos se muestran en la figura 67.

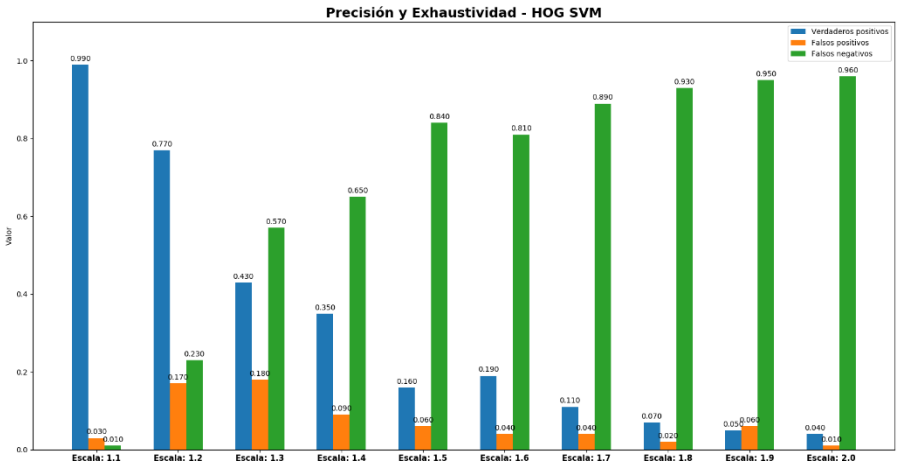


Figura 67: Resultados Precision and Recall de HOG SVM con mapas de calor

En esta ocasión llama la atención lo rápidamente que degeneran las detecciones positivas en cuanto aumenta el factor de escala. Aunque si contrastamos los valores anteriores con los tiempos empleados vemos que los tiempos requeridos son inferiores al caso de Viola Jones.

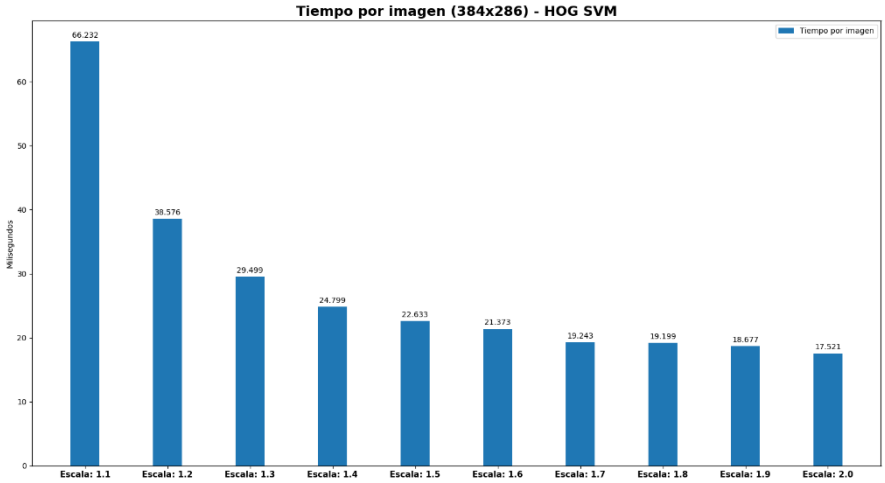


Figura 68: Tiempo de ejecución por imagen de HOG SVM con mapas de calor

Estos resultados se pueden mejorar reduciendo el valor de desplazamiento entre diferentes ventanas. Por ejemplo, los resultados para un valor de desplazamiento de 20 píxeles se pueden considerar bastante mejores.

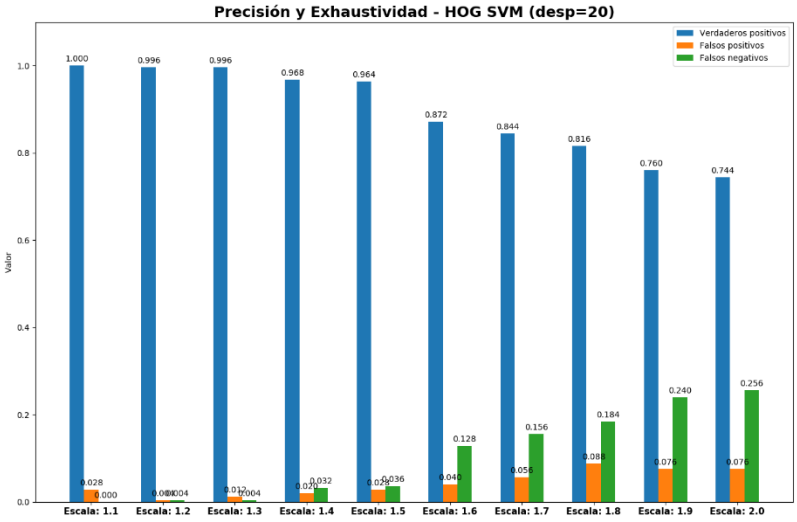


Figura 69: Resultados Precision and Recall HOG SVM con desplazamiento de 20 píxeles

En este caso se ha producido una notable mejoría en las detecciones de caras, siendo los resultados incluso mejores que los que se habían obtenido con el método de Viola Jones. La parte negativa de la reducción del desplazamiento entre ventanas es el mayor número de ventanas a procesar y, por tanto, el incremento en el tiempo de procesamiento requerido.

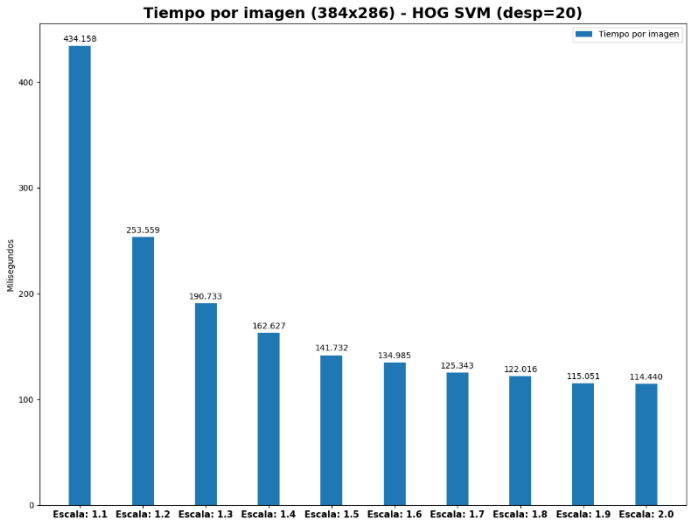


Figura 70: Tiempo de ejecución por imagen para HOG SVM con desplazamiento de 20 píxeles

Como se ve, lo tiempos han aumentado proporcionalmente al número de ventanas analizadas, no superando en ningún momento la tasa de 10 frames por segundo. Una alternativa a este problema es aprovecharse de la disponibilidad de varios núcleos del procesador de la Raspberry Pi y analizar paralelamente diferentes ventanas en diferentes hilos. Esto proporciona una importante mejora en los tiempos requeridos, aproximándose a los que se obtuvieron con el algoritmo de Viola-Jones.

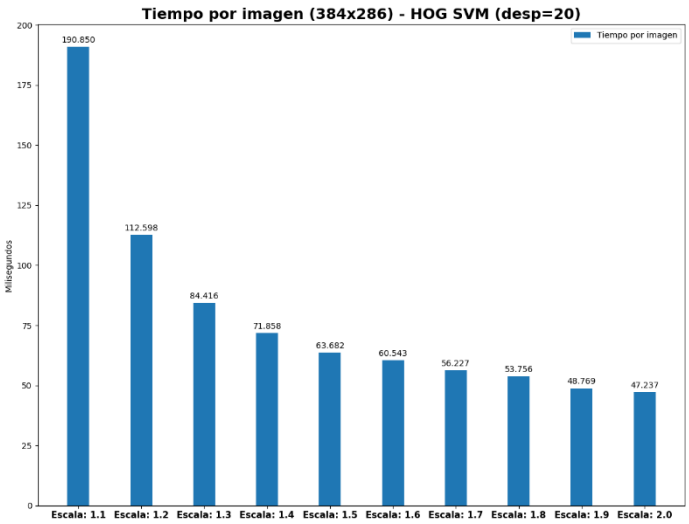


Figura 71: Tiempo de ejecución por imagen de HOG SVM con desplazamiento de 20 píxeles y multithreading

Un punto para destacar antes de finalizar es respecto a la calidad de los resultados de las detecciones con HOG y SVM. El criterio que se ha seguido para determinar que una detección es positiva es simplemente que ambos ojos se encuentren ubicados dentro del rectángulo detectado. Como se ve en las siguientes imágenes, puede ocurrir que haya detecciones positivas, pero de mala calidad, haciéndolas inviables para el proceso de reconocimiento de la cara. Esto puede ocurrir porque la detección no ocupe toda la cara o bien porque el área detectada exceda del tamaño de la cara.

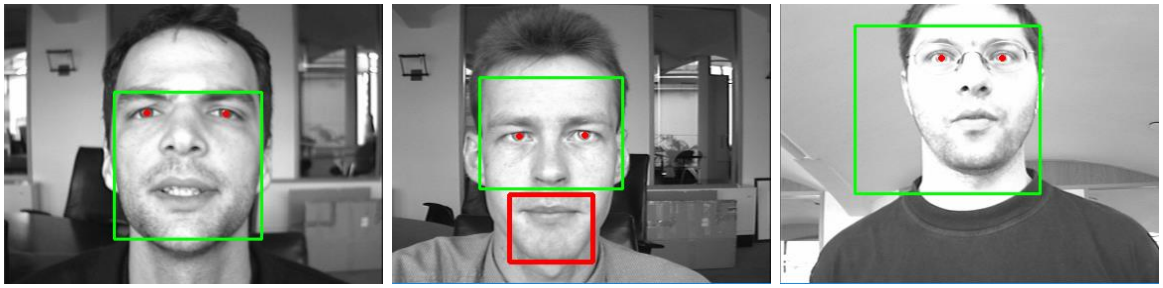


Figura 72: Detecciones consideradas positivas, pero no aptas para reconocimiento facial

5.4.- Resultados del reconocimiento facial

La última fase en el proceso de análisis de los resultados es el que corresponde al reconocimiento facial. En este caso el objetivo es entrenar el reconocedor y aplicarlo a diversas imágenes de pruebas con objeto de evaluar la precisión con que se identifica a qué sujeto pertenece cada cara.

Hay una serie de aspectos previos que se deben tener en consideración antes de proceder a analizar los resultados:

- El programa con el que se han generado los resultados se encuentra en el mismo directorio que los de los pasos anteriores y se denomina `recognizer_results.py`. Ofrece un menú donde seleccionar el tipo de prueba que se quiere realizar, correspondiendo con cada uno de los apartados que se verán a continuación.

- Para esta parte es necesaria una base de datos que disponga de varias imágenes de cada sujeto y que estén etiquetadas. Se ha optado por escoger la base de datos de caras de Caltech, que incluye diversas imágenes de 18 sujetos (se han descartado algunos sujetos cuyo número de imágenes disponibles es insuficiente para disponer de un conjunto de imágenes de entrenamiento y de prueba). Cada sujeto tiene entre 20 y 25 imágenes, de las que se escogen aleatoriamente 10 para el entrenamiento del reconocedor y el resto se utilizan para extraer resultados de la eficiencia de los algoritmos.
- Para independizar los resultados de esta fase de los algoritmos de las fases anteriores, se han extraído las caras detectadas de las imágenes y se han almacenado, eliminando manualmente todos los falsos positivos. De esta forma, los errores del proceso de detección no interfieren con los resultados del proceso de reconocimiento. En la figura 73 se muestra un ejemplo de las imágenes de caras que se han utilizado.

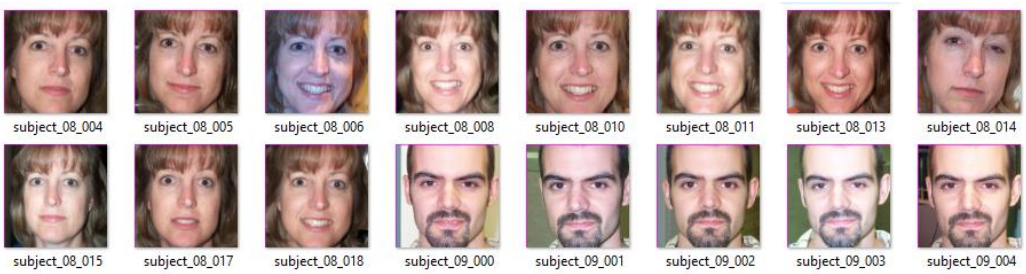


Figura 73: Ejemplos de imágenes de la base de datos utilizada para pruebas de reconocimiento

- El proceso de reconocimiento implica dos fases: la preparación de las caras y el reconocimiento propiamente dicho.

5.4.1.- Eigenfaces

El primer algoritmo analizado es **Eigenfaces**. Este algoritmo únicamente requiere de dos parámetros: el número de componentes y el umbral. Los resultados obtenidos con variaciones de umbral no se reproducen aquí ya que su influencia se centra en el discernimiento entre detecciones incorrectas y desconocidas. Un valor umbral tenderá a asociar la cara con un sujeto, aunque este se encuentre alejado en el espacio. Reducir el valor umbral implica que las caras que no tengan ningún sujeto próximo en el espacio se clasifiquen como desconocidas.

Como se ve en la figura 74, el número de componentes tiene una influencia relativamente baja en el número de identificaciones correctas, manteniéndose este valor por encima del 90% independientemente del número de componentes escogido. Si que afecta en el número de imágenes que no identifica correctamente, ya que con valores muy bajos siempre etiqueta las imágenes, aunque sea incorrectamente, mientras que con valores altos tiende a etiquetas las imágenes dudosas como desconocidas.

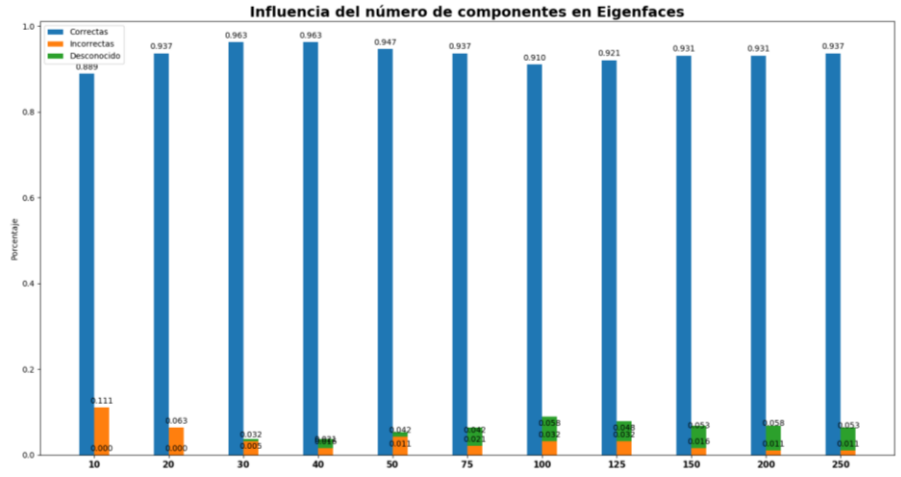


Figura 74: Influencia del número de componentes en Eigenfaces

Si analizamos el tiempo empleado en reconocer cada cara, hay dos aspectos que llaman la atención:

- El tiempo necesario para etiquetar una cara es muy bajo y es proporcional al número de componentes. Por ejemplo, para 40 componentes únicamente requiere 6 milisegundos de procesador.
- El mayor consumo de CPU se encuentra en la etapa de procesamiento donde se prepara la imagen para su reconocimiento, requiriendo en torno a 300 ms por imagen. Posteriormente, en el apartado 5.5 se analizará la causa exacta de este elevado tiempo de computación y se estudiarán las posibles soluciones.

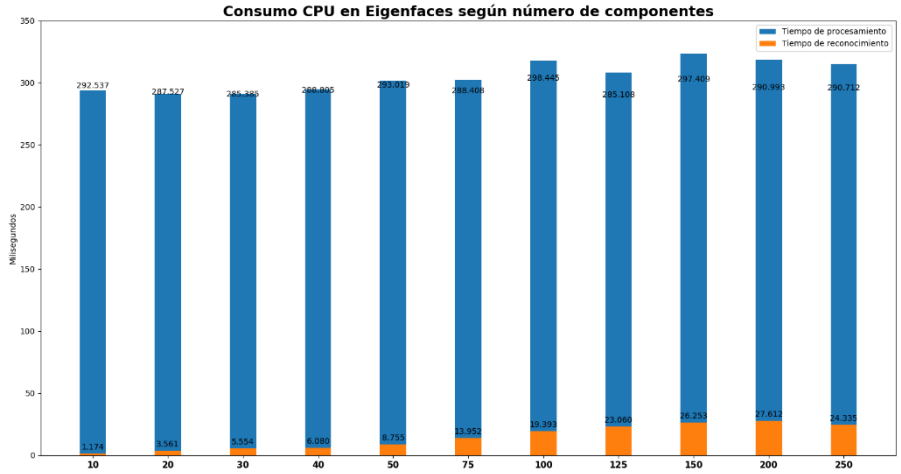


Figura 75: Consumo de CPU con Eigenfaces según el número de componentes

5.4.2.- Fisherfaces

El método de Fisherfaces es muy similar al anterior, pero los números que proporciona son significativamente mejores. Con diversos valores en el número de componentes, la tasa de identificaciones correctas siempre se encuentra por encima del 96% de identificaciones positivas, llegando prácticamente al 99% para valores entre 30 y 50 componentes.

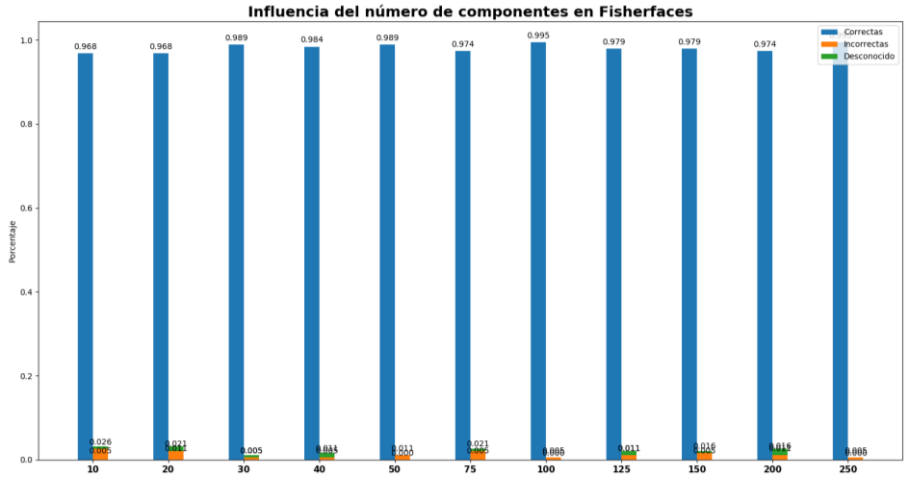


Figura 76: Influencia del número de componentes en Fisherfaces

En cuanto al tiempo de computación requerido, vuelve a surgir el problema visto en el apartado anterior. La mayor carga computacional es provocada por la fase de procesamiento, pero en este caso, la fase de reconocimiento requiere menos tiempo que en el caso de Eigenfaces, y, además, es independiente del número de componentes, manteniéndose siempre en torno a los 3 milisegundos.

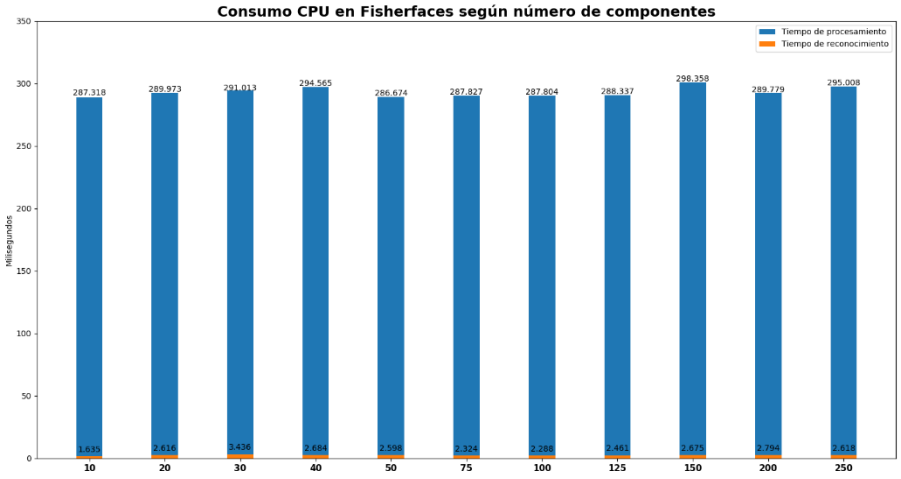


Figura 77: Consumo de CPU con Fisherfaces según el número de componentes

5.4.3.- Local Binary Patterns

El último método analizado es LBP. En las siguientes gráficas se muestran los resultados obtenidos con este método, de las que se pueden extraer las siguientes conclusiones:

- La tasa de identificaciones correctas es alta, equiparable a los otros dos algoritmos. Destaca en la gráfica que no hay identificaciones marcadas como desconocida, pero se debe al valor umbral escogido, tal como se explicó en el apartado anterior.
- El tiempo de reconocimiento es muy alto comparado sobre todo con Fisherfaces, llegando a ser hasta 30 veces superior.

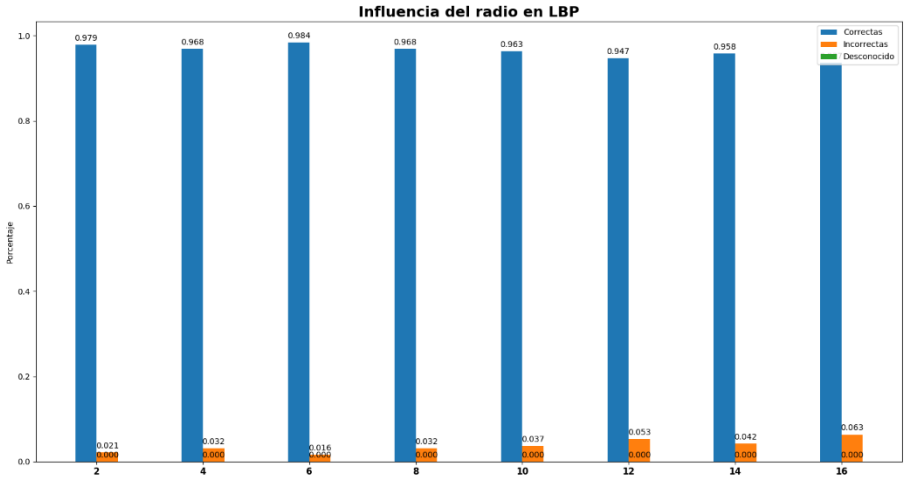


Figura 78: Influencia del radio escogido en LBP

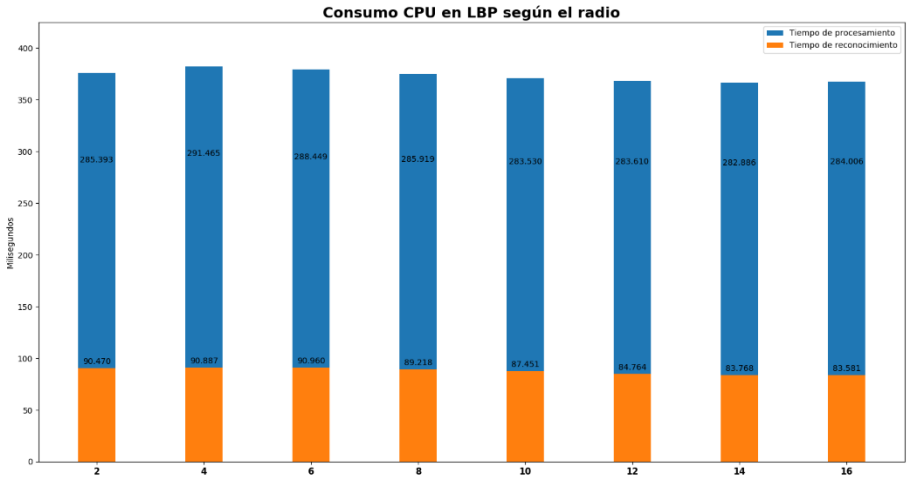


Figura 79: Consumo de CPU con LBP según el radio

5.5.- Rendimiento

En este punto se han determinado los algoritmos que mejores resultados han demostrado sobre las bases de datos de pruebas, por lo que el siguiente paso será ensamblar todos los componentes para implementar una aplicación de reconocimiento facial, y poder evaluar su rendimiento sobre la Raspberry Pi. En este apartado se incidirá especialmente en el tiempo de computación requerido y se propondrán posibles cambios que tengan sus consecuencias en una reducción del tiempo de CPU empleado.

5.5.1.- Metodología de las pruebas

Dado el carácter variable de la entrada de la cámara de vídeo, y con objeto de mantener la uniformidad de estos datos para las diversas pruebas, se utilizarán como origen de los datos tres vídeos ubicados en el directorio `rasprec/resources/videos`. Estos vídeos contienen 313, 421 y 520 *frames* respectivamente, en los que en todo momento se encuentra una cara de un mismo sujeto.

Las diversas pruebas que se realicen se implementarán en el programa `perfomance_results.py`, ubicado en la carpeta de programas.

Se utilizará un ***profiler*** para evaluar el impacto computacional de cada una de las fases del proceso de detección y reconocimiento. Un *profiler* es una herramienta que permite realizar un análisis dinámico que mide el tiempo de ejecución de un programa descomponiéndolo en cada una de las partes del mismo. Hay diferentes tipos de *profilers* para Python, que pueden recoger diferentes tipos de medidas, como gasto de CPU o consumo de memoria, pero, como el objetivo final es optimizar el tiempo de ejecución, se utilizará **cProfile**, un módulo de C incluido en Python con una sobrecarga razonable y que ofrece información bastante detallada.

El módulo cProfile se puede llamar en línea de comandos al invocar Python utilizando el parámetro `-m cProfile`. En la figura 80 se puede ver un ejemplo de la información proporcionada por este módulo.

```
         137237 function calls (134219 primitive calls) in 39.802 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    189/1    0.030    0.000   39.803   39.803 {built-in method builtins.exec}
        1    0.039    0.039   39.803   39.803 20_perfomance_results.py:1(<module>)
      314    0.193    0.001   19.475    0.062 Pipeline.py:85(__next__)
        1   10.290   10.290   10.291   10.291 Processor.py:15(__init__)
      313    0.095    0.000    8.440    0.027 DetectorViola.py:75(detect_multiscale)
      313    8.343    0.027    8.343    0.027 {method 'detectMultiScale' of 'cv2.CascadeClassifier' objects}
        5    6.572    1.314    6.572    1.314 {built-in method builtins.input}
      306    0.027    0.000    5.017    0.016 Processor.py:447(load)
      306    0.038    0.000    3.419    0.011 Processor.py:81(__detect_eyes)
      306    3.257    0.011    3.378    0.011 EyeDetector.py:32(detect)
      313    1.581    0.005    3.062    0.010 Preprocessor.py:129(prepare)
      330    0.007    0.000    2.526    0.008 {built-in method builtins.next}
```

Figura 80: Salida del módulo cProfile de Python

Aunque la información proporcionada es bastante completa y presentada de una forma clara, puede suponer un problema navegar entre los cientos de líneas que genera la llamada anterior. Por ello, hay diversas herramientas que analizan estos resultados y los muestran de forma gráfica, facilitando de esta manera su estudio e interpretación.

Las herramientas visuales que se han utilizado son **SnakeViz**, que muestra claramente el tiempo de ejecución empleado por cada una de las funciones del programa y que permite navegar en diferentes niveles de profundidad, y **QCacheGrind**, que muestra un mapa de ejecución del programa, así como diferentes medidas de recursos empleados por cada función del programa analizado.

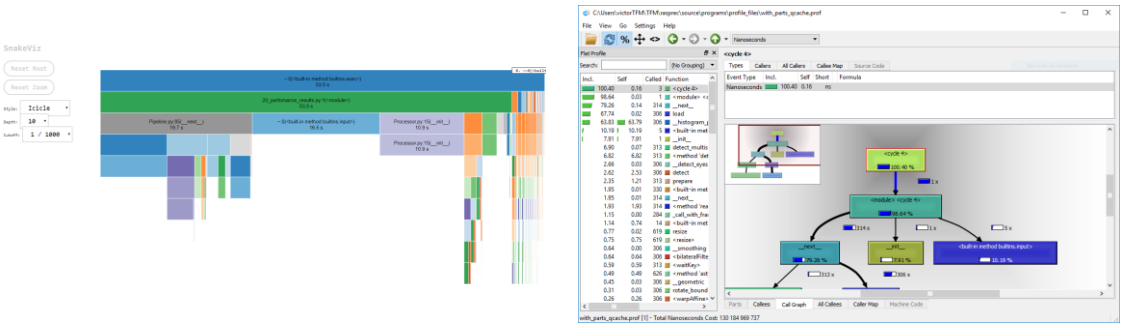


Figura 81: Captura de pantalla de SnakeViz y QProfiler

5.5.2.- Resultados

Al realizar el *profiling* sobre el proceso de reconocimiento de las caras de un vídeo y visualizarlo con SnakeViz se obtienen los resultados que se pueden ver en la figura 82:

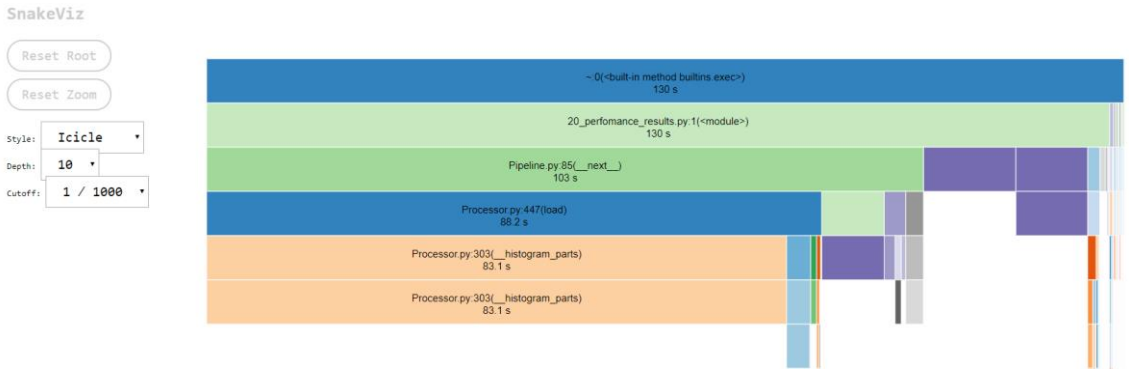


Figura 82: Resultados del profiling sobre la aplicación de reconocimiento

En esta captura se puede constatar lo que ya se vio en el apartado anterior: el tiempo requerido para el procesamiento de las caras previamente a su reconocimiento es excesivo. Es concretamente la función `__histogram_parts()` la que consume la mayor parte del tiempo, 83,1 segundos de los 130 segundos que ha llevado la ejecución de todo el programa (un 64% del tiempo total). Esta función es la que realiza el histograma por partes sobre la cara y es probable que su costo computacional se genera la imagen píxel a píxel fusionando los píxeles de tres los tres histogramas de la imagen original (el de la parte izquierda, el de la parte derecha y el de toda la imagen).

Se han analizado diversas soluciones para soslayar este problema minimizando el tiempo requerido sin afectar excesivamente a las ratios de efectividad del proceso, pero ninguna ha proporcionado unos resultados significativamente mejores que evitar la utilización del histograma por partes.

Si se ejecuta el programa sobre un fragmento de vídeo eliminando el paso de la generación del histograma por partes en la etapa de procesamiento se puede apreciar un reparto más equilibrado, tal y como se puede ver en la siguiente imagen:

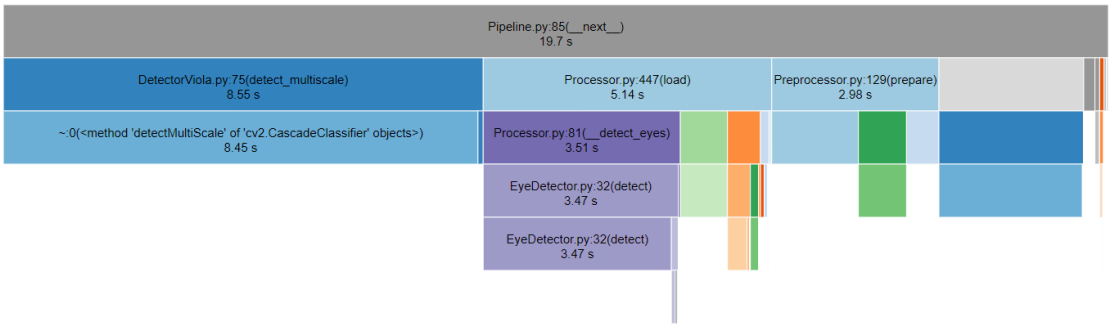


Figura 83: Profiling del procesamiento de las imágenes tras eliminar la fase de creación de histograma por partes

Aquí se aprecia un reparto más equilibrado del tiempo entre las distintas fases. Considerando que son necesarios 19,7 segundos para procesar el vídeo, se puede apreciar que:

- Casi la mitad del tiempo, 8,55 segundos, corresponde al proceso de detección de las caras. Como se vio anteriormente, este tiempo se podría reducir ampliando el factor de escala en las ventanas deslizantes, aunque a costa de una menor precisión.
- El procesamiento de las caras para el reconocimiento supone un 26% del tiempo, 5,14 segundos, siendo la función que más tiempo requiere la encargada de detectar los ojos en la imagen. Esto imposibilita cualquier tiempo de economía de proceso aquí, ya que esta función es imprescindible para el resto de las operaciones de procesamiento.
- La siguiente etapa en costo de tiempo es la de preprocesamiento, supone un 15% del tiempo total y comprende las funciones de redimensionado, conversión a escala de grises y generación de histograma.
- Un tiempo muy similar (13%) es el empleado por el proveedor de las imágenes de entrada.
- Por último, se podría destacar que el proceso de reconocimiento tiene un tiempo inapreciable frente al resto de operaciones, suponiendo 0,0833 segundos de los 19,7 segundos que supone todo el conjunto de etapas.

En la siguiente imagen se ve un mapa de la relación entre todas estas etapas, marcando las funciones que mayor impacto tienen sobre el tiempo de procesamiento. Conviene destacar que en este gráfico los porcentajes son relativos al tiempo de ejecución del programa, lo que incluye los tiempos de carga e inicialización de todas las clases, así como los tiempos de espera de la entrada del usuario.

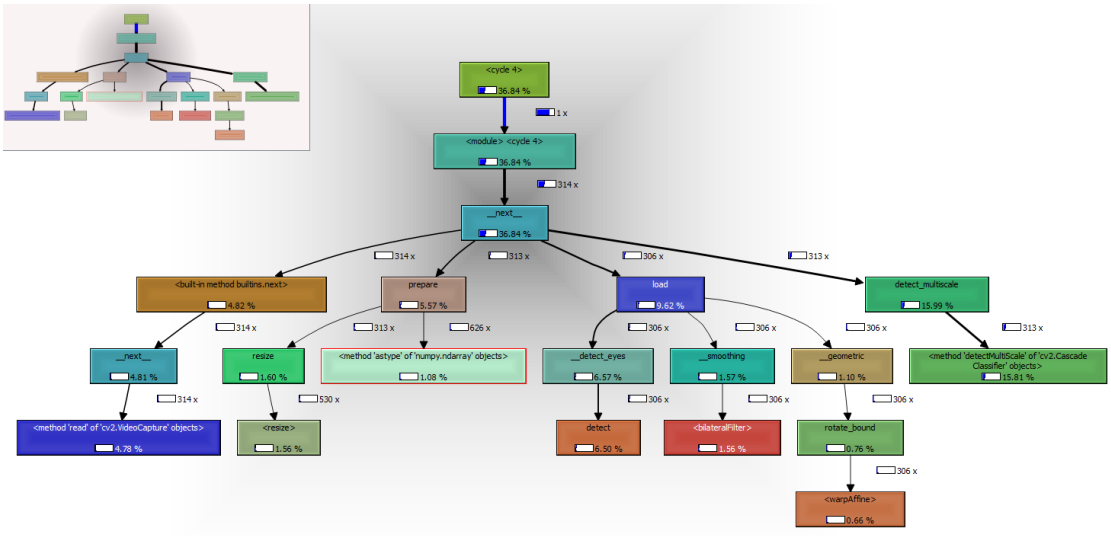


Figura 84: Árbol de funciones computacionalmente más intensivas del proceso de reconocimiento

5.6.- Análisis energético

Para el análisis del consumo energético se ha utilizado el dispositivo UM24C, que conectado entre la fuente de alimentación y la Raspberry Pi realiza mediciones en tiempo real del voltaje, la corriente y la potencia consumida. Como ya se explicó, la elección de este dispositivo se debe a su capacidad para conectarse por Bluetooth a un ordenador o teléfono móvil y generar un fichero Excel que refleje la evolución de estas tres medidas.

Las primeras medidas que se han tomado con objeto de establecer unos límites máximos y mínimos esperados en el consumo energético. Obviamente, el consumo mínimo se producirá cuando el dispositivo se encuentre en *idle*, es decir, cuando únicamente los procesos propios del sistema operativo se encuentren en ejecución. Por otro lado, es de esperar que el máximo consumo se produzca cuando todos los núcleos del procesador tengan un porcentaje de utilización del 100%.

Figura 85: Raspberry Pi 3 B con la cámara y el dispositivo UM24C

La gráfica mostrada a continuación muestra las lecturas obtenidas en estos dos escenarios:

- Los primeros 80 segundos no ha habido ninguna interacción del usuario con el sistema, por lo que el porcentaje de uso de CPU es próximo al 0%.
- Durante los siguientes 80 segundos se ha ejecutado la orden `sysbench –test=cpu –cpu-max-prime=20000 –num-threads=4 run`. Esta orden realiza un test de stress sobre el procesador que mantiene ocupados al 100% todos los núcleos del mismo durante todo el tiempo que dure el test.

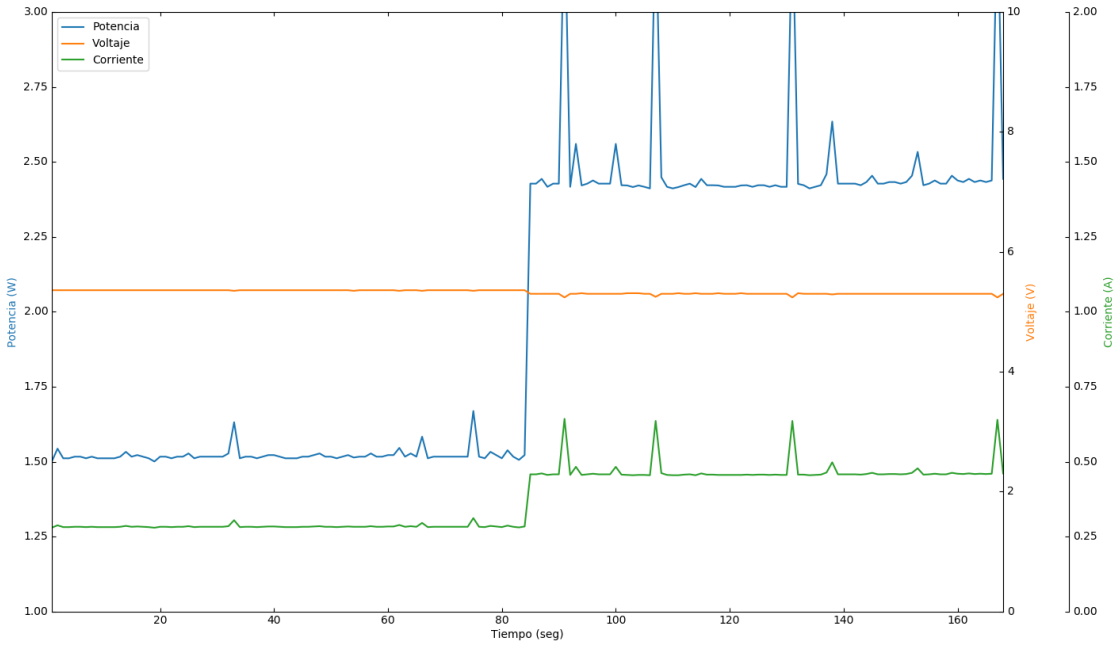


Figura 86: Consumo en idle y en test de stress

Manteniéndose el voltaje constante con una media de 5,329V, la corriente varía entre los 283,9 mA de media cuando el sistema se encuentra en idle y los 467,8 mA que se obtienen en las lecturas cuando todos los núcleos están al 100% de su capacidad. Esto implica unos consumos de 1,5216 W cuando el sistema no está realizando ninguna operación hasta un consumo de 2,4784 W que sería el máximo consumo de la Raspberry Pi.

La siguiente medición corresponde a los datos obtenidos al ejecutar el programa `power_measures_viola.py`. Este programa realiza el proceso de detección y reconocimiento sobre un vídeo utilizando los algoritmos de Viola Jones para la detección de las caras y de Fisherfaces para el reconocimiento. Además, utiliza la función `cpu_percent()` de la librería `psutil` de Python para calcular el uso medio de cada uno de los núcleos del procesador durante el proceso de reconocimiento.

En la gráfica se pueden apreciar dos partes claramente diferenciadas, separadas por un intervalo de espera de 3 segundos introducido para distinguirlas. La primera fase, que dura unos 80 segundos, corresponde a la carga y preparación de todo el sistema para funcionar. Esto involucra la carga de las librerías utilizadas (OpenCV, dlib, …), la inicialización de todas las clases utilizadas (por ejemplo, aquí parte del tiempo es consumido en la carga del fichero de entrenamiento de Viola Jones o el de los *facial landmarks*) y también el proceso de entrenamiento de Fisherfaces, que se ha optado por introducirlo en el propio programa en lugar de cargarlo desde un fichero generado anteriormente.

La segunda parte, a partir del segundo 80, es la que corresponde al proceso de reconocimiento de las caras en el vídeo. Las mediciones para esta parte indican una corriente media de 406,4 mA y un consumo de 2,1578 W.

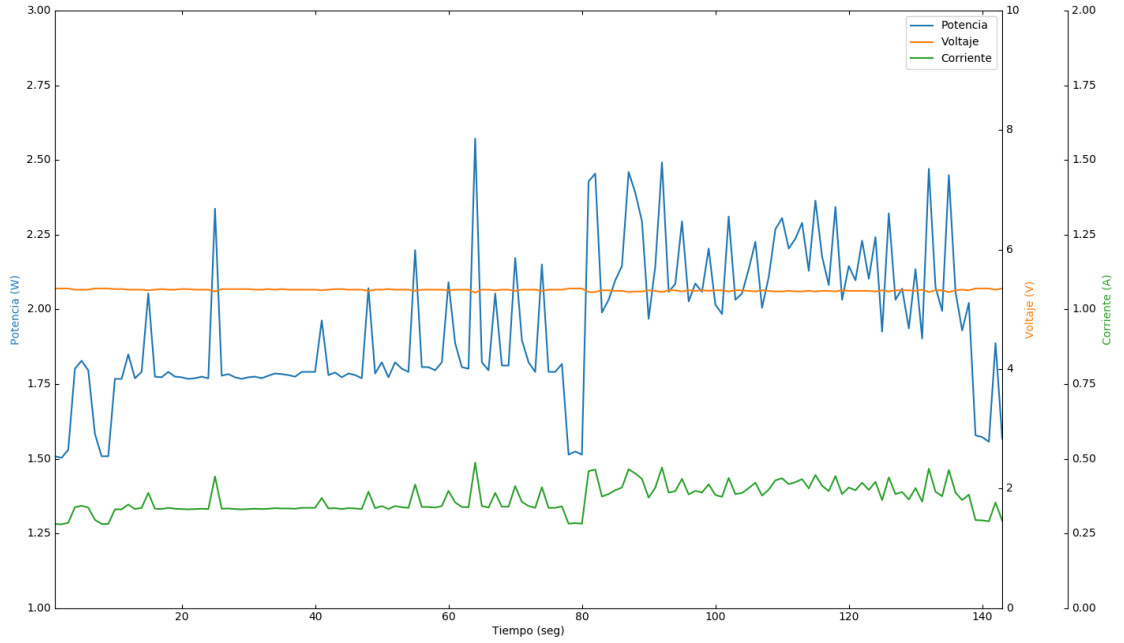


Figura 87: Consumo con Viola Jones y Fisherfaces

La otra medición se ha realizado con el programa `power_measures_hog_svm.py`, con una funcionalidad similar al anterior, pero utilizando el algoritmo HOG-SVM en lugar del de Viola Jones para la detección de las caras. Los resultados se pueden ver en la siguiente gráfica.

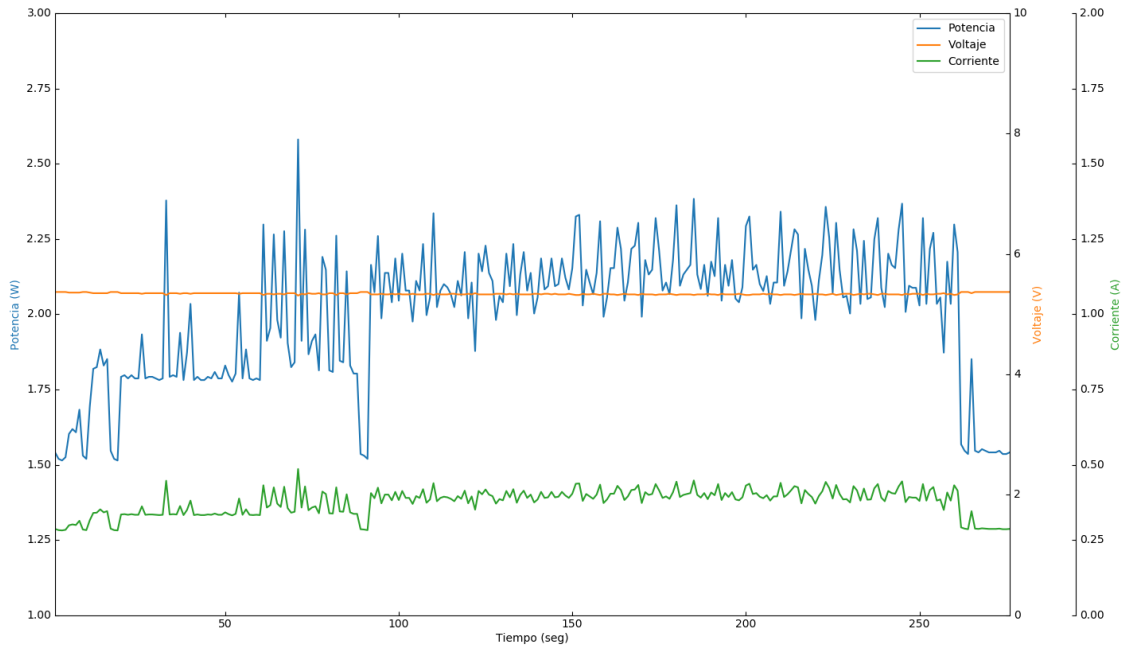


Figura 88: Consumo con HOG SVM y Fisherfaces

Nuevamente hay unos primeros 80 segundos dedicados a la preparación del entorno, así como el entrenamiento de los algoritmos. En este caso es necesario realizar el entrenamiento tanto para SVM como para Fisherfaces.

El resto del tiempo corresponde al reconocimiento del vídeo. En esta parte, los valores medios obtenidos en las diferentes medidas son 391,9 mA para el caso de la corriente y de 2,0902 W para la potencia consumida.

En cuanto a los porcentajes de uso del procesador se puede apreciar que son muy similares, teniendo valores medios próximos al 50% de uso. Sin embargo, aquí destaca que, mientras que en la implementación con HOG y SVM tiene un reparto equitativo entre todos los núcleos, en la implementación con Viola Jones hay un núcleo que tiene un uso bastante más alto que el resto de los núcleos.

```
Porcentajes de uso del procesador - Viola Jones
------------------------------------------------
Core 1: 36.700
Core 2: 75.600
Core 3: 44.300
Core 4: 36.500

Media: 48.275
```

```
Porcentajes de uso del procesador - HOG/SVM
-------------------------------------------
Core 1: 49.200
Core 2: 49.500
Core 3: 45.200
Core 4: 56.700

Media: 50.150
```

Figura 89: Porcentajes de uso del procesador

Esto probablemente sea debido a que el algoritmo de Viola Jones, implementado completamente en OpenCV, no esté optimizado para el uso en sistemas multiprocesador, mientras que en el otro algoritmo se ha utilizado la librería `threading` para repartir la tarea de análisis de las subimágenes obtenidas en el método de las ventanas deslizantes entre los diferentes procesadores.

Un resumen de todos los datos obtenidos en las mediciones de rendimiento se expone en la figura 90. Para ello, hay que tener en cuenta que el vídeo utilizado en las pruebas realizadas

contiene 520 frames que se redimensionan a 320x240 píxeles, lo que supone que se procesan un total de 39.936 megapíxeles (320x240x520).

	Tiempo	FPS	CPU	Consumo	Rendimiento	Rendimiento/Potencia
	(segs.)	(frames por seg.)		(vatios)	(megapíxel por segundo)	(megapíxel por segundo por vatio)
Viola Jones + Fisherfaces	58,2 s.	8,92 fps	48,275%	2,1578 W	0,6855 mpps	0,3176 mpps/W
HOG-SVM + Fisherfaces	187,8 s.	2,76 fps	50,150%	2,0902 W	0,2126 mpps	0,1017 mpps/W

Figura 90: Resumen de las mediciones de consumo

6.- Conclusiones

Uno de los objetivos planteados al principio de esta memoria era conseguir implementar un sistema de reconocimiento facial en tiempo real, es decir, que fuera capaz de procesar 24 imágenes por segundo. Los resultados obtenidos distan de este objetivo, limitándose a 14 imágenes por segundo cuando únicamente se realiza el proceso de detección de imágenes, y descendiendo hasta 7 imágenes por segundo si también hay un proceso de reconocimiento sobre las caras detectadas, lo que supone un rendimiento de 768,76 kilopíxeles por segundo.

Desde un punto de vista más amplio, se establecía como objetivo principal evaluar la viabilidad de implantar un sistema de detección y reconocimiento facial en un dispositivo de bajo coste y bajo consumo, tal como la plataforma Raspberry Pi.

De este objetivo principal, se desprenden varios sub-objetivos, que se exponen a continuación:

- Analizar las diferentes fases que comprende el proceso de detección y reconocimiento facial, tales como la generación del descriptor de la imagen o la posterior clasificación de dicho descriptor. En cada fase se han analizado los algoritmos que se puedan adaptar mejor a los objetivos perseguidos.
- Implementar una librería que desarrolle un marco de trabajo que encapsule todo el proceso de detección y reconocimiento facial. Esta librería tendrá un desarrollo modular que permita intercambiar fácilmente los algoritmos utilizados en cada fase del proceso.
- Evaluar la idoneidad de los diferentes algoritmos disponibles con objeto de seleccionar la combinación de los mismos que maximice el rendimiento y la confiabilidad de la detección y minimice el consumo.
- La evaluación de los algoritmos se ha de realizar en tres puntos clave del proceso:
    - Clasificación de las imágenes como caras o no caras.
    - Detección de las caras dentro de una imagen
    - Reconocimiento de las caras.

Todos los objetivos enumerados anteriormente sí que han sido alcanzados de forma satisfactoria, ya que en este trabajo se ha conseguido lo siguiente:

- Se ha descompuesto el proceso de detección y reconocimiento facial en diferentes etapas, estudiando las diferentes alternativas para cada una de estas etapas.
- Se ha implementado una librería que modele las conclusiones extraídas del punto anterior. Esta librería, cuyo detalle se expone en el apartado 3.2, implementa diferentes algoritmos para las diferentes fases del proceso de detección y reconocimiento.
- Se ha llevado a cabo un estudio de las diferentes combinaciones de descriptores de características y clasificadores para la clasificación de caras y no caras. Este estudio, cuyos resultados se exponen en el apartado 5.2, se ha basado en tres puntos:
    - Elección de los parámetros óptimos de cada algoritmo.

  - Ejecución del algoritmo sobre un conjunto de imágenes de entrenamiento ya conocidas con objeto de obtener los valores de precisión, exhaustividad y exactitud, según la métrica conocida como *precision and recall*.
  - Análisis del tiempo empleado por cada algoritmo en procesar cada imagen.

  Se ha determinado que la mejor combinación de algoritmos es HOG + SVM, con una precisión de 0,992 y un tiempo de 1,807 ms por imagen.
- Se ha estudiado el comportamiento de los clasificadores al integrarlos en el proceso más complejo que es localizar la posición de una cara dentro de una imagen, donde intervienen factores como la descomposición de la imagen en ventanas y la fusión de los resultados, así como la eliminación de los falsos positivos. Los resultados de este estudio se reflejan en el apartado 5.3 y nuevamente se centran en la métrica *precision and recall* y la medición del tiempo requerido.

  Se han obtenidos los mejores resultados para el algoritmo de Viola Jones, con una precisión de 0,838 y un tiempo de 73,929 ms en una imagen de 384x286 píxeles.
- El último estudio, que se desarrolla en el apartado 5.4, se ha centrado en la capacidad de los diferentes algoritmos de reconocimiento facial para identificar una cara. En este caso, los valores a medir han sido nuevamente el tiempo y el porcentaje de identificaciones correctas obtenidas en cada uno.

  Los mejores valores se han obtenido al usar Fisherfaces, con una tasa de aciertos de 0,984 y un tiempo de procesamiento de 294,565 ms.
- Posteriormente, se ha realizado un *profiling* para localizar los puntos críticos del proceso de reconocimiento que mayor tiempo de CPU requieren.
- Por último, en el apartado 5.6, se ha realizado un análisis del consumo energético del sistema con objeto de relacionar el rendimiento con el consumo, y que ha reflejado unos valores de 0,3176 mpps/W para la combinación de algoritmos más eficiente, que es Viola Jones para la detección y Fisherfaces para el reconocimiento.

A la vista de los resultados, una reflexión a realizar es la gran diferencia obtenida cuando se intentan extrapolar los resultados obtenidos con los bancos de prueba de entrenamientos a entornos reales con condiciones variables. En ese caso las tasas de éxito, tanto de reconocimiento como de detección, disminuyen notablemente. En mi opinión, este hecho enfatiza en la importancia del procesamiento de las imágenes antes de enviarlas a los algoritmos de detección y reconocimiento, para conseguir una mayor homogeneización de estas, eliminando la influencia de factores como la iluminación o la pose.

Por otro lado, otra conclusión importante extraída del desarrollo de este proyecto es la gran importancia de una correcta selección de los parámetros de los diversos algoritmos utilizados. Estos parámetros no solo influyen en las tasas de acierto obtenidas, sino que son determinantes para los tiempos empleados en procesar las imágenes. Además, no se puede hablar de unos valores óptimos, ya que su elección se ve influida en gran medida por las características de las imágenes procesadas. Durante las diversas pruebas realizadas se ha demostrado que, unos parámetros que devuelven unos resultados óptimos pasan a ser mediocres cuando se aplican a imágenes obtenidas con unas condiciones de iluminación diferentes o simplemente en otro entorno.

De esto se puede inferir que uno de los puntos críticos en la construcción de un sistema de reconocimiento es la acertada elección de estos parámetros, y que esta no puede realizarse de forma genérica, sino que tiene enfocarse hacia el tipo de imágenes concretas que se van a procesar.

7.- Bibliografía

Akash, D. (17 de Enero de 2019). *Understanding AdaBoost*. Obtenido de https://towardsdatascience.com/understanding-adaboost-2f94f22d5bfe

Aldama, Z. (27 de Abril de 2018). *Videovigilancia: China se queda con tu cara*. Obtenido de https://retina.elpais.com/retina/2018/04/25/tendencias/1524640135_207540.html

Bandara, R. (30 de Agosto de 2017). *Bag-of-Features Descriptor on SIFT Features with OpenCV (BoF-SIFT).* Obtenido de https://www.codeproject.com/Articles/619039/Bag-of-Features-Descriptor-on-SIFT-Features-with-O

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). *SURF: Speeded Up Robust Features.* ETH Zurich.

Ben Fraj, M. (5 de Enero de 2018). *In Depth: Parameter tuning for SVC*. Obtenido de https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769

Blesoe, W. (1963). A Study to Determine the Feasibility of a Simplified Face Recognition Machine. Palo Alto, California.

Brownlee, J. (27 de Marzo de 2019). *How to Evaluate Pixel Scaling Methods for Image Classification With CNNs.* Obtenido de https://machinelearningmastery.com/how-to-evaluate-pixel-scaling-methods-for-image-classification/

Dalal, N., & Triggs, B. (2010). Histograms of Oriented Gradients for Human Detection. *International Conference on Computer Vision.* San Diego, Estados Unidos.

Davida, B. (3 de Julio de 2018). *Bag of Visual Words in a Nutshell.* Obtenido de https://towardsdatascience.com/bag-of-visual-words-in-a-nutshell-9ceea97ce0fb

Davida, B. (3 de Julio de 2018). *Towards Data Science*. Obtenido de https://towardsdatascience.com/

Davis West, J. (1 de Agosto de 2017). *A Brief History of Face Recognition*. Obtenido de https://www.facefirst.com/blog/brief-history-of-face-recognition-software/

Dawson-Howe, K. (2014). *A Practical Introduction to Computer Vision with OpenCV.* John Wiley & Sons.

Freund, Y., & Schapire, R. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference*.

Geitgey, A. (5 de Mayo de 2014). *Machine Learning is Fun*. Obtenido de https://medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471

Ha The Hien, D. (28 de Abril de 2017). *The Modern History of Object Recognition — Infographic*. Obtenido de https://medium.com/@nikasa1889/the-modern-history-of-object-recognition-infographic-aea18517c318

Howse, J., Joshi, P., & Beyeler, M. (2016). *OpenCV: Computer Vision Projects with Python.* Packt Publishing.

Joshi, P., Millán Escrivá, D., & Godoy, V. (2016). *OpenCV by Example.* Packt Publishing.

Jung, H. (9 de Abril de 2018). *Adaboost for Dummies: Breaking Down the Math (and its Equations) into Simple Terms.* Obtenido de https://towardsdatascience.com/adaboost-for-dummies-breaking-down-the-math-and-its-equations-into-simple-terms-87f439757dcf

Kaehler, A., & Bradski, G. (2008). *Learn OpenCV. Computer Vision in C++ with the OpenCV Library.* O'Reilly.

Kazemi, V., & Sullivan, J. (2014). One Millisecond Face Alignment with an Ensemble of RegressionTrees. Stockhol: Sweden.

Li, S., & Jain, A. (2011). *Handbook of Face Recognition (Second Edition).* Springer.

London, I. (6 de Mayo de 2016). *Image Classification in Python with SIFT Features.* Obtenido de https://ianlondon.github.io/blog/how-to-sift-opencv/

Lopez Briega, R. (10 de Junio de 2017). *Boosting en Machine Learning con Python*. Obtenido de https://relopezbriega.github.io/blog/2017/06/10/boosting-en-machine-learning-con-python/

Lowe, D. (January de 2004). Dinstictive Image Features from Scale-Invariant Keypoints. Vancouver, Canada: University of British Columbia.

Malisiewincz, T. (20 de Enero de 2015). *From feature descriptors to deep learning: 20 years of computer vision.* Obtenido de http://www.computervisionblog.com/2015/01/from-feature-descriptors-to-deep.html

Mallick, S. (30 de Enero de 2017). *Handwritten Digits Classification : An OpenCV (C++ / Python) Tutorial.* Obtenido de https://www.learnopencv.com/handwritten-digits-classification-an-opencv-c-python-tutorial/

McConnell, R. (1986). *Method of and Apparatus for Pattern Recognition.* U.S. Patent No. 4567610.

Mithi. (28 de Marzo de 2017). *Vehicle Detection with HOG and Linear SVM.* Obtenido de https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a

Ning, M. (9 de Abril de 2019). *SIFT(Scale-invariant feature transform).* Obtenido de https://towardsdatascience.com/sift-scale-invariant-feature-transform-c7233dc60f37

NIST. (1993). *Face Recognition Technology (FERET)*. Obtenido de https://www.nist.gov/programs-projects/face-recognition-technology-feret

Pajares Martisanz, G., & de la Cruz García, J. (2007). *Visión por computador. Imágenes digitales y aplicaciones.* Ra-MA.

Papageorgiou, C., Oren, M., & Poggio, T. (1998). *A General Framework for Object Detection.* Cambridge, MA: MIT.

Recuero de los Santos, P. (23 de Enero de 2018). *Machine Learning a tu alcance: la matriz de confusión*. Obtenido de https://empresas.blogthinkbig.com/ML-a-tu-alcance-matriz-confusion/

Rosebrock, A. (9 de Noviembre de 2015). *Pedestrian Detection OpenCV.* Obtenido de https://www.pyimagesearch.com/2015/11/09/pedestrian-detection-opencv/

Rosebrock, A. (3 de Abril de 2017). *Facial landmarks with dlib, OpenCV, and Python*. Obtenido de https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/

Sakai, T., Nagao, M., & Kanade, T. (1971). *Computer analysis and classification of photographs of human faces.* Kyoto, Japón: Kyoto University.

Saxena, S. (11 de Mayo de 2018). *Precision vs Recall*. Obtenido de https://towardsdatascience.com/precision-vs-recall-386cf9f89488

Sinhal, K. (23 de Enero de 2017). *Training a better Haar and LBP cascade based Eye Detector using OpenCV*. Obtenido de https://www.learnopencv.com/training-better-haar-lbp-cascade-eye-detector-opencv/

Sirovich, L., & Kirby, M. (1986). Low Dimensional Procedure for the Caracterization of Human Faces. Providence, Rhode Island.

Viola, P., & Jones, M. (2001). Rapid Object Detection using a Boosted Cascade of Simple Features. *Computer Vision and Pattern Recognition. Proceedings of the 2001 IEEE Computer Society Conference on, 1*, I-511.

Vyas, K. (13 de Julio de 2016). *Bag of Visual Words Model for Image Classification and Recognition.* Obtenido de https://kushalvyas.github.io/BOV.html

Yang, M.-H., Kriegman, D., & Ahuja, N. (Enero de 2002). Detecting Faces in Images: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 24*(1).

Zafeiriou, S., Zhang, C., & Zhang, Z. (2015). A survey on face detection in the wild: past, present and future. *Computer vision and image understanding, 138*, 1-24.

8.- Glosario de siglas y términos

AdaBoost	Adaptative Boosting
AHE	Adaptative Histogram Equalization
API	Application Programming Interface
ARM	Advanced RISC Machines
BoVW	Bag of Visual Words
BoW	Bag of Words
CBCL	Center of Biological & Computacional Learning
CLAHE	Constrast Limit Adaptative Histogram Equalization
CNN	Convolutional Neural Network
CSI	Camera Serial Interface
DARPA	Defense Advanced Research Projects
DSI	Display Serial Interface
FERET	Face Recognition Technology
GPIO	General Purpose Input/Output
HDMI	High-Definition Multimedia Interface
HOG	Histogram of Gradients
HSV	Hue Saturation Value
IoT	Internet of Things
kNN	k-Nearest Neighbours
LBP	Local Binary Patterns
LDA	Linear Discriminant Analysis
MIT	Massachusetts Institute of Technology
NIST	National Institute of Standards and Technology
NMS	Non-Maximum Suppression
ORB	Oriented FAST and Rotated BRIEF
PCA	Principal Component Analysis
PIN	Personal Identification Number
RBF	Radial Basis Function
RGB	Red Green Blue

SIFT	Scale Invariant Feature Transform
SOC	System on Chip
SURF	Speeded Up Robust Features
SVM	Support Vector Machines

Anexo I

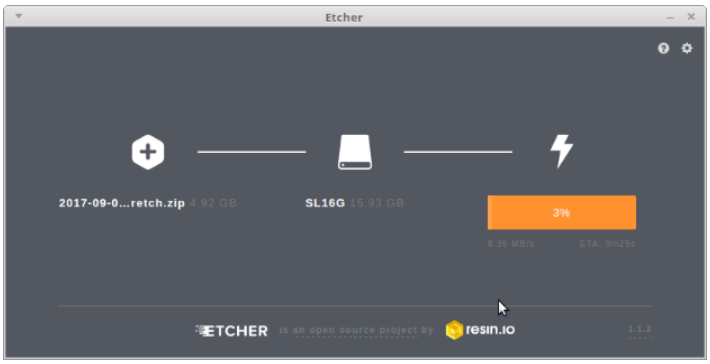
I.1.- Instalación del sistema

El primer paso es instalar el sistema operativo. Hay múltiples sistemas operativos disponibles para esta plataforma, pero la distribución oficial mantenida por la Fundación Raspberry Pi es **Raspbian**, estando disponibles imágenes de esta ya preparadas para copiarlas directamente en la tarjeta Micro SD.

Aunque este paso es trivial, es conveniente realizar algunas operaciones sobre la imagen instalada para optimizarla para el uso que se le va a dar. A continuación, se indican los pasos a realizar.

1.- Descargar la imagen de Raspbian de los repositorios oficiales. En el proyecto se ha utilizado la versión Raspbian Jessie[8].

2.- Se graba la imagen descargada en la tarjeta MicroSD utilizando la aplicación **Etched** para Linux. La tarjeta MicroSD debe tener una capacidad mínima de 16 GB, ya que de lo contrario no se podrá compilar OpenCV.



3.- Como en toda instalación nueva de una distribución Linux el primer paso a realizar es la actualización de los paquetes mediante los repositorios.

4.-Si se instalara de otra manera será necesario expandir el sistema de ficheros ya que puede ocurrir que la imagen grabada no ocupe todo el espacio disponible en la tarjeta de memoria. Con Etched no es necesario, pero se puede verificar utilizando el comando:

```
$ df -h
```

Si fuera necesario expandir el sistema de ficheros se debe hacer mediante el comando `raspi-config`.

5.- Por defecto el usuario es **pi** y la contraseña **raspberry**. Para mayor seguridad cambiamos la contraseña.

```
$ passwd
```

[8] http://downloads.raspberrypi.org/raspbian/images/raspbian-2016-05-31/

6.- Raspbian tiene algunos paquetes instalados que ocupan bastante espacio en la tarjeta de memoria. Es posible liberar casi 1GB eliminando el Wolfram Engine y algunos otros paquetes no esenciales.

```
$ sudo aptitude purge wolfram-engine
$ sudo aptitude purge minecraft-pi
$ sudo aptitude purge sonic-pi
$ sudo apt-get purge libreoffice*
$ sudo apt-get clean
$ sudo apt-get autoremove
```

7.- La librería OpenCV tiene muchas dependencias, por lo que debemos instalar los correspondientes paquetes.

```
$ sudo aptitude install build-essential cmake pkg-config
$ sudo aptitude install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
$ sudo aptitude install libavcodec-dev libavformat-dev libswscale-dev
  libv4l-dev libavdevice-dev
$ sudo aptitude install libxvidcore-dev libx264-dev ffmpeg
$ sudo aptitude install libgtk2.0-dev
$ sudo aptitude install libatlas-base-dev gfortran
```

8.- Si Python no estuviera instalado habría que instalarlo. Todo el código fuente de este proyecto está en Python 3.6.

```
$ sudo aptitude install python3-dev
```

9.- Es muy probable que el teclado español no se configure correctamente. Para solucionar este problema hay que añadir la siguiente línea en el fichero `/home/pi/.profile`

```
# Activamos teclado español
setxkbmap es
```

10.- Si se va a trabajar de forma remota con la Raspberry Pi será necesario instalar `ssh` y el paquete `xrdp` que permitirá el acceso mediante escritorio remoto. Para habilitar dicho acceso será necesario, una vez instalados estos paquetes, hacerlo mediante el comando `raspi-config`.

```
$ sudo aptitude install ssh xrdp
```

11.- Se puede comprobar si están correctamente instalados comprobando que los puertos 22 y 3389 están abiertos.

```
$ sudo nmap localhost
```

12.- Ya se puede acceder desde un equipo Windows con *Conexión a escritorio remoto* o desde Linux con *Vinagre*. En este último caso se puede escoger el protocolo **ssh** para acceder en línea de comandos o el protocolo **rdp** para acceder al entorno gráfico.

I.2.- Instalación de OpenCV

Una vez preparado el sistema es el momento de instalar OpenCV. Será necesario descargar el código fuente y compilarlo ya que no hay ningún otro medio disponible para disponer de la librería OpenCV en la Raspberry Pi.

1.- Descargamos el código fuente de OpenCV:

```
$ cd
$ wget -O opencv.zip https://github.com/Itseez/opencv/archive/3.6.4.zip
$ unzip opencv.zip
```

2.- También hay que descargar el código fuente de *contrib*, ya que incluye algunas características no disponibles en la versión estándar por motivos de derechos de autor como SIFT o SURF:

```
$ wget -O opencv_contrib-zip https://github.com/Itseez/opencv_contrib/ \
  archive/3.6.4.zip
$ unzip opencv_contrib.zip
```

3.- Para la gestión de paquetes de Python instalamos `pip`, para ello hay que descargarlo directamente del repositorio y ejecutarlo.

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python get-pip.py
```

4.- Una característica muy interesante de Python es la posibilidad de utilizar **entornos virtuales** que nos permitirán en cierto modo compartimentalizar las librerías instaladas en Python. Esto significará que cuando se instale una librería únicamente pertenecerá a un entorno virtual, por lo que por ejemplo se podrían tener diferentes versiones de la misma librería simultáneamente en diferentes entornos virtuales.

```
$ sudo pip install virtualenv virtualenvwrapper
$ sudo rm -rf ~/.cache/pip
```

5.- Hay que preparar el entorno virtual, el primer paso es editar el fichero `~/.profile` añadiendo las siguientes líneas:

```
# virtualenv y virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

6.- Para que se apliquen los cambios hay que reiniciar o cargar el fichero anterior:

```
$ source ~/.profile
```

7.- Creamos el entorno virtual, en este caso para Python 3.6

```
$ mkvirtualenv cv -p python3
```

8.- Una vez preparado el entorno virtual accedemos a él utilizando el comando `workon` e instalamos más librerías utilizadas en el proyecto: `numpy` e `imutils`.

```
$ workon cv
(cv) $ pip install numpy
(cv) $ pip install imutils
```

9.- Ya se puede compilar OpenCV. Hay que asegurarse de incluir la opción WITH_OPENMP=ON para que habilite el uso del procesamiento paralelo en las funciones de OpenCV y así aprovechar los 4 núcleos del procesador de la Raspberry Pi.

```
(cv) $ cd ~/opencv-3.2.0
(cv) $ mkdir build
(cv) $ cd build
(cv) $ cmake -D MAKE_BUILD_TYPE=RELEASE \
        -D CMAKE_INSTALL_PREFIX=/usr/local \
        -D INSTALL_PYTHON_EXAMPLES=ON \
        -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-3.2.0/modules \
        -D BUILD_EXAMPLES=ON
        -D WITH_OPENMP=ON
        -D WITH_FFMPEG=HISOTON
        -D WITH_V4L=ON
        ..
```

Conviene verificar la salida para asegurarnos de que no haya ningún error.

10.- Es el momento de compilar.

```
(cv) $ make -j4
```

El modificador -j4 indica el número de núcleos que se utilizarán para la compilación. Acelera bastante el proceso de compilación, pero en alguna ocasión ha dado errores. Si así fuera hay que limpiar y compilar utilizando un único núcleo.

```
(cv) $ make clean
(cv) $ make
```

11.- Una vez compilado hay que instalarlo

```
(cv) $ sudo make install
(cv) $ sudo ldconfig
```

12.- Al utilizar entornos virtuales hay que crear los enlaces necesarios.

```
(cv) $ cd $HOME/.virtualenvs/cv/lib/python2.7/site-packages/
(cv) $ ln -s /usr/local/lib/python2.7/site-packages/cv2.so cv2.so
```

I.3.- Comprobación de la cámara

1.- Hay que habilitar la cámara a través de `raspi-config` en la opción Interfacing Options -> P1 Camera. A continuación, hay que reiniciar la Raspberry.

```
$ sudo raspi-config
```

2.- Se puede verificar el funcionamiento de la cámara mediante el comando `raspistill`.

```
$ raspistill -o imagen.jpg
```

3.- Ahora instalamos los módulos Python para acceder a la cámara. En lugar de instalar el módulo `picamera` instalamos el sub-módulo opcional `picamera[array]`. Esto permitirá utilizar OpenCV con la cámara ya que este módulo dispone de la posibilidad de obtener las imágenes de la cámara como arrays NumPy.

```
(cv) $ pip install “picamera[array]”
```

I.4.- Instalación de Dlib

Otra librería importante en el desarrollo del proyecto es la librería Dlib. Al igual que OpenCV también es necesario compilarla a partir del código fuente, pero supone un problema mayor ya que la Raspberry Pi únicamente dispone de 1 GB de memoria RAM y es insuficiente para el proceso de compilación, que se interrumpirá con un mensaje de error.

Por tanto, antes de compilar será necesario incrementar todo lo posible la memoria RAM disponible realizando los siguientes pasos:

1.- En primer lugar, debemos incrementar el tamaño de swap. En la Raspberry Pi la gestión de la swap se realiza a través del fichero `/etc/dphys-swapfile` por lo que hay que editarlo y modificar el valor CONF_SWAPSIZE (por defecto 100) para que valga 1024MB.

```
# CONF_SWAPSIZE=100
CONF_SWAPSIZE=1024
```

2.- Reiniciamos el servicio de swap.

```
$ sudo /etc/init.d/dphys-swapfile stop
$ sudo /etc/init.d/dphys-swapfile start
```

3.- Confirmamos que el tamaño de la swap ha sido modificado.

```
$ free -m
```

4.- Para conseguir más memoria libre arrancamos sin PIXEL GUI, es decir, sin el entorno gráfico. Esto lo conseguimos con la orden `raspi-config`. Ahí seleccionamos Boot options -> Desktop / CLI -> Console Autologin.

5.- Antes de salir de `raspi-config` reducimos el tamaño de la memoria asignada a la GPU para disponer de más memoria para la compilación. Esto lo hacemos en Advanced options -> Memory Split e indicamos un valor de **16**.

6.- Ahora ya se puede reiniciar el sistema.

7.- Una vez reiniciado podemos proceder a instalar las dependencias de **dlib**.

```
$ sudo aptitude update
$ sudo aptitude install build-essential cmake libgtk-3-dev libboost-all-dev
```

8.- No nos olvidamos de acceder al entorno virtual.

```
$ workon cv
```

9.- Es necesario instalar **numpy**, **scipy** y **scikit-image** lo cual hacemos a través de **pip**.

```
(cv)$ pip install numpy
(cv)$ pip install scipy
(cv)$ pip install scikit-image
```

10.- Y ya podemos instalar **dlib**.

```
(cv)$ pip install dlib
```

11.- Una vez instalado hay que volver a asignar 100MB a la memoria swap mediante el fichero `/etc/dphys-swapfile`.

```
CONF_SWAPSIZE=100
```

12.- Y en **raspi-config** volvemos a asignar 64MB a la GPU en Advanced options -> Memory Split y le indicamos que se debe arrancar nuevamente en el entorno gráfico en Boot options -> Desktop / CLI -> Desktop Autologin.

Anexo II. Bases de datos de imágenes

II.1.- Bases de datos

Para realizar las pruebas en un entorno controlado y poder comparar los resultados de las diferentes ejecuciones se ha recurrido a cuatro bases de datos de caras ampliamente utilizadas:

- **Base de datos de caras de Yale**: esta base de datos contiene 165 imágenes en escala de grises de 320x243 píxeles pertenecientes a 15 individuos. Hay 11 imágenes de cada individuo con diferentes expresiones faciales o configuración: luz centrada, con gafas, feliz, iluminado por la izquierda, sin gafas, iluminado por la derecha, triste, somnoliento, sorprendido y con los ojos cerrados.
  Esta base de datos se puede considerar como un entorno más controlado ya que las caras se encuentran todas centradas en la imagen y además se muestran sobre un fondo blanco (salvo la sombra de la propia cabeza), por lo que se puede considerar que, al tener un fondo más homogéneo, se minimiza la probabilidad de falsos negativos.
- **Base de datos de caras de BioID:** esta base de datos consta de 1521 imágenes en escala de grises con un tamaño de 384x286 píxeles perteneciendo a 23 personas diferentes.

  Al contrario que la base de datos de Yale, esta base de datos incide especialmente en intentar reproducir situaciones del *mundo real*. Por ello las imágenes muestran una gran variedad de iluminaciones, fondos o tamaños de la cara frente a las imágenes de Yale que se muestran más homogéneas.


Muestra de imagen de BioID


Muestra de imagen de Yale


Muestra de imagen de MIT CBCL


Muestra de imagen de HELEN

- **Base de datos de caras del MIT CBCL**: el CBCL (Center of Biological & Computacional Learning) del MIT fue fundado en el año 1992 teniendo como premisa que el entrenamiento es una parte fundamental e imprescindible para el problema de la inteligencia, tanto biológica como artificial. Como parte de los recursos disponibles se encuentra una base de datos que dispone de 2429 imágenes de caras y 4548 imágenes que no son caras, todas en formato PGM. Esto me ha resultado muy útil ya que a pesar de que hay muchas bases de datos de caras, hay muy pocas que disponen de imágenes que no son caras, algo que es imprescindible para el proceso de entrenamiento de los clasificadores.

En cuanto a las características de las imágenes, tienen un tamaño de 19x19 píxeles en escala de grises. Al contrario que las bases de datos anteriores, las imágenes de caras se centran únicamente en la parte distintiva de la cara sin elementos periféricos como el pelo, barbilla, …. En realidad, se puede decir que se encuentran en un estado muy similar al de las imágenes de caras del presente proyecto tras el paso de procesado de las mismas previo a la etapa de reconocimiento.

Cabe destacar que la base de datos ya no se encuentra disponible en la página oficial del CBCL. Sin embargo, aún se encuentra disponible en otras ubicaciones en la red[9].

- **Base de datos de caras HELEN**: esta base de datos ha sido recopilada por los autores del artículo *Interactive Facial Feature Localization* centrado en la detección de características faciales. Para ello los autores necesitaban una gran cantidad de imágenes que cubrieran un amplio rango variaciones faciales, incluyendo posición, luz, expresión, oclusión y diferencias individuales. Dada la inexistencia de un base de datos que se adaptara a sus requisitos optaron por crearla extrayendo las imágenes de Flicker. Esto hace que esta base de datos tenga unas características muy particulares que la diferencian de las nombradas anteriormente: las imágenes son todas en color y con gran resolución, normalmente superiores a 2000x2000 píxeles. Se muestran en gran variedad de entornos y no incluyen únicamente la cara sino también el entorno de esta.

  Obviamente estas imágenes no son adecuadas para el proceso de aprendizaje de los algoritmos, pero se utilizarán para evaluar el rendimiento de los algoritmos de detección facial en entornos más parecidos a la realidad.

II.2.- Preparación de los datasets

Las bases de datos de caras utilizadas en el proyecto son útiles para la prueba de los diferentes algoritmos de detección y reconocimiento. Sin embargo, han resultado poco aptas para el entrenamiento de los algoritmos de detección supervisada. Esto se debe principalmente a que por norma general la cara no se encuentra ubicada exactamente en la misma posición en cada una de las imágenes y a que incluyen una gran parte de fondo que realmente no va a aportar información al clasificador al no tener nada que ver con lo que realmente es la cara.

Para solucionar este problema, se ha optado por crear una selección de imágenes de caras relativamente homogénea que sirva para alimentar todos los clasificadores. El punto de partida de esta base de datos será una de las bases de datos indicadas en el apartado 6.1. En concreto se ha utilizado la base de datos de caras de BioID, que contiene un total de 1521 imágenes de caras correspondientes a 23 sujetos.

El proceso seguido para adaptar las imágenes a otras aptas para alimentar los clasificadores es:

[9] https://github.com/galeone/face-miner

- Se utilizará el algoritmo de Viola-Jones, incluido en la librería OpenCV y ya entrenado, para detectar el área de la cara en la imagen original.
- Una vez detectada la cara, es necesario recortarla y redimensionarla para que sean homogéneas, utilizando para ello los ojos como punto de referencia. Teniendo la cara, la posición de los ojos es fácilmente obtenible utilizando el algoritmo *facial landmarks* integrado en la librería Dlib.
- Una vez conocidas las coordenadas de los ojos se recorta la imagen para que los ojos estén a un 25% de distancia de la parte superior de la imagen y a un 25% de los bordes laterales de la misma.
- Por último, se redimensionan las caras para que tengan un tamaño homogéneo de 70x70 píxeles.

Anexo III: Diagrama de clases