_ TELECOM ESCUELA
TÉCNICA VLC SUPERIOR
DE UPV INGENIEROS
DE TELECOMUNICACIÓN

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

# Detección de texto en escenas naturales

**Rafael López González**

**Tutor: Antonio Albiol Colomer**

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2015-16

Valencia, 29 de junio de 2016

## Resumen

En el presente trabajo se pretende analizar los actuales algoritmos de estado del arte en detección y reconocimiento autónoma de texto en imágenes de escenas naturales, así como agilizar su ejecución para que puedan ser procesados por plataformas móviles en tiempo real, debido a que dichos algoritmos requieren una potencia de computación elevada para poder ejecutarse de forma rápida. Para alcanzar el objetivo propuesto se han desarrollado algoritmos de bajo coste computacional cuya función se basa en la selección de zonas de la imagen donde la probabilidad de encontrar texto es alta, debido a ciertas características que se detallaran a largo del presente documento. La finalidad de este trabajo es ayudar a las personas invidentes a interactuar más fácilmente con su entorno, proporcionándoles una herramienta que les permita ser capaces de obtener más información y que pueden llevar a cualquier lugar cómodamente gracias al poder de procesamiento de los nuevos teléfonos inteligentes.

## Resum

En el present treball es pretén analitzar els actuals algoritmes d'estat de l'art en detecció y reconeixement autònoma de text en imatges d'escenes naturals, així com agilitzar la seua execució per a que puguen ser processats per plataformes móviles a temps real, degut a que aquests algoritmes necessiten una potencia de computació elevada per a poder executar-se de forma rápida. Per a assolir el objectiu proposat s'han desenvolupat algoritmes de baix cost computacional, que tenen la funció de seleccionar zones de la imatge on la probabilitat de trobar text es alta, degut a diverses característiques que es desenvoluparan al llarg d'aquest document. La finalitat del present treball es ajudar a les persones invidents a interactuar més fàcilment amb el seu entorn, proporcionant-los una ferramenta amb la qual siguen capaços d'obtindre més informació i que poden portar a qualsevol lloc còmodament gràcies al poder de processament dels nous telèfons intel·ligents.

## Abstract

Along this thesis the actual state of the art algorithms of autonomous text localization and recognition in natural scene images are going to be analyzed, as well as methods for reaching a real-time execution on these matters are going to be developed, in order to create suitable algorithms for performing those tasks on mobile platforms. The developed algorithms for making the text localization faster are based on the detection of regions where is very likely to find text due to properties which are going to be detailed along this document. The main objective of this thesis is helping blind people to interact easily with their environment by given them a portable tool for obtaining more information of their surroundings.

# Índice

Capítulo 1. Introducción ..... 3
1.1 Motivación ..... 3
1.2 Objetivos ..... 3
Capítulo 2. Estado del arte ..... 4
2.1 Algoritmos de localización y reconocimiento de texto ..... 4
2.1.1 Método basado en regiones MSER (Maximally Stable Extremal Regions) ..... 5
2.1.2 Método basado en regiones ER (Extremal Region) ..... 7
2.2 Plataformas ..... 8
2.2.1 Ubuntu ..... 8
2.2.2 Android ..... 11
Capítulo 3. Análisis del problema ..... 16
3.1 Elección del método de detección y localización adecuado ..... 16
3.2 Software utilizado ..... 17
3.2.1 Eclipse ..... 17
3.2.2 OpenCV ..... 18
3.2.3 Android Studio ..... 19
3.3 Planificación inicial y cambios de la misma ..... 20
3.3.1 División por tareas ..... 20
3.3.2 Distribución temporal inicial ..... 21
3.3.3 Distribución temporal final ..... 22
3.4 Presupuesto ..... 23
Capítulo 4. Desarrollo y resultados del trabajo ..... 26
4.1 Desarrollo en C++ ..... 26
4.1.1 Algoritmo de predetección ..... 26
4.1.2 Algoritmo de detección y reconocimiento de texto ..... 45
4.2 Tiempo de ejecución del algoritmo en C++ ..... 51
4.3 Desarrollo en Android ..... 53
4.3.1 Compilación Nativa con Android NDK (Native Development Kit) ..... 53
Capítulo 5. Conclusiones ..... 56
5.1 Conceptos aprendidos ..... 56
5.2 Dificultades encontradas ..... 56
5.2.1 Durante el testeo del algoritmo en C++ ..... 56
5.2.2 Durante la implementación del algoritmo en Android ..... 61
5.3 Trabajo futuro ..... 61
Capítulo 6. Bibliografía ..... 62

# Capítulo 1. Introducción

## 1.1 Motivación

Una de las principales motivaciones que me ha llevado a elegir este trabajo es que se requerían conocimientos en Tratamiento de imágenes, la disciplina que más me ha llamado la atención a lo largo de mis estudios del grado en Ingeniería de tecnologías y servicios de telecomunicación. Dicha disciplina me parece interesante debido a que sus posibilidades de aplicación combinadas con el poder computacional del que disponen tanto los ordenadores como los dispositivos móviles de última generación pueden generar grandes beneficios a la sociedad.

Un claro ejemplo de dichos beneficios es la detección y reconocimiento de texto en escenas naturales, debido a que este campo aun desarrollo puede ayudar entre otras cosas a que las personas invidentes sean capaces de interactuar con mucha más facilidad con su entorno, convirtiéndolos en personas más independientes. Una situación típica en la que este campo puede ayudar a personas invidentes es orientarse dentro de un centro comercial, lugar donde podrían saber el nombre de la tienda que tienen enfrente fácilmente con su teléfono inteligente.

Aunque este trabajo va estar orientado a la aplicación mencionada en el apartado anterior, los conocimientos utilizados en la misma pueden ser utilizados para muchos otros propósitos apasionantes como la clasificación automática de imágenes por su contenido, lectura de libros para personas invidentes, entre otras muchas. Posibilidades que considero muy interesantes y en las cuales podría trabajar en un futuro gracias a los conceptos aprendidos durante la realización de este trabajo.

## 1.2 Objetivos

El principal objetivo de este proyecto es la creación de una aplicación capaz de detectar y reconocer texto en imágenes de escenas naturales, es decir, imágenes tomadas en situaciones de la vida cotidiana, como por ejemplo fotografías tomadas en la calle o en un centro comercial.

Los factores clave de esta aplicación deben ser la precisión en la detección y reconocimiento del texto y el tiempo de ejecución. El tiempo de ejecución debe de ser reducido al máximo para obtener una ejecución en tiempo real de la aplicación, debido a que los algoritmos de detección de texto suelen tener un alto coste computacional. Por otra parte, se debe intentar que el error tanto en la detección como en el reconocimiento del texto sean lo más reducidos posibles para evitar crear confusión al usuario de la aplicación.

Para la consecución de los objetivos planteados se deberá estudiar los algoritmos de estado del arte en detección y reconocimiento de texto, para posteriormente incluirlos en un algoritmo propio que sea capaz de satisfacer los requisitos establecidos en este apartado.

# Capítulo 2. Estado del arte

A lo largo de este capítulo vamos a analizar los principales conceptos que se han estudiado para la realización del trabajo. Nos centraremos en 2 principales grupos, los cuales son: Algoritmos de localización y reconocimiento de texto, y plataformas sobre las cuales es interesante aplicar la aplicación a desarrollar.

## 2.1 Algoritmos de localización y reconocimiento de texto

La localización y reconocimiento de texto en imágenes naturales, es decir, imágenes tomadas en escenas de la vida cotidiana ha sido un campo en el que se ha investigado mucho en la última década. A diferencia del reconocimiento de texto en documentos, tarea que se ha resuelto con éxito gracias a los algoritmos de estado del arte de OCR (Optical Character Recognition), la localización y reconocimiento de texto en imágenes naturales sigue siendo un problema abierto, donde aún no se ha conseguido una solución completamente satisfactoria para los problemas que este campo plantea. Las mayores dificultades que se plantean a la hora de localizar y reconocer textos en imágenes naturales son, entre otros: los fondos de los textos no uniformes; la inmensa variedad de tipografías y fuentes; el alineamiento del texto no sigue las normas estrictas de los documentos impresos; muchas de las encontradas son nombres propios (marcas, nombres de negocios, etc.) por lo que utilizar un diccionario no es efectivo.

La mayoría de los métodos de reconocimiento y localización de texto publicados están basados en un procesado secuencial que consta de 3 etapas: localización de candidatos, segmentación del texto y por último reconocimiento de texto mediante sistemas OCR para documentos impresos. En dichas implementaciones, el porcentaje de éxito del algoritmo es un producto de los ratios de cada una de sus etapas.

Según los algoritmos de localización de texto que implementen dichos métodos pueden ser clasificados en 2 grupos: métodos basados en ventana deslizante, métodos basados en componentes conexas.

Los métodos de **ventana deslizante** se basan en pasar una ventana deslizante a lo largo de toda la imagen para encontrar posibles candidatos a texto, para posteriormente utilizar técnicas de 'machine learning' (aprendizaje automático) para identificar textos. Este tipo de métodos suelen ser lentos debido a que la imagen debe ser procesada en múltiples escalas.

Los métodos basados en **componentes conexas** obtienen candidatos a letras analizando características comunes en píxeles vecinos, para posteriormente agrupar dichos candidatos en texto, además se deben realizar comprobaciones para evitar falsos positivos. Los métodos de este tipo más utilizas son los basados en **MSER** (Maximally Stable Extremal Region) y **ER** (Extremal regions).

A lo largo de los siguientes subapartados se detallarán implementaciones de los diversos métodos descritos anteriormente, para posteriormente poder hacer una elección informada sobre que método puede ser el adecuado para la creación de la aplicación propuesta.

### 2.1.1 *Método basado en regiones MSER (Maximally Stable Extremal Regions)*

Para comprender el funcionamiento de los métodos basados en regiones MSER, el primer paso que se debe tomar es definir que propiedades tienen dichas regiones y que las hace interesantes para la localización de textos en escenas naturales.

Una región MSER es por definición una componente conexa estable a lo largo de diferentes niveles de gris en una imagen, es decir, es una agrupación de píxeles que permanece uniforme cuando se le aplica un amplio rango de umbrales distintos a la imagen de la que forma parte. La palabra 'Extremal' hace referencia a la propiedad de que todos los píxeles dentro de la región MSER tendrán a un valor muy elevado (bright extremal regions) o un valor muy bajo (dark extremal regions) en comparación con los pixeles de su contorno exterior.

La principal característica que hace que los métodos basados en regiones MSER sean adecuados para la localización de textos en escenas naturales es su buen funcionamiento en imágenes que contienen regiones homogéneas con bordes distintivos. Además, dichos métodos muestran unas buenas prestaciones con regiones pequeñas, una característica muy importante también, debido a que los textos en muchos casos ocupan un porcentaje muy pequeño del tamaño total de la imagen.

Una vez definidas las regiones MSER y sus características principales, es posible analizar un método de localización y detección de texto en imágenes naturales basado en dichas regiones. Concretamente se va a estudiar el método presentado por Jiri Matas y Lukas Neumann en noviembre de 2010. El diagrama de bloques de dicho método puede observarse en la Figura 1.

**Figura 1: Diagrama de bloques del método basado en regiones MSER**

Como se aprecia en la Figura 1, el primer paso del método propuesto es la obtención de la MSER, para posteriormente clasificar dichas regiones en letra o no letra utilizando características de las mismas (relación de aspecto, número de agujeros, entre otras). En la Figura 2 pueden un ejemplo de las MSER obtenidas en una imagen, las zonas verdes son regiones clasificadas como letras, por contrapartida las regiones rojas son clasificadas como no letras.

**Figura 2: Clasificación letra no letra**

Posteriormente se forman líneas de texto a partir de las regiones etiquetadas como letra, partiendo desde la región que se encuentra más arriba a la derecha de todas las clasificadas, creando una cadena de regiones con propiedades similares, es decir, palabras. En la Figura 3 puede apreciarse como se forman las hipótesis de relación entre las diferentes regiones para finalmente seleccionar la que con más probabilidad será una palabra.

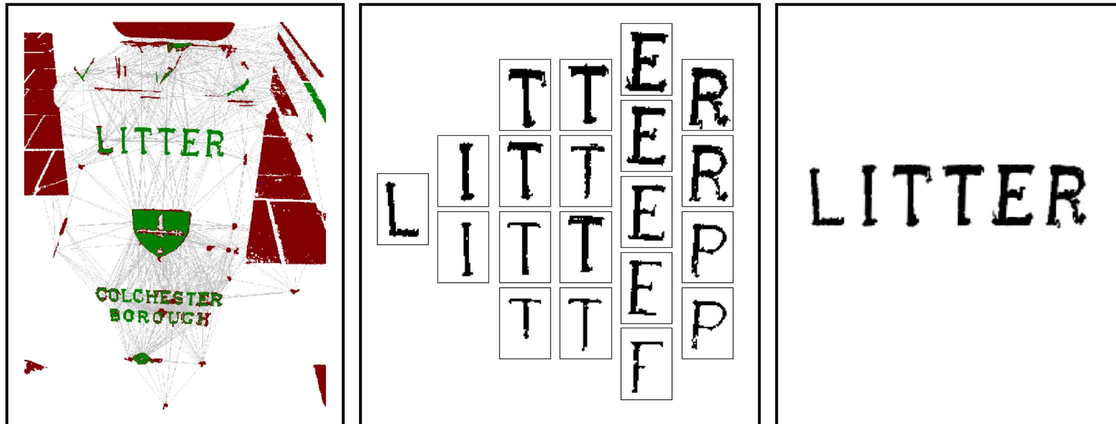


**Figura 3: Formación de líneas de texto**

Una vez obtenidas las posibles líneas de texto se procede a aplicar una transformación geométrica para corregir cualquier distorsión que se pueda producir en el texto por la perspectiva en la que se ha tomado la imagen, debido a que los algoritmos de reconocimiento de texto están entrenados con textos vistos desde una perspectiva frontal. En la Figura 4 puede observarse como se transforma la orientación del texto para prepararlo para el algoritmo de reconocimiento de caracteres OCR.

**Figura 4: Normalización Geométrica**

El algoritmo OCR estudiará cada región individualmente para obtener la letra que está representando dicha región, para posteriormente estudiar la separación entre los caracteres con el objetivo de discernir si la línea de texto que se está estudiando está formada por más de una palabra. Para finalizar la ejecución del método se comparará cada palabra seleccionada individualmente con un diccionario para obtener finalmente la lectura que el algoritmo ha hecho de la imagen.

**Figura 5: Resultados del método basado en regiones MSER**

### 2.1.2 *Método basado en regiones ER (Extremal Region)*

Del mismo modo que en el apartado anterior para proceder a estudiar el método basado en regiones ER, es necesario conocer que características tienen dichas regiones y porque son interesantes para la localización de texto.

Las regiones MSER explicadas en el subapartado anterior son un caso particular de las regiones ER, siendo la estabilidad el factor que las hace particulares. Por tanto, las regiones ER son el mismo concepto que el explicado anteriormente, pero con la diferencia de que dichas regiones no deben tener estrictamente un tamaño similar a lo largo de los diferentes umbrales aplicados a la imagen. En la Figura 6 puede observarse el comportamiento de una imagen a través de los diferentes umbrales (de 0 a 255) aplicados para obtener las regiones ER de la misma.

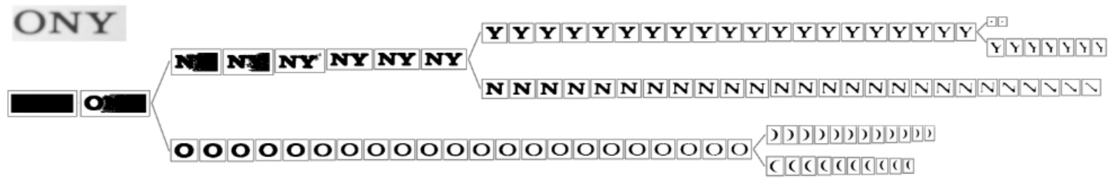


**Figura 6: Obtención de regiones ER**

El método basado en regiones ER que se va a analizar fue también propuesto por Jiri Matas y Lukas Neumann, en junio de 2012. Este método es superior al anterior debido a que utiliza menos memoria para su ejecución y muestra mejores resultados en imágenes borrosas.

El primer paso del método que estamos analizando es obtener las regiones ER, para posteriormente clasificar dichas regiones en caracter o no caracter a través de dos etapas bien diferenciadas. Primero se realizará un primer filtrado utilizando características que son poco exigentes computacionalmente (área, *bounding box*, perímetro, etc), que ayudarán a deshacerse de muchas regiones no deseadas. Posteriormente se obtendrán las regiones clasificadas como texto mediante un segundo filtro que evaluará características más exigentes computacionalmente (relación de casco convexo, número de puntos de inflexión de contorno exterior y relación de área de los orificios). Una vez obtenidos los caracteres se agruparán en palabras para finalmente ser procesados por un algoritmo OCR que mostrará la interpretación obtenida de la imagen de entrada.

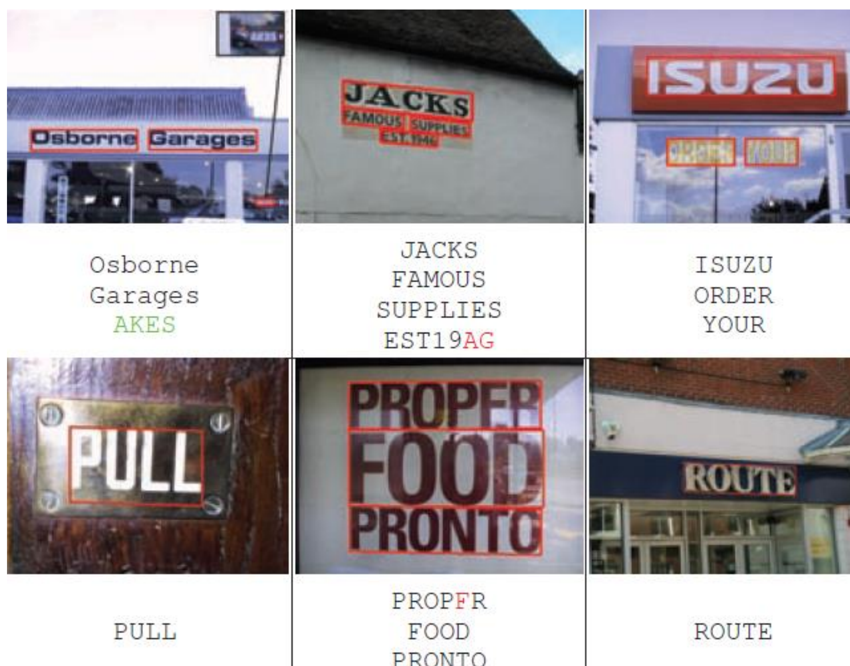


**Figura 7: Resultados del método basado en regiones ER**

## 2.2 Plataformas

### *2.2.1 Ubuntu*

La plataforma en la que se ha realizado todo el desarrollo de código ha sido Ubuntu, una distribución Linux que ofrece un sistema operativo predominante enfocado a ordenadores de escritorio, aunque también proporciona soporte para servidores.

Basada en Debian GNU/Linux, Ubuntu concentra su objetivo en la facilidad de uso, la libertad en la restricción de uso, los lanzamientos regulares (cada 6 meses) y la facilidad en la instalación. Ubuntu es patrocinado por Canonical Ltd., una empresa privada fundada y financiada por el empresario sudafricano Mark Shuttleworth.

El nombre de la distribución proviene del concepto zulú y xhosa de *ubuntu*, que significa "humanidad hacia otros" o "yo soy porque nosotros somos". Ubuntu es un movimiento sudafricano encabezado por el obispo Desmond Tutu, quien ganó el Premio Nobel de la Paz en 1984 por sus luchas en contra del *Apartheid* en Sudáfrica. El sudafricano Mark Shuttleworth, mecenas del proyecto, se encontraba muy familiarizado con la corriente. Tras ver similitudes entre los ideales de los proyectos GNU, Debian y en general con el movimiento del software libre, decidió aprovechar la ocasión para difundir los ideales de *Ubuntu*. El eslogan de Ubuntu – "Linux para seres humanos" (en inglés "Linux for Human Beings") – resume una de sus metas principales: hacer de Linux un sistema operativo más accesible y fácil de usar.

#### *2.2.1.1 Historia*

El 8 de julio de 2004, Mark Shuttleworth y la empresa Canonical Ltd. anunciaron la creación de la distribución Ubuntu. Ésta tuvo una financiación inicial de 10 millones de dólares (US$). El proyecto nació por iniciativa de algunos programadores de los proyectos Debian, Gnome porque se encontraban decepcionados con la manera de operar del proyecto Debian, la distribución Linux sin ánimo de lucro más popular del mundo.

De acuerdo con sus fundadores, Debian era un proyecto demasiado burocrático donde no existían responsabilidades definidas y donde cualquier propuesta interesante se ahogaba en un mar de discusiones. Asimismo, Debian no ponía énfasis en estabilizar el desarrollo de sus versiones de prueba y sólo proporcionaba auditorías de seguridad a su versión estable, la cual era utilizada sólo por una minoría debido a la poca o nula vigencia que poseía en términos de la tecnología Linux actual.

Tras formar un grupo multidisciplinario, los programadores decidieron buscar el apoyo económico de Mark Shuttleworth, un emprendedor sudafricano que vendió la empresa Thawte a VeriSign, cuatro años después de fundarla en el garaje de su domicilio, por 575 millones de dólares estadounidenses.

Shuttleworth vio con simpatía el proyecto y decidió convertirlo en una iniciativa autosostenible, combinando su experiencia en la creación de nuevas empresas con el talento y la experiencia de los programadores de la plataforma Linux. De esta forma nació la empresa Canonical, la cual se encarga de sostener económicamente el proyecto mediante la comercialización de servicios y soporte técnico a otras empresas. Mientras los programadores armaban el sistema, Shuttleworth aprovechó la ocasión para aplicar una pequeña campaña de mercadotecnia para despertar interés en ladistribución sin nombre (en inglés: the no-name-distro).

Tras varios meses de trabajo y un breve período de pruebas, la primera versión de Ubuntu (Warty Warthog) fue lanzada el 20 de octubre de 2004.

#### 2.2.1.2 *Características*

1. Basada en la distribución Debian.
2. Disponible en 4 arquitecturas: Intel x86, AMD64, SPARC (para esta última sólo existe la versión servidor).
3. Los desarrolladores de Ubuntu se basan en gran medida en el trabajo de las comunidades de Debian y GNOME.
4. Las versiones estables se liberan cada 6 meses y se mantienen actualizadas en materia de seguridad hasta 18 meses después de su lanzamiento.
5. La nomenclatura de las versiones no obedece principalmente a un orden de desarrollo, se compone del dígito del año de emisión y del mes en que esto ocurre. La versión 4.10 es de octubre de 2004, la 5.04 es de abril de 2005, la 5.10 de octubre de 2005, la 6.06 es de junio de 2006, la 6.10 es de octubre de 2006 y la 7.04 es de abril de 2007.
6. El entorno de escritorio oficial es Gnome y se sincronizan con sus liberaciones.
7. Para centrarse en solucionar rápidamente los bugs, conflictos de paquetes, etc. se decidió eliminar ciertos paquetes del componente main, ya que no son populares o simplemente se escogieron de forma arbitraria por gusto o sus bases de apoyo al software libre. Por tales motivos inicialmente KDE no se encontraba con más soporte de lo que entregaban los mantenedores de Debian en sus repositorios, razón por la que se sumó la comunidad de KDE distribuyendo la distro llamada Kubuntu.
8. De forma sincronizada a la versión 6.06 de Ubuntu, apareció por primera vez la distribución Xubuntu, basada en el entorno de escritorio XFce.
9. El navegador web oficial es Mozilla Firefox.
10. El sistema incluye funciones avanzadas de seguridad y entre sus políticas se encuentra el no activar, de forma predeterminada, procesos latentes al momento de instalarse. Por eso mismo, no hay un firewall predeterminado, ya que no existen servicios que puedan atentar a la seguridad del sistema.
11. Para labores/tareas administrativas en terminal incluye una herramienta llamada sudo (similar al Mac OS X), con la que se evita el uso del usuario root (administrador).
12. Mejora la accesibilidad y la internacionalización, de modo que el software está disponible para tanta gente como sea posible. En la versión 5.04, el UTF-8 es la codificación de caracteres en forma predeterminada.
13. No sólo se relaciona con Debian por el uso del mismo formato de paquetes deb, también tiene uniones muy fuertes con esa comunidad, contribuyendo con cualquier cambio directa e inmediatamente, y no solo anunciándolos. Esto sucede en los tiempos de lanzamiento. Muchos de los desarrolladores de Ubuntu son también responsables de los paquetes importantes dentro de la distribución de Debian.
14. Todos los lanzamientos de Ubuntu se proporcionan sin costo alguno. Los CDs de la distribución se envían de forma gratuita a cualquier persona que los solicite mediante el servicio ShipIt (la versión 6.10 no se llegó a distribuir de forma gratuita en CD, pero la versión 7.04 sí). También es posible descargar las imágenes ISO de los discos por transferencia directa o bajo la tecnología Bittorrent.
15. Ubuntu no cobra honorarios por la suscripción de mejoras de la “Edición Enterprise”.

#### 2.2.1.3 *Organización de paquetes*

Ubuntu divide todo el software en cuatro secciones, llamadas **componentes**, para mostrar diferencias en licencias y la prioridad con la que se atienden los problemas que informen los usuarios. Estos componentes son: main, restricted, universe y multiverse.

Por defecto, se instala una selección de paquetes que cubre las necesidades básicas de la mayoría de los usuarios de computadoras. Los paquetes de Ubuntu generalmente se basan en los paquetes de la rama inestable (*Sid*) de Debian.

##### 2.2.1.3.1 *El componente main*

El componente *main* contiene solamente los paquetes que cumplen los requisitos de la licencia de Ubuntu, y para los que hay soporte disponible por parte de su equipo. Éste está pensado para que incluya todo lo necesario para la mayoría de los sistemas Linux de uso general. Los paquetes de este componente poseen ayuda técnica garantizada y mejoras de seguridad oportunas.

##### 2.2.1.3.2 *El componente restricted*

El componente *restricted* contiene el programa soportado por los desarrolladores de Ubuntu debido a su importancia, pero que no está disponible bajo ningún tipo de licencia libre para incluir en *main*. En este lugar se incluyen los paquetes tales como los controladores propietarios de algunas tarjetas gráficas, como, por ejemplo, los de NVIDIA. El nivel de la ayuda es más limitado que para *main*, puesto que los desarrolladores puede que no tengan acceso al código fuente.

##### 2.2.1.3.3 *El componente universe*

El componente *universe* contiene una amplia gama del programa, que puede o no tener una licencia restringida, pero que no recibe apoyo por parte del equipo de Ubuntu. Esto permite que los usuarios instalen toda clase de programas en el sistema guardándolos en un lugar aparte de los paquetes soportados: *main* y *restricted*.

##### 2.2.1.3.4 *El componente commercial*

Como lo indica su clasificación, contiene programas comerciales.

##### 2.2.1.3.5 *El componente multiverse*

Finalmente, se encuentra el componente *multiverse*, que contiene los paquetes sin soporte debido a que no cumplen los requisitos de Software Libre.

#### 2.2.1.4 *Variantes*

Existen diversas variantes de Ubuntu disponibles, las cuales poseen lanzamientos simultáneos con Ubuntu. Las más significativas son:

1. Kubuntu, el cual utiliza KDE en vez de GNOME.
2. Edubuntu, diseñado para entornos escolares.
3. Xubuntu, el cual utiliza el entorno de escritorio Xfce.

Kubuntu, Edubuntu y Xubuntu son proyectos oficiales de la Ubuntu Foundation. Kubuntu y Edubuntu se encuentran incluidos dentro del programa *ShipIt*.

Mark Shuttleworth también ha apoyado la creación de una distribución derivada de Ubuntu que utilizaría sólo software aprobado por la Free Software Foundation. Hasta ahora no ha sido lanzada ninguna versión oficial de 'Ubuntu-Libre', debido a dificultades en la gestión de lanzado el 2 de noviembre de 2006. Sin embargo, no es una versión oficial de Ubuntu.

**Figura 8: Logo Ubuntu**

### 2.2.2 *Android*

Android es un sistema operativo basado en el núcleo Linux. Fue diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tablets o tabléfonos; y también para relojes inteligentes, televisores y automóviles. Inicialmente fue desarrollado por Android Inc., empresa que Google respaldó económicamente y más tarde, en 2005, adquirió. Android fue presentado en 2007 junto a la fundación del Open Handset Alliance (un consorcio de compañías dehardware, software y telecomunicaciones) para avanzar en los estándares abiertos de los dispositivos móviles. El primer móvil con el sistema operativo Android fue el HTC Dream y se vendió en octubre de 2008. Los dispositivos Android venden más que las ventas combinadas de Windows Phone e IOS.

El éxito del sistema operativo se ha convertido en objeto de litigios sobre patentes en el marco de las llamadas «Guerras por patentes de teléfonos inteligentes» (en inglés, Smartphone patent wars) entre las empresas de tecnología. Según documentos secretos filtrados en 2013 y 2014, el sistema operativo es uno de los objetivos de las agencias de inteligencia internacionales.

#### *2.2.2.1 Historia*

En octubre de 2003, en la localidad de Palo Alto, Andy Rubin, Rich Miner, Chris White y Nick Sears fundan Android Inc. con el objetivo de desarrollar un sistema operativo para móviles basado en Linux. En julio de 2005, la multinacional Google compra Android Inc. El 5 de noviembre de 2007 se crea la Open Handset Alliance, un conglomerado de fabricantes y desarrolladores de hardware, software y operadores de servicio. El mismo día se anuncia la primera versión del sistema operativo: Android 1.0 Apple Pie. Los terminales con Android no estarían disponibles hasta el año 2008.

Las unidades vendidas de teléfonos inteligentes con Android se ubican en el primer puesto en los Estados Unidos, en el segundo y tercer trimestres de 2010, con una cuota de mercado de 43,6% en el tercer trimestre. A escala mundial alcanzó una cuota de mercado del 50,9% durante el cuarto trimestre de 2011, más del doble que el segundo sistema operativo (iOS de Apple, Inc.).

Tiene una gran comunidad de desarrolladores creando aplicaciones para extender la funcionalidad de los dispositivos. A la fecha, se ha llegado ya al 1.000.000 de aplicaciones disponibles para la tienda de aplicaciones oficial de Android: Google Play, sin tener en cuenta aplicaciones de otras tiendas no oficiales para Android como la tienda de aplicaciones Samsung Apps de Samsung, slideme de java y amazon appstore. Google Play es la tienda de aplicaciones en línea administrada por Google, aunque existe la posibilidad de obtener software externamente. La tienda F-Droid es completamente de código abierto así como sus aplicaciones, una alternativa al software privativo. Los programas están escritos en el lenguaje de programación Java. No obstante, no es un sistema operativo libre de malware, aunque la mayoría de él es descargado de sitios de terceros.

El anuncio del sistema Android se realizó el 5 de noviembre de 2007 junto con la creación de la Open Handset Alliance, un consorcio de 78 compañías de hardware, software y telecomunicaciones dedicadas al desarrollo de estándares abiertos para dispositivos móviles. Google liberó la mayoría del código de Android bajo la licencia Apache, una licencia libre y de código abierto.

#### *2.2.2.2 Característias*

La estructura del sistema operativo Android se compone de aplicaciones que se ejecutan en un framework Java de aplicaciones orientadas a objetos sobre el núcleo de las bibliotecas de Java en una máquina virtual Dalvik con compilación en tiempo de ejecución hasta la versión 5.0, luego cambió al entorno Android Runtime (ART).

Las bibliotecas escritas en lenguaje C que contiene el sistema operativo incluyen: un administrador de interfaz gráfica (surface manager), un framework OpenCore, una base de datos relacional SQLite, una Interfaz de programación de API gráfica OpenGL ES 2.0 3D, un

motor de renderizado WebKit, un motor gráfico SGL, SSL y una biblioteca estándar de CBionic. El sistema operativo está compuesto por 12 millones de líneas de código, incluyendo 3 millones de líneas de XML, 2,8 millones de líneas de lenguaje C, 2,1 millones de líneas de Java y 1,75 millones de líneas de C++.

###### *2.2.2.2.1 Diseño de dispositivo*

La plataforma es adaptable a pantallas de gran resolución resolución, VGA, biblioteca de gráficos 2D, biblioteca de gráficos 3D basada en las especificaciones de la OpenGL ES 2.0 y diseño de teléfonos tradicionales.

###### *2.2.2.2.2 Almacenamiento*

Para el propósito de almacenamiento de información, se utiliza SQLite, una base de datos liviana, que proporciona una gestión de información con bajo coste computacional.

###### *2.2.2.2.3 Conectividad*

Android soporta las siguientes tecnologías de conectividad: GSM/IDEN, CDMA, EVDO, UMTS, Bluetooth, Wi-Fi, LTE, HSDPA, HSPA+, NFC y WiMAX. GPRS, UMTS y HSDPA+.

###### *2.2.2.2.4 Soporte de Java*

Aunque la mayoría de las aplicaciones están escritas en Java, no hay una máquina virtual Java en la plataforma. El bytecode Java no es ejecutado, sino que primero se compila en un ejecutable Dalvik y se ejecuta en la Máquina Virtual Dalvik. Dalvik es una máquina virtual especializada, diseñada específicamente para Android y optimizada para dispositivos móviles que funcionan con batería y que tienen memoria y procesador limitados. A partir de la versión 5.0, se utiliza Android Runtime (ART).

###### *2.2.2.2.5 Soporte para Hardware adicional*

Android soporta cámaras de fotos, de vídeo, pantallas táctiles, GPS, acelerómetros, giroscopios, magnetómetros, sensores de proximidad y de presión, sensores de luz, gamepad, termómetro y aceleración por GPU 2D y 3D.

###### *2.2.2.2.6 Entorno de desarrollo*

Incluye un emulador de dispositivos, herramientas para depuración de memoria y análisis del rendimiento del software. Inicialmente el entorno de desarrollo integrado (IDE) utilizado era Eclipse con el plugin de Herramientas de Desarrollo de Android (ADT). Ahora se considera como entorno oficial Android Studio, descargable desde la página oficial de desarrolladores de Android.

###### *2.2.2.2.7 Multi-táctil*

Android tiene soporte nativo para pantallas capacitivas con soporte multi-táctil que inicialmente hicieron su aparición en dispositivos como el HTC Hero. La funcionalidad fue originalmente desactivada a nivel de kernel (posiblemente para evitar infringir patentes de otras compañías). Más tarde, Google publicó una actualización para el Nexus One y el Motorola Droid que activaba el soporte multi-táctil de forma nativa.

###### *2.2.2.2.8 Multitarea*

Multitarea real de aplicaciones está disponible, es decir, las aplicaciones que no estén ejecutándose en primer plano reciben ciclos de reloj.

#### *2.2.2.2.9 Compilación Nativa*

Es posible crear aplicaciones Android utilizando código escrito en C++, característica que será de vital importancia para la implementación del código creado en la aplicación Android que se pretende crear.

### *2.2.2.3 Arquitectura*

Las componentes principales del sistema operativo Android son:

1. **Aplicaciones**: las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en lenguaje de programación Java.
2. **Marco de trabajo de aplicaciones**: los desarrolladores tienen acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes, cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de las mismas (sujeto a reglas de seguridad del framework).
3. **Runtime de Android**: Android incluye un set de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecutaba hasta la versión 5.0 archivos en el formato Dalvik Executable (.dex), el cual está optimizado para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato.dex por la herramienta incluida "dx". Desde la versión 5.0 utiliza el ART, que compila totalmente al momento de instalación de la aplicación.
4. **Bibliotecas**: Android incluye un conjunto de bibliotecas de C/C++ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del marco de trabajo de aplicaciones de Android; algunas son: System C library (implementación biblioteca C estándar), bibliotecas de medios, bibliotecas de gráficos, 3D y SQLite, entre otras.
5. **Núcleo Linux**: Android depende de Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto de la pila de software.

**Figura 9: Arquitectura global Android**

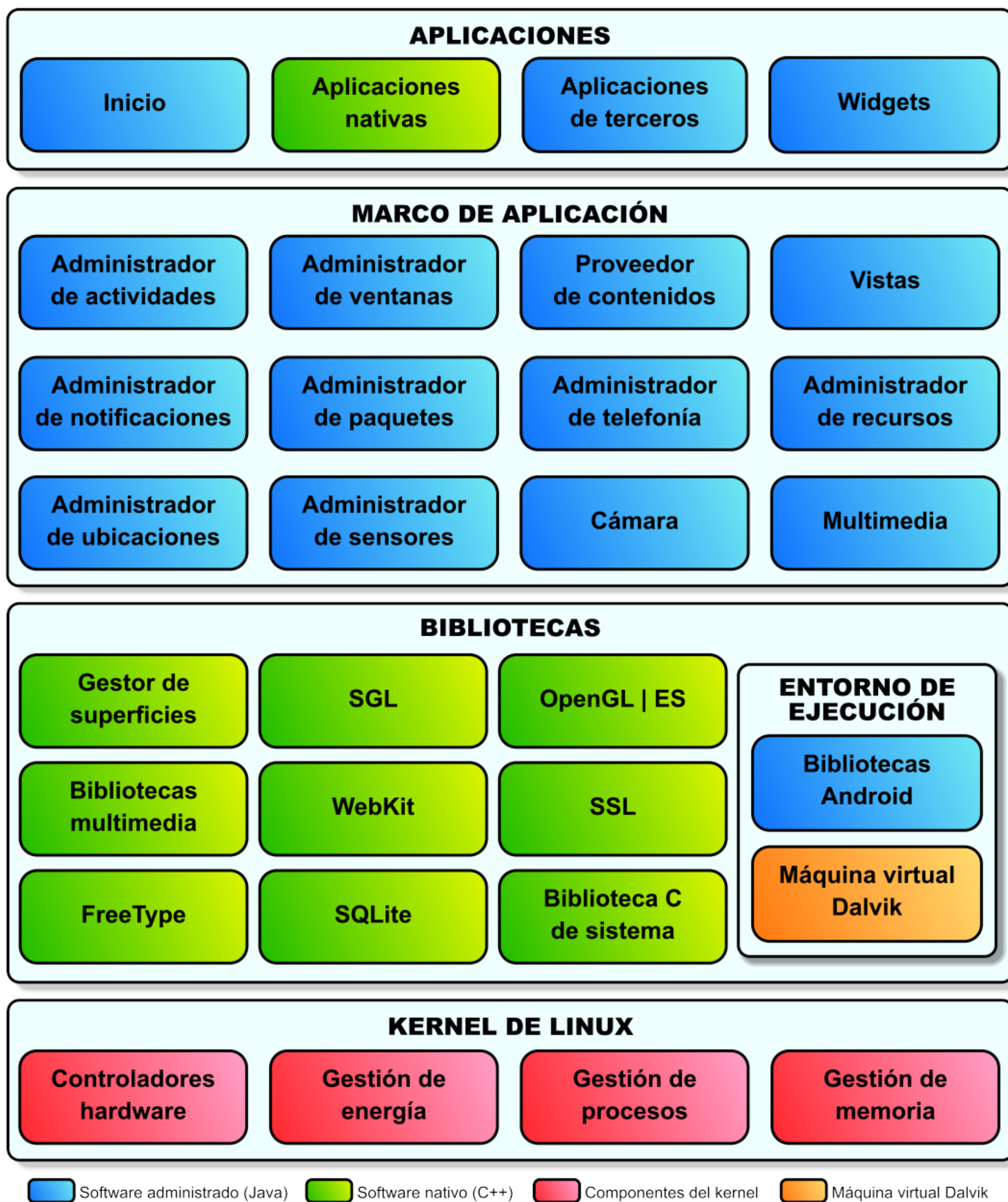


**Figura 10: Arquitectura específica Android**

#### 2.2.2.4 *Aplicaciones*

Las aplicaciones se desarrollan habitualmente en el lenguaje Java con Android Software Development Kit (Android SDK), pero están disponibles otras herramientas de desarrollo, incluyendo un Kit de Desarrollo Nativo para aplicaciones o extensiones en C o C++, Google App Inventor, un entorno visual para programadores novatos y varios marcos de aplicaciones basadas en la web multiteléfono.

El desarrollo de aplicaciones para Android no requiere aprender lenguajes complejos de programación. Todo lo que se necesita es un conocimiento aceptable de Java y estar en posesión del kit de desarrollo de software o «SDK» provisto por Google el cual se puede descargar gratuitamente.

Todas las aplicaciones están comprimidas en formato APK y se pueden instalar sin dificultad desde cualquier explorador de archivos en la mayoría de dispositivos.

#### 2.2.2.5 *Usos y dispositivos*

El sistema operativo Android se usa en teléfonos inteligentes, ordenadores, portátiles, netbooks, tabletas, Google TV, relojes de pulse, auriculares y otros dispositivos, siendo este sistema operativo accesible desde terminales de menos de 50€ hasta terminales que superan por mucho los 1000€.

La plataforma de hardware principal de Android es la arquitectura ARM. Hay soporte para x86 en el proyecto Android-x86,71 y Google TV utiliza una versión especial de Android x86.

#### 2.2.2.6 *Seguridad*

Según un estudio de Symantec de 2013,59 demuestra que en comparación con iOS, Android es un sistema explícitamente menos vulnerable. El estudio en cuestión habla de 13 vulnerabilidades graves para Android y 387 vulnerabilidades graves para iOS. El estudio también habla de los ataques en ambas plataformas, en este caso Android se queda con 113 ataques nuevos en 2012 a diferencia de iOS que se queda en 1 solo ataque. Incluso así Google y Apple se empeñan cada vez más en hacer sus sistemas operativos más seguros incorporando más seguridad tanto en sus sistemas operativos como en sus mercados oficiales.

**Figura 11: Logo de Android**

# Capítulo 3. Análisis del problema

A lo largo de este capítulo se va a realizar un análisis exhaustivo de las herramientas y tareas necesarias para poder crear un algoritmo de detección y reconocimiento de texto suficientemente rápido para funcionar en tiempo real en dispositivos móviles.

Este análisis va a estar formado por 4 pilares clave, que van a ser: la elección del método de estado del arte adecuado de entro los estudiados en el capítulo anterior; el análisis del software necesario para crear el algoritmo; la planificación, es decir, la división por tareas del problema y la distribución temporal de las mismas; y por último el presupuesto necesario para llevar a cabo el trabajo. A lo largo de los siguientes apartados se estudiarán en detalle cada uno de los procesos descritos.

## 3.1 Elección del método de detección y localización adecuado

Para recapitular lo estudiado en el capítulo anterior es necesario resaltar que en la actualidad existen 3 tipos de métodos principales para la detección y reconocimiento de texto en escenas naturales, los que están basados en regiones MSER, los basados en regiones ER, y por último los basados en ventana deslizante. Para poder elegir correctamente cual será el más adecuado debemos conocer cuáles son nuestras necesidades y cuáles son las principales ventajas y desventajas de dichos métodos.

Como se especificó en el apartado de objetivos, la meta principal de este trabajo es crear una aplicación para personas invidentes que sea capaz de detectar y reconocer texto en tiempo real, por tanto, la velocidad va a ser un factor clave a la hora de escoger un determinado método.

En la actualidad ninguno de los métodos de detección y reconocimiento de texto destaca por su velocidad, pero los métodos basados en ventana deslizante, por el contrario, destacan por su lentitud, por la tanto no son una alternativa viable para la realización del trabajo propuesto.

Una vez descartados los métodos por ventana deslizante tenemos que elegir entre los dos restantes, es decir métodos basados en regiones MSER o ER, puesto que ambos métodos tienen una velocidad de ejecución similar, este no puede ser el parámetro al que tengamos que prestar atención para decir entre uno u otro. En este caso deberemos de prestar atención a la precisión de ambos a la hora de detectar texto.

Si se repasa el estudio de los dos métodos restantes en el capítulo anterior, se podrá observar que los métodos basados en regiones ER se comportan mucho mejor con imágenes desenfocadas (borrosas), un tipo de imágenes que es muy factible que aparezcan durante el uso de la aplicación por parte de personas invidentes, por lo tanto, el método elegido para la creación del algoritmo de detección y reconocimiento de texto va a ser el basado en regiones ER.

Como se ha estudiado anteriormente la ejecución de dicho algoritmo en tiempo real no es factible en teléfonos móviles, por ello, se va a crear un algoritmo rápido de predetección de

texto propio para que este método pueda ser ejecutado de la forma más fluida posible en dispositivos móviles.

## 3.2 Software utilizado

Para la llevar a cabo la aplicación de detección y reconocimiento de texto propuesta se ha decidido crear una versión inicial del algoritmo en C++ debido a que es el lenguaje con el que más soltura se maneja el autor de este trabajo, para posteriormente portar dicho algoritmo a un sistema operativo móvil, que en este caso será Android debido a la mayor facilidad de acceso a su kit de desarrollo y a su superior cuota de mercado con respecto a las alternativas. Por último, será fundamental el uso de la librería de tratamiento de imagen OpenCV para la agilización del trabajo, debido a que sin ella la realización del mismo sería considerablemente más costosa.

A lo largo de los siguientes subapartados se van a explicar en detalle tanto los entornos de desarrollo utilizados como la librería OpenCV, con el objetivo de tener una mejor comprensión de las herramientas software utilizadas.

### *3.2.1 Eclipse*

Como entorno de desarrollo para el algoritmo en C++ se ha decidido utilizar Eclipse, debido a que es un IDE con el que se puede trabajar con mucha comodidad y rapidez. Algunas de sus características más destacables son las siguientes:

**Perspectivas, editores y vistas**: en Eclipse el concepto de trabajo está basado en las perspectivas, que no es otra cosa que una preconfiguración de ventanas y editores, relacionadas entre sí, y que nos permiten trabajar en un determinado entorno de trabajo de forma óptima.

**Gestión de proyectos**: el desarrollo sobre Eclipse se basa en los proyectos, que son el conjunto de recursos relacionados entre sí, como puede ser el código fuente, documentación, ficheros configuración, árbol de directorios, etc. El IDE nos proporcionará asistentes y ayudas para la creación de proyectos.

**Depurador de código**: se incluye un potente depurador, de uso fácil e intuitivo, y que visualmente nos ayuda a mejorar nuestro código. Para ello sólo debemos ejecutar el programa en modo depuración (con un simple botón). De nuevo, tenemos una perspectiva específica para la depuración de código, la **perspectiva depuración**, donde se muestra de forma ordenada toda la información necesaria para realizar dicha tarea.

**Extensa colección de** *plug-ins*: están disponibles en una gran cantidad, unos publicados por Eclipse, otros por terceros. Al haber sido un estándar de facto durante tanto tiempo (no el único estándar, pero sí uno de ellos), la colección disponible es muy grande. Los hay gratuitos, de pago, bajo distintas licencias, pero casi para cualquier cosa que nos imaginemos tenemos el *plug-in* adecuado.

**Figura 12: Logo IDE Eclipse C++**

### *3.2.2 OpenCV*

OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

Open CV es multiplataforma, existiendo versiones para GNU/Linux, Mac OS X, Windows, Android y IOS. Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estérea y visión robótica. También incluye funciones relacionadas con la detección y reconocimiento de texto, las cuales serán fundamentales para la creación del algoritmo creado a lo largo del trabajo propuesto.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado realizando su programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los procesadores multinúcleo. OpenCV puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel. Estas características de eficiencia y optimización serán clave para que la aplicación que se pretende crear funcione en tiempo real.

OpenCV presenta una estructura modular, es decir, sus diferentes funciones estas agrupadas dentro de módulos que pueden ser fácilmente adjuntadas a cualquier proyecto. Algunos de sus principales módulos son los siguientes:

**core**: Este es el módulo básico de OpenCV. Incluye las estructuras de datos básicas y las funciones esenciales de procesamiento de imágenes. Este módulo también es utilizado por otros módulos como highgui.

**highhui**: Este módulo provee interfaz de usuario, códecs de imagen y vídeo y capacidad para capturar imágenes y vídeo, además de otras capacidades como la de capturar eventos del ratón, etc.

**imgproc**: Este módulo incluye algoritmos básicos de procesado de imágenes, incluyendo filtrado de imágenes, transformado de imágenes, etc.

**video**: Este módulo de análisis de vídeo incluye algoritmos de seguimiento de objetos, entre otros.

Además de los módulos principales, que son los más soportados y estables, OpenCV posee módulos extra, que están menos desarrollados debido a que están basados en campos abiertos, como en este caso la detección y el reconocimiento de texto, los cuales se encuentran dentro del módulo extra llamado **text.**

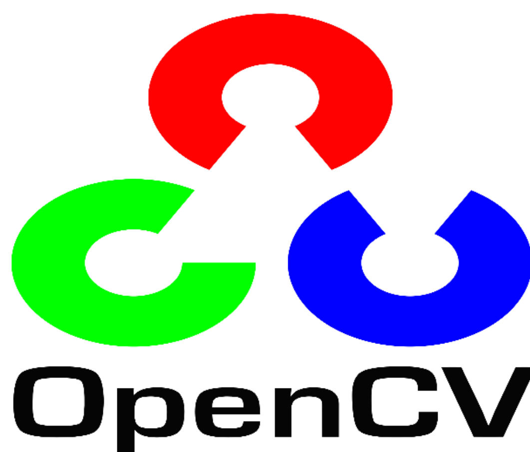


**Figura 13: Logo de OpenCV**

### *3.2.3 Android Studio*

Como entorno de desarrollo para Android se ha utilizado inicialmente Eclipse con el plugin ADT, el cual está diseñado para dotar a Eclipse de las configuraciones y opciones necesarias para desarrollar aplicaciones Android, pero se ha decidido dejar de utilizar dicho IDE debido a que este ya no recibía soporte. Por tanto, el entorno de desarrollo utilizado principalmente para programar en Android ha sido Android Studio.

Android Studio es el IDE oficial de Android para el desarrollo de aplicaciones. Basado en IntelliJ IDEA, un entorno o ambiente de desarrollo para programas que posee potentes herramientas de edición de código. Una de sus principales ventajas es su análisis de código, que al igual que Eclipse, destaca los errores cometidos al programar de forma inmediata, por lo que pueden ser solucionados por el programador de forma rápida y sencilla.

Como herramientas integradas para el desarrollo o construcción de las interfaces de las aplicaciones, Android Studio contiene un modo de previsualización de dichas interfaces, con variados modelos de pantalla, donde los elementos pueden ser desplazados o editados sin tener que acceder al código XML.

Recientemente Google ha lanzado la versión 2.0 de Android Studio, la cual implementa interesantes novedades como "Cloud Test Lab Integration", una característica que permite utilizar servicios en la nube para ejecutar la aplicación que se esté desarrollando en una gran variedad de dispositivos. Además, el emulador de Android para PC que incorpora Android Studio será hasta 4 veces más rápido que el incorporado en versiones anteriores del IDE.

Cabe destacar que la principal cualidad de Android Studio para el propósito de nuestra aplicación es que se permite la compilación de código Nativo, es decir, ofrece la posibilidad de poder utilizar código implementado en C++ para el desarrollo de aplicaciones. Aunque, la compilación de código en C++ es un proceso complejo si el proyecto depende de múltiples librerías como es el caso de la presente aplicación, que requiere de una extensa experiencia en la programación Android, por tanto, esta será una de las mayores dificultades a la crear la aplicación propuesta.

**Figura 14: Logo de Android Studio**

## 3.3 Planificación inicial y cambios de la misma

Para llevar a cabo cualquier proyecto con cierta envergadura es necesaria una planificación detallada de los pasos que se van a tomar para alcanzar con éxito los objetivos propuestos. A lo largo de los siguientes subapartados se van a detallar las distintas tareas en las que se ha creído conveniente dividir el presente trabajo, así como la distribución temporal de las mismas.

### *3.3.1 División por tareas*

En este subapartado se van a detallar las diferentes en las que se ha dividido la creación de la aplicación propuesta, así como la redacción de la memoria de la misma. Dichas tareas son las siguientes:

**Búsqueda de información**: para empezar el trabajo es fundamental hacer una búsqueda exhaustiva de información acerca de la localización y reconocimiento de texto, a partir de la cual ha sido posible escribir el capítulo 'Estado del Arte' de la presente memoria, además de hacer una elección informada del método más apropiado para implementar en la aplicación que se pretende desarrollar.

**Creación de un algoritmo de predetección**: como se ha explicado anteriormente, para que la aplicación propuesta funcione en tiempo real es necesario crear un algoritmo de predetección, cuya función primordial es aligerar la carga computacional final de la misma, mediante detección de zonas con alta probabilidad de contener texto, a través de algoritmos rápidos.

**Creación de un algoritmo de detección y reconocimiento de texto**: Una vez creado el algoritmo de predetección, el siguiente paso será combinar dicho algoritmo con el método de detección y reconocimiento de texto escogido, el cual está implementado en las funciones del módulo text de OpenCV.

**Aprendizaje del lenguaje Android**: Puesto que el desarrollador de la aplicación propuesta no conoce el lenguaje Android, será necesario incorporar el aprendizaje del mismo en la planificación. Dicho aprendizaje se llevará a cabo mediante videocursos impartidos en la propia Universidad Politécnica de Valencia por el profesor Jesús Tomás.

**Traducción del algoritmo de C++ a Android**: Una vez creado el algoritmo y aprendido el lenguaje de programación Android, el último paso necesario para la creación de la aplicación será portar el algoritmo creado a Android.

**Redacción de la memoria del proyecto**: Puesto que la memoria del proyecto es un requisito indispensable para la evaluación del mismo, es imprescindible incluirla en la planificación del mismo, debido a que es un documento extenso y de redacción costosa, por tanto, es pertinente tener esta tarea muy en cuenta para poder cumplir los plazos de entrega.

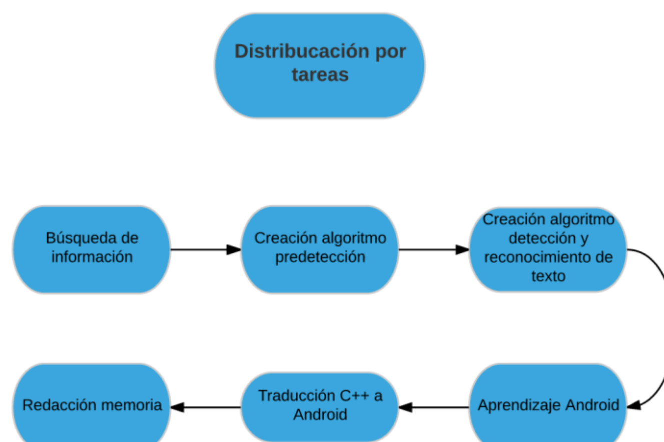


**Figura 15: Diagrama de flujo de las tareas del proyecto**

### *3.3.2 Distribución temporal inicial*

Una vez dividido el proyecto en las tareas necesarias para llevarlo a cabo, será pertinente asignarle una duración apropiada a las mismas para poder estimar cuanto tiempo será necesario para realizar el trabajo. A lo largo del actual subapartado se va a detallar la distribución temporal del proyecto, con la ayuda visual del diagrama de Gantt, el cual puede observarse en la Tabla 1. Dado que la distribución temporal reflejada en dicha tabla es solo una estimación, puede estar sujeta a cambios debido a las dificultades que puedan producirse durante la realización del trabajo. En caso de que dichos cambios se produzcan serán reflejados en el siguiente subapartado.

| Mes | Marzo | | | | | Abril | | | | Mayo | | | | | Junio | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Semana | 1-6 | 7-13 | 14-20 | 21-27 | 28-3 | 4-10 | 11-17 | 18-24 | 25-1 | 2-8 | 9-15 | 16-22 | 23-29 | 30-5 | 6-13 | 14-20 | 21-27 |
| Búsqueda de información | | | | | | | | | | | | | | | | | |
| Creación algoritmo predetección | | | | | | | | | | | | | | | | | |
| Creación algoritmo detección y reconocimiento de texto | | | | | | | | | | | | | | | | | |
| Aprendizaje Android | | | | | | | | | | | | | | | | | |
| Compilación C++ en Android | | | | | | | | | | | | | | | | | |
| Redacción memoria | | | | | | | | | | | | | | | | | |

**Tabla 1: Diagrama de Gantt inicial de la planificación del proyecto**

Como puede observarse en la tabla de Gantt la duración asignada a la **búsqueda de información** (color rojo en el diagrama) ha sido de **dos semanas**, dado que este proceso es de recopilación de información se ha considerado un tiempo más que suficiente para llevar a cabo dicha tarea.

Para **la creación del algoritmo de predetección** (color naranja en el diagrama) se ha tomado la decisión de asignar un intervalo de **un mes** debido a que se trata de una tarea compleja que necesitará de muchas pruebas para ser implementada de manera aceptable. Así pues, esta tarea es considerada una de las más complejas del proyecto, hecho que se ve reflejado en la distribución temporal.

A **la creación de un algoritmo de detección y localización de texto** (color verde en el diagrama) se ha decido asignarle el mismo tiempo que a la tarea anterior, es decir, **un mes**. Esta decisión se ha tomado por el mismo motivo que en la tarea anterior, la complejidad del trabajo a realizar, debido a que se necesitará un conocimiento profundo de las funciones de la librería OpenCV para poder implementarlas de manera adecuada en un algoritmo que cumpla los objetivos marcados.

En el caso del **aprendizaje del lenguaje Android** (color azul claro en el diagrama) se ha decidido asignar **2 semanas**, debido a que no se pretende obtener un nivel avanzado en el mismo, si no interiorizar los conceptos básicos para que se puede portar con suficiencia la aplicación inicial creada en C++. En caso de que se necesite más tiempo para alcanzar dicho objetivo podrá verse reflejado en el siguiente subapartado.

Para la **compilación del algoritmo en C++ en Android** (color azul oscuro en el diagrama), se ha decidido asignar **3 semanas**, dado que el objetivo es crear una aplicación prototipo para realizar una evaluación del algoritmo final en plataformas móviles. Si se pretendiese crear una aplicación mucho más funcional o estable el tiempo necesario para el desarrollo de la misma sería muy superior.

Finalmente se ha decidido dedicarle **2 semanas** a la **redacción de la presente memoria** (color morado en el diagrama) debido a que se ha calculado que a un ritmo de 5 páginas por día pueden llegar a escribirse 60, un número más que suficiente para describir con claridad el trabajo realizado a lo largo de los 4 meses de duración estimada del trabajo.

### *3.3.3 Distribución temporal final*

Como era de esperar a lo largo de la realización del proyecto se han presentado ciertos contratiempos que han hecho que se haya modificado la distribución en el tiempo de las tareas propuestas. A lo largo de este subapartado se incluye un nuevo diagrama de Gantt con la duración actualizada de las tareas (Tabla 2) así como una explicación de los cambios más importantes que se ha producido en dicha duración.

| Mes | Marzo | | | | | Abril | | | | Mayo | | | | | Junio | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Semana | 1-6 | 7-13 | 14-20 | 21-27 | 28-3 | 4-10 | 11-17 | 18-24 | 25-1 | 2-8 | 9-15 | 16-22 | 23-29 | 30-5 | 6-13 | 14-20 | 21-27 |
| Búsqueda de información | | | | | | | | | | | | | | | | | |
| Creación algoritmo predetección | | | | | | | | | | | | | | | | | |
| Creación algoritmo detección y reconocimiento de texto | | | | | | | | | | | | | | | | | |
| Aprendizaje Android | | | | | | | | | | | | | | | | | |
| Compilación C++ en Android | | | | | | | | | | | | | | | | | |
| Redacción memoria | | | | | | | | | | | | | | | | | |

**Tabla 2: Diagrama de Gantt final del proyecto**

Uno de los cambios más notorios y llamativos en el diagrama de Gantt se ha producido en la tarea de **búsqueda de información**, que como puede apreciarse ha pasado de ser de 2 semanas a un total de 10, debido a que se ha intercalado constantemente con las demás tareas. Esto ha sido así debido a que se ha necesitado buscar información constantemente a cerca de funciones que han sido incorporadas en los diferentes algoritmos que componen la aplicación que se ha desarrollado.

Otro cambio notable se ha producido en **la creación del algoritmo de predetección**, que en lugar de 4 semanas ha tomado 6, debido a su complejidad. Esto ha sido así debido a que el incremento de la velocidad del mismo suponía una disminución importante en su tasa de acierto. La consecuencia de este incremento de tiempo ha sido la necesidad de intercalar la programación del algoritmo de predetección con la del algoritmo de detección y reconocimiento de texto.

Debido a que se ha tenido que intercalar la programación de los dos algoritmos citados en el párrafo anterior la finalización del **algoritmo de detección y localización de texto** también ha llevado más tiempo del esperado, pasando de una duración de 4 semanas a 5. Este hecho ha producido que la finalización de este algoritmo se haya compaginado con el inicio del aprendizaje del lenguaje Android.

Un cambio menos notable, pero que también se cree conveniente destacar la prolongación en 1 semana del **aprendizaje del lenguaje Android** debido a que este es más complejo de lo que se estimó en la planificación inicial, tomando finalmente un total de 3 semanas.

Y por último un cambio que también ha sido muy significativo es la prolongación de la compilación del algoritmo en C++ en Android de 3 semanas a 5 semanas, debido la complejidad inesperada de dicha tarea, llevando a esto a no poder finalizar la aplicación móvil por falta de tiempo.

## 3.4 Presupuesto

A lo largo del presente apartado se va a analizar el presupuesto estimado del proyecto teniendo en cuenta los materiales utilizados, el coste de la mano de obra y el coste del local necesario para el trabajo.

En cuanto a los materiales utilizados podemos encontrar dos elementos básicos, los cuales serán descritos a continuación:

1) Ordenador portátil Dell Inspiron 15SE con las siguientes especificaciones técnicas:
   a) Procesador Intel Core i7-3612QM a 2.1 GHz
   b) 8 GB de memoria RAM
   c) Sistema operativo Windows 10
   d) 1 TB de HDD + 32 GB de SSD de almacenamiento
   e) Pantalla HD de 15''

2) Teléfono móvil Xiaomi Redmi Note 3 con las siguientes especificaciones técnicas:
   a) Procesador de ocho núcleos Helio X10 a 2 GHz
   b) Cámara de 13 megapíxeles
   c) 3 GB de memoria RAM
   d) 32 GB de almacenamiento interno

Con respecto a la mano de obra, solo ha trabajado en el proyecto un programador (el alumno), cuyo salario se estimará a partir del salario mínimo de un programador sin experiencia laboral. Y por último se considerará que el trabajo ha sido realizado en una oficina de 30 metros, fijando como coste de la misma el precio medio de los locales con dichas características en Valencia utilizando un comparador inmobiliario sumado con el gasto de limpieza del mismo.

| Emplazamiento de trabajo | Precio menusal | Número de meses | Precio Final |
|---|---|---|---|
| Oficina | 200,00 € | 4 | 800,00 € |
| Mantenimiento | 40,00 € | 4 | 160,00 € |
| | | Total | 960,00 € |

**Tabla 3: Presupuesto del emplazamiento de trabajo**

| Mano de obra | Precio menusal | Número de meses | Precio Final |
|---|---|---|---|
| Programador novel | 1.000,00 € | 4 | 4.000,00 € |
| | | Total | 4.000,00 € |

**Tabla 4: Presupuesto de la mano de obra**

| Herramientas de trabajo | Precio |
|---|---|
| Ordenador portátil DELL inspiron 15 SE | 1.000,00 € |
| Teléfono móvil Xiaomi Redmi Note 3 | 200,00 € |
| Licencias Software | - € |
| Total | 1.200,00 € |

**Tabla 5: Presupuesto de las herramientas de trabajo**

| Presupuesto total | Precio |
|---|---|
| Herraminetas de trabajo | 1.200,00 € |
| Mano de Obra | 4.000,00 € |
| Emplazamiento de trabajo | 960,00 € |
| Total | 6.160,00 € |

**Tabla 6: Presupuesto total**

Como puede observarse en la Tabla 3, comparando precios de oficinas de 30 metros cuadrados se ha estimado un coste de 200 € mensuales, al que se le han añadido 40 euros por el coste de mantenimiento, sumando un total de 240 € por mes trabajo. Esta cifra multiplicada por los 4 meses que se han estimado para la finalización del proyecto nos proporciona el coste total del emplazamiento de trabajo, el cual asciende a 960 €.

En cuanto a la mano de obra, se ha considerado que un salario aceptable para un programador novel asciende a 1000 € mensuales. Por tanto, el precio final de la mano de obra asciende a 4000€ debido a los 4 meses de trabajo necesarios, como puede observar en la Tabla 4.

Para calcular el precio de las herramientas de trabajo solo se ha necesitado sumar el precio del teléfono móvil y del ordenador portátil, debido a que no ha sido necesario la adquisición de ninguna licencia software para la realización del proyecto, ya que todo el software utilizado es completamente gratuito. Finalmente, el precio de las herramientas de trabajo asciende a 1200 €, como puede verse en la Tabla 5.

Para finalizar el presupuesto solo queda sumar todos los gastos reflejados en los párrafos anteriores, obteniendo un coste final de 6.160 € para la realización de nuestro proyecto, como se puede ver en la Tabla 6.

# Capítulo 4. Desarrollo y resultados del trabajo

A lo largo de este capítulo se van a explicar en profundidad los diferentes componentes del algoritmo programado para la consecución de los objetivos fundamentales establecidos en el inicio del proyecto. Dichos componentes pueden dividirse en dos partes fundamentales, el desarrollo en C++ y el desarrollo en Android. A lo largo de los siguientes apartados, dichas partes van ser estudiadas en profundidad para que el lector obtenga una comprensión profunda del funcionamiento del algoritmo propuesto.

## 4.1 Desarrollo en C++

La primera parte del desarrollo del proyecto ha sido llevada a cabo en C++, esta sección del proyecto es sin duda la más importante y fundamental, debido a que es la que más esfuerzo y estudio ha requerido para su consecución. Esto ha sido así debido a que para programar dicho algoritmo ha habido que realizar un profundo estudio de los métodos de detección y reconocimiento de texto además de dedicar una gran cantidad de tiempo probando diferentes formas de acelerar la ejecución del mismo.

### *4.1.1 Algoritmo de predetección*

El algoritmo de predetección se ha realizado debido a la necesidad de agilizar los algoritmos de detección y localización de texto, ya que estos no son lo suficientemente rápidos para funcionar en tiempo real. Pero dicha aceleración tendrá consecuencias negativas, como es de esperar. En este caso, el precio que se debe pagar para una detección más rápida es la no detección de ciertos textos, que el algoritmo de detección y reconocimiento de texto localizaría con normalidad, pero que no son detectados por el algoritmo de predetección.

Sopesando los pros y los contras comentados anteriormente, se ha considerado que la disminución de detecciones es un precio que es aceptable pagar con el fin de aumentar la velocidad de la aplicación, con la cual, se ha decidido incluir dicho algoritmo de predetección. En los siguientes subapartados se detallarán más en profundidad las virtudes y defectos del algoritmo creado.

#### *4.1.1.1 Reescalado de la imagen*

El tamaño de la imagen con la que se va a trabajar es un factor fundamental, por dos motivos principales. El más importante es la velocidad de procesamiento, es decir, a menor de tamaño de la imagen mayor va a ser la velocidad con la que será procesada, por tanto, reducir el tamaño de la misma será importante para obtener buenos resultados en los tiempos de ejecución del algoritmo. Por otra parte, una imagen demasiado pequeña no tendrá los detalles suficientes como para que se pueda trabajar con ella de forma apropiada, por lo cual, no se puede reducir la imagen a tamaños demasiado pequeños.

Mediante el método de prueba y revisión se ha llegado a la conclusión de que el punto intermedio para que nuestro algoritmo funcione de manera apropiada y a su vez lo haga con

unos tiempos de ejecución aceptables, está en un tamaño de alrededor de 1000 x 700 píxeles, por tanto, toda imagen que se utilice como input en nuestro algoritmo deberá ser reescalada a dicho tamaño.

Para analizar el comportamiento de nuestro algoritmo, se ha creado una biblioteca de más de 60 imágenes, tomadas con el teléfono Xiaomi Redmi Note 3 descrito en el apartado de presupuesto. Dichas imágenes tienen un tamaño de 4160 x 3120 píxeles, por lo que aplicarles un reescalado para reducir su tamaño es imprescindible para que el algoritmo funcione a una velocidad razonable. En la Figura 16 se puede ver de forma gráfica la proporción en la que se reducen las imágenes con las que se ha probado el algoritmo. Que como es de esperarse necesitan ser reducidas a una cuarta parte de su tamaño para alcanzar el tamaño óptimo mencionado en el párrafo anterior. En la Figura 17 dicha disminución puede apreciarse de forma numérica.

**Figura 16: Ejemplo gráfico del reescalado**

```
Tamaño de la imagen de entrada: 4160x3120
Tamaño de la imagen reescalada: 1040x780
```

**Figura 17: Ejemplo numérico del reescalado**

#### *4.1.1.2 Conversión a escalada de grises*

Las imágenes digitales a color de la actualidad están formadas a partir de 3 matrices de píxeles, cuya intensidad de píxel variará entre 0 y 1 o entre 0 y 255 dependiendo del formato. Estas tres matrices representan la intensidad del color rojo, verde y azul (RGB) respectivamente. Por ejemplo, en la matriz que representa el color rojo en la imagen y suponiendo un formato en el que la intensidad varía 0 y 255, un pixel con valor 255 en dicha matriz significará la presencia máxima del rojo en dicho punto, mientras que un píxel con valor 0 significará la ausencia de rojo. Si representamos cada una de estas matrices (o canales) individualmente podremos apreciar que cada una de ellas es una imagen de grises, y al combinarlas las 3 obtenemos una imagen a color. El objetivo de esta parte del algoritmo es obtener la matriz de grises que representará la imagen de entrada utilizando los tres canales explicados durante este párrafo.

Una forma visual de entender los conceptos explicados en el párrafo anterior es con el ejemplo visual que se puede apreciar en las Figuras 18 a la 21. Donde se muestran las 3 componentes que forman la imagen de entrada (Figura 18). Si el lector pone atención en el símbolo “$O_2$” se puede apreciar que en la componente azul de la imagen (Figura 20) tiene un valor claro, debido a que la presencia de azul en ese símbolo es elevada, sin embargo, en las dos componentes restantes (Figuras 19 y 21), se puede apreciar que dicho símbolo está oscuro, ya que su color no contiene ni rojo ni verde. En el ejemplo propuesto se pueden encontrar más casos de este tipo en los otros canales, debido a que se ha escogido una imagen en la que aparecen los colores rojo, verde y azul, para hacer la explicación lo más visual posible para el lector.

**Figura 18: Imagen de entrada**

**Figura 19: Canal rojo (R)**

**Figura 21: Canal verde (G)**

**Figura 20: Canal azul (B)**

Las 3 canales de la imagen explicados anteriormente van a ser utilizados por nuestro algoritmo para obtener la imagen a escala de grises de la imagen a color de la entrada, debido a que a partir de este punto solo trabajaremos con dicha imagen de grises ya que es monocanal, lo cual va a hacer que las operaciones que realicemos posteriormente sean 3 veces más rápidas, debido a que solo se deberá procesar un canal en lugar de 3. Para obtener dicha imagen de grises, la librería de tratamiento de imagen utilizada (OpenCV) incorpora una función con la que realizar dicho proceso muy fácilmente. En la Figura 23 se puede apreciar la salida que obtenemos de dicha función al utilizar como entrada la imagen de la Figura 22 (reescalada convenientemente en la sección anterior de nuestro algoritmo).

Figura 22: Imagen reescalada

Figura 23: Imagen de grises

#### *4.1.1.3 Obtención de los bordes*

Una vez obtenida la imagen de grises, el siguiente paso que realiza el algoritmo creado es la detección de los bordes de la imagen, que serán las zonas en la que la imagen cambia bruscamente en pixeles cercanos, dichas zonas son conocidas también como altas frecuencias.

Para la detección de bordes se ha utilizado un método conocido en tratamiento de imagen como "Canny Detector", cuyas prestaciones son las siguientes:

1. **Buena detección**, lo cual significa que el algoritmo consigue marcar la mayoría de los bordes de la imagen.
2. **Buena localización**, es decir, los bores marcados por el algoritmo están muy cerca a los bordes de la imagen real, y en la mayoría de los casos la posición coincide exactamente.
3. **Respuesta mínima**, que es una característica del algoritmo que le permite marcar los bordes encontrados en la imagen una vez, y al mismo tiempo evita que el ruido de la misma genere falsos bordes.

Estas características son conseguidas gracias a que es un algoritmo basado en la reducción de ruido, el cálculo del gradiente, la eliminación de pixeles no considerados parte de un borde y por último la histéresis. A continuación, se van a explicar en detalle dichos métodos.

Es inevitable que las imágenes tomadas por una cámara tengan cierta cantidad de ruido. Para prevenir que dicho ruido sea confundido con bordes debe de ser reducido en la medida de lo posible. El proceso que se encargará de reducir dicho ruido será el emborronado de la imagen mediante la aplicación de un filtro Gaussiano (también conocido como filtro paso bajo en tratamiento de imagen). Mediante la convolución de la imagen de grises obtenida en la sección anterior del algoritmo, obtendremos una imagen de grises más borrosa, pero con menos ruido.

En la Figura 24 podemos observar la Matriz de filtro Gaussiano con una desviación típica de σ = 1.4 y un tamaño de 5x5, y en las Figuras 25 y 26 los efectos de convolucionar la imagen de grises con dicha matriz.

$$B = \frac{1}{159} \cdot \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

**Figura 24: Kernel del filtro Gaussiano**

**Figura 26: Imagen de grises**

**Figura 25: Imagen de grises ligeramente emborronada**

El algoritmo "Canny detector", básicamente, encuentra bordes donde la intensidad en la imagen de grises sufre cambios más bruscos, dichos cambios son localizados mediante el cálculo de los gradientes de la imagen. Los gradientes de la imagen en cada punto son obtenidos aplicando el operador de Sobel. El primer paso es aproximar el gradiente en la dirección X y en la dirección Y, convolucionando la imagen de grises con los operadores de las Figuras 27 y 28 respectivamente.

$$K_{\mathrm{GX}} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

**Figura 27: Operador Sobel coordenada X**

$$K_{\mathrm{GY}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

**Figura 28: Operador Sobel coordenada Y**

Finalmente, la magnitud del gradiente en cada punto puede ser obtenida como la distancia Euclídea como puede verse en la Ecuación (1), aunque en ocasiones esta expresión es simplificada para reducir el coste computacional, como puede verse en la Ecuación (2). En la Figura 30 puede observarse el resultado obtenido al calcular el gradiente de la imagen de grises.

$$|G| = \sqrt{G_{\mathrm{x}}^2 + G_{\mathrm{y}}^2} \quad \textbf{(1)}$$

$$|G| = |G_{\mathrm{x}}| + |G_{\mathrm{y}}| \quad \textbf{(2)}$$

**Figura 29: Imagen de grises ligeramente emborronada**

**Figura 30: Gradiente**

Como puede apreciarse en la imagen 25, los bordes obtenidos en el gradiente no son suficientemente precisos para trabajar con ellos. Por este motivo deberán ser procesados y mejorados para obtener bordes más ajustados a los bordes reales de la imagen. Un parámetro de suma importancia para realizar dichos ajustes es la dirección de los ejes, que debe ser determinada a partir de la expresión que puede verse en la Ecuación 3.

$$\theta = \arctan\left(\frac{|G_{\mathrm{y}}|}{|G_{\mathrm{x}}|}\right) \quad \textbf{(3)}$$

El siguiente paso a realizar para la obtención de los bordes de la imagen será la eliminación de los píxeles que no sean considerados máximos. El objetivo de este procedimiento es convertir los ejes borrosos obtenidos por el gradiente en bordes más finos y precisos. Para conseguir dicho objetivo se tendrá que aplicar el siguiente procedimiento a cada pixel del gradiente:

1. Redondear la dirección del gradiente en el píxel a la subdivisión de 45º más cercana, dividiendo así todas las posibles direcciones en 8 grupos.
2. Comparar la magnitud del gradiente de dicho píxel, con la magnitud del gradiente de los pixeles en la dirección del gradiente obtenida, en el paso anterior, tanto en dirección positiva como negativa. Por ejemplo, si se ha obtenido que la dirección del píxel es de 90º, se comparará dicho píxel con el píxel de arriba y con el de abajo.

3. Si la magnitud del píxel es mayor que la de los píxeles con los que se le ha comparado en el paso anterior, se conserva la magnitud de dicho pixel. En caso contrario, se pondrá a cero la magnitud del gradiente de dicho píxel.

Un ejemplo simple del procedimiento explicado puede verse en la Figura 31, donde la mayoría de los píxeles tienen orientación norte (90º), por tanto, son comparados con los píxeles de arriba y de abajo. Los píxeles que han resultado ser máximos en dicha comparación están marcados con bordes blancos, siendo todos los demás suprimidos.

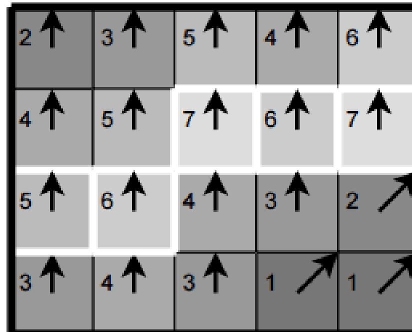


**Figura 31: Ejemplo de supresión de píxeles no máximos**

Una vez llevada a cabo la supresión de los no máximos, los píxeles de la imagen continúan definidos por la magnitud del gradiente de la imagen original. Muchos de estos píxeles pueden ser, en efecto, bordes de la dicha imagen, pero algunos de ellos pueden estar causados por ruido o incluso por variaciones bruscas del color en determinados tipos de superficies. La manera más sencilla de clasificar un píxel como parte de un borde real o, como producto del ruido, es la umbralización, la cual consiste en fijar un cierto valor de magnitud del gradiente, y solo en caso de que la magnitud del píxel sea superior a dicho valor fijado será clasificado como borde.

El algoritmo "Canny detector" utiliza para este propósito un doble umbral, es decir, se fijarán dos magnitudes del gradiente diferentes, una superior y otra inferior. Los píxeles cuya magnitud sea mayor al umbral superior serán clasificados como bordes "fuertes", por contrapartida, los píxeles cuya magnitud de gradiente sea más baja que el umbral inferior, serán suprimidos, y finalmente, los píxeles cuya magnitud se encuentre entre los dos umbrales serán considerados como bordes "débiles".

Una vez los bordes bordes han sido clasificados como "fuertes" y "débiles", el último paso a realizar para completar la ejecución del algoritmo "Canny detector" es el seguimiento de bordes por histéresis. Que consiste en la eliminación de todos los bordes "débiles" salvo los que estén conectados a bordes "fuertes", debido a que hay una alta probabilidad que los demás bordes "débiles" hayan sido producidos por el ruido existente en la imagen original.

EL seguimiento de bordes puede ser implementado mediante el "BLOB-analysis" (Binary Large OBject). Los píxeles de la imagen obtenida después de la aplicación de la doble umbralización son segmentados en BLOBs conexos, utilizando una clasificación de componentes conexas con vecindad 8 (Esto significa que se analizan como posibles componentes conexas a cada uno de los 8 píxeles adyacentes a cada píxel). Todos los BLOBs que contengan al menos un borde "fuerte" serán conservados, mientas que todos los demás serán suprimidos. El efecto de los procesos mencionados anteriormente sobre el gradiente, pueden observarse en la Figura 32.

Figura 33: Gradiente

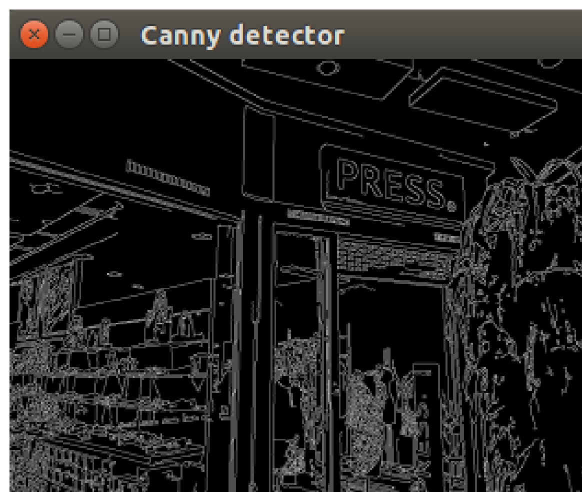


Figura 32: Bordes Canny detector

#### *4.1.1.4 Segmentación de la imagen de bordes*

Una vez obtenidos los bordes de la imagen, en los que claramente pueden apreciarse los contornos del texto (Figura 32), el siguiente paso será obtener una segmentación de dichos bordes. En este caso serán segmentados como contornos de figuras, es decir, nuestro objetivo es obtener una lista con los píxeles que componen cada uno de los contornos de la imagen, para posteriormente poder obtener parámetros de los mismos, con los que poder clasificar dichos contornos como letras en caso de que lo sean.

Para realizar esta tarea se ha utilizado una función implementada en la librería OpenCV llamada "findcontours", la cual devolverá un vector de listas de puntos, siendo cada una de las componentes de dicho vector una lista de los puntos que forman un borde concreto. Una vez utilizada "findcontours", se utilizará otra función (llamada "drawcontours") para poder ver de forma gráfica la segmentación en contornos llevada a cabo. "drawcontours" mostrará por pantalla los contornos segmentados con diferentes colores para que su distinción sea más sencilla, como puede apreciarse en la Figura 34.

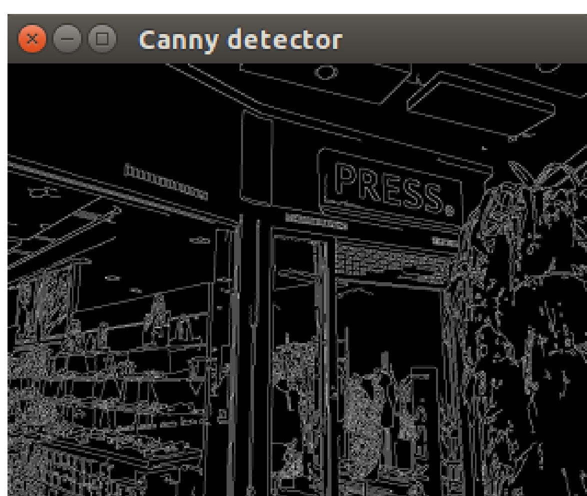


Figura 35: Bordes Canny Detector

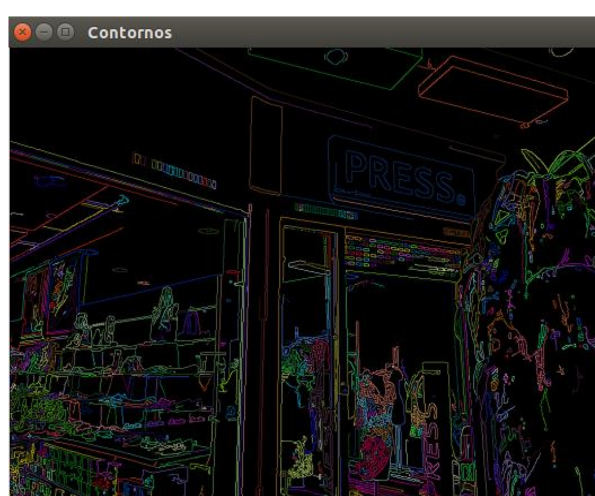


Figura 34: Segmentación por contornos findcontours

### *4.1.1.5 Obtención de parámetros de los bordes (Bounding Box)*

Una vez segmentados los diferentes objetos de la imagen gracias a los contornos obtenidos, podemos obtener características de los mismos. Para obtener dichas características utilizaremos las *Bounding Boxes* de dichos objetos, las cuales son una herramienta muy útil en el tratamiento de imágenes. Una *Bounding Box* es el rectángulo más pequeño en el que se puede encerrar un objeto. Gracias a las *Bounding Boxes* es posible obtener parámetros de los objetos como la altura, la anchura y el área, entre otros. En la Figura 36 podemos observar *Bounding Boxes* (cuadrados de color rosa) de forma gráfica.

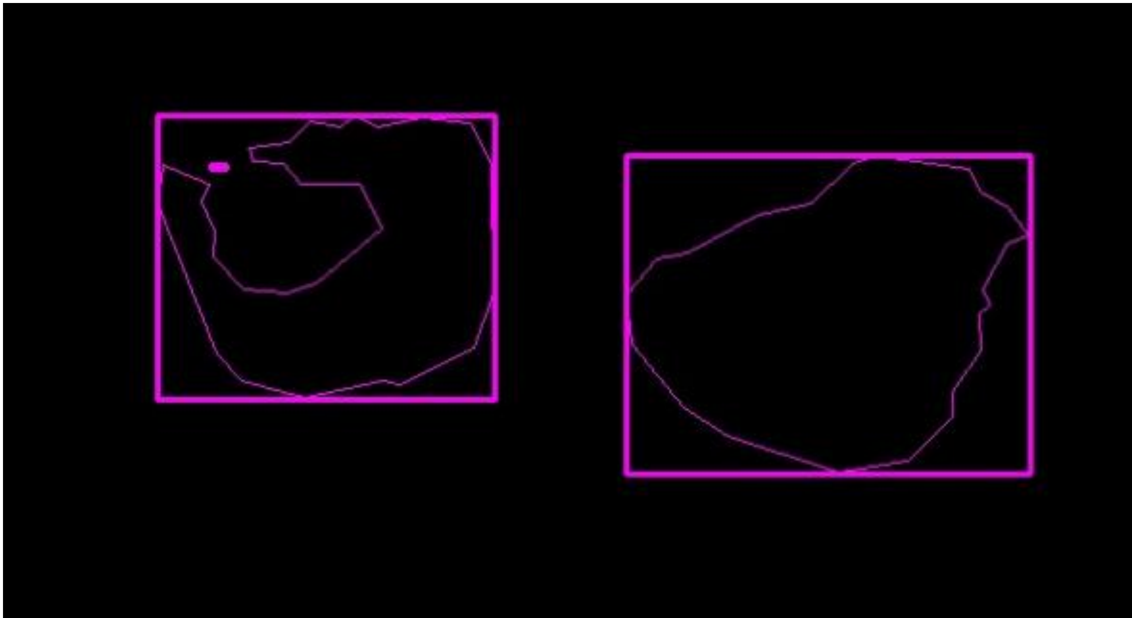

**Figura 36: Ejemplo de Bounding Boxes**

Para la obtención de los las *Bounding Boxes* de los contornos segmentados se ha utilizado otra función de la librería OpenCV llamada "boundingRect". Una vez obtenidas las *Bounding Boxes* se han calculado algunos parámetros de cada uno de los objetos segmentados, para la posterior clasificación de los mismos. Dichos parámetros son: el área, la anchura, la altura, la relación de aspecto y la coordenada Y del centro de la *Bounding Box*. La utilización de dichos parámetros será explicada en subapartados posteriores.

Para que las *Bounding Boxes* obtenidas puedan analizarse de forma gráfica se han dibujado sobre la imagen de contornos, gracias a la función de OpenCV "rectangle", la cual nos permite dibujar rectángulos en imágenes. El resultado de aplicar dicha función a la imagen de contornos obtenida en el apartado anterior puede apreciarse en la Figura 37.

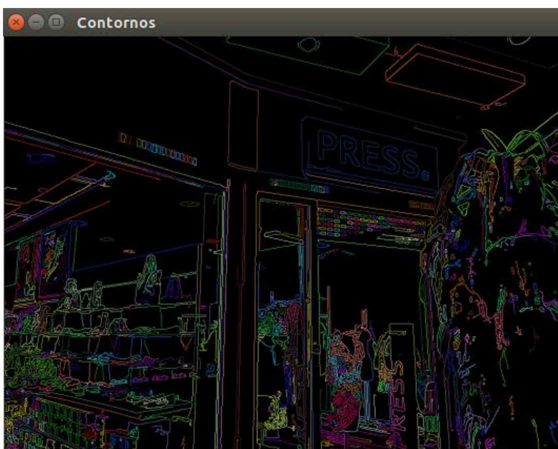


**Figura 38: Segmentación por contornos findcontours**

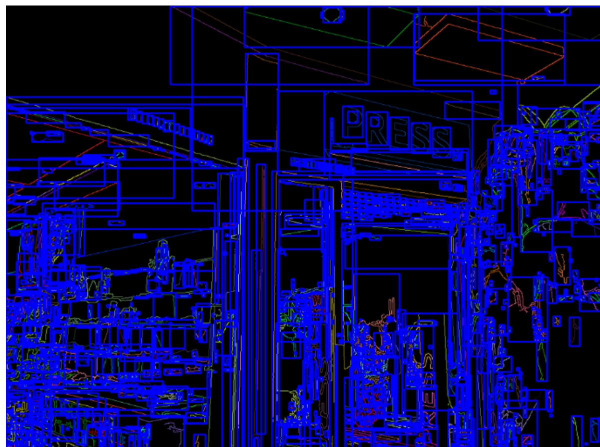

**Figura 37: Bounding Boxes sobre la imagen de contornos**

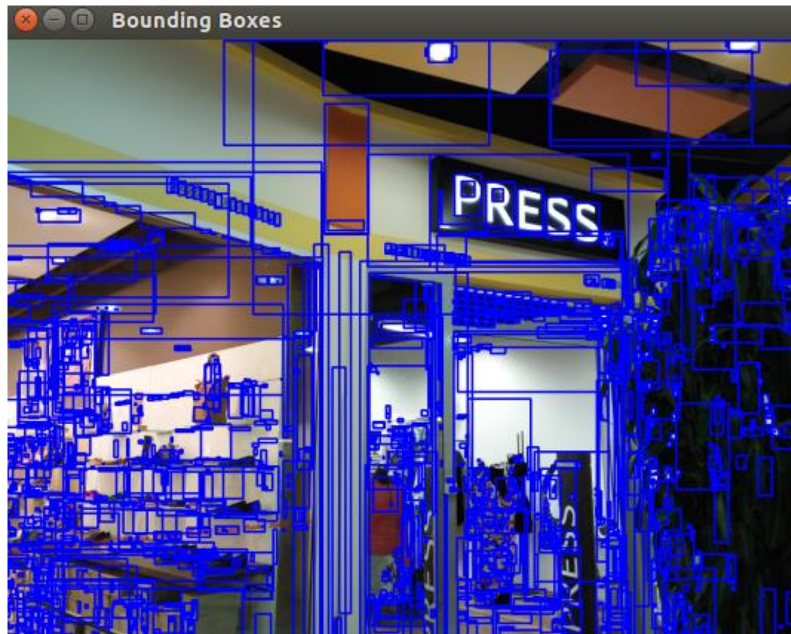


**Figura 39: Bounding Boxes sobre la imagen de entrada**

Como se puede ver en la Figura 37 es casi imposible distinguir la *Bounding Box* de un objeto individual debido a su abultada cantidad. Sin ir más lejos, la imagen de ejemplo contiene 1935 contornos, y por tanto 1138 *Bounding Boxes*. El objetivo de los siguientes bloques del algoritmo de predetección será pues, reducir dicha cantidad lo máximo posible, descartando todos los objetos que no sean letras.

```
Contornos obtenidos: 1935
```

**Figura 40: Número de contornos obtenidos en la imagen de ejemplo**

#### *4.1.1.6 Primer filtro de candidatos*

Como se ha comentado en el apartado anterior, el objetivo de los siguientes bloques será descartar el máximo número de objetos que no sean letras posible. Para ello utilizaremos las características obtenidas en el bloque anterior. En el caso del primer filtrado de objetos se utilizarán características poco exigentes computacionalmente, debido a que el número de objetos a analizar es elevado.

Para el descarte de objetos en este caso los parámetros utilizados han sido el área y la relación de aspecto. Estableciendo que el área de cado uno de los objetos debe ser estar comprendida entre un tamaño máximo (tamaño de la imagen divido entre 50) y un tamaño mínimo (tamaño de la imagen dividido por mil), debido a que los textos encontrados en escenas naturales no suelen ser de un tamaño muy elevado, y en caso de ser muy pequeños no podrán detectarse. En cuanto a la relación de aspecto, se ha decido que no pueda ser mayor que 2 o menor que 0.5, es decir, los objetos que sean el doble de anchos que altos y viceversa serán descartados, debido a que la inmensa mayoría de las letras presentan una relación de aspecto contenida en dicho rango. Los resultados del primer filtro de candidatos pueden observarse en la Figura 42.

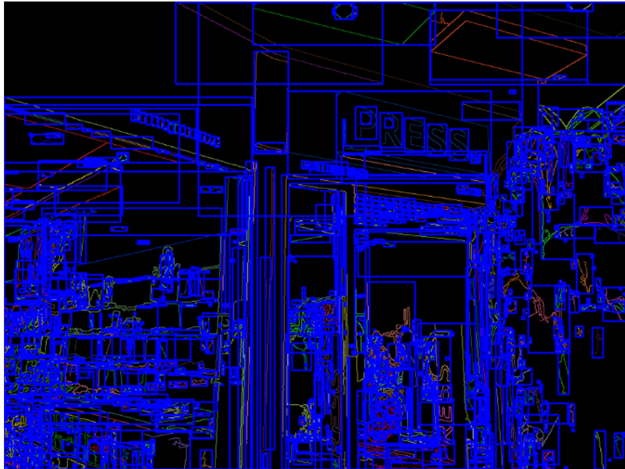

Figura 41: Imagen de Bounding Boxes

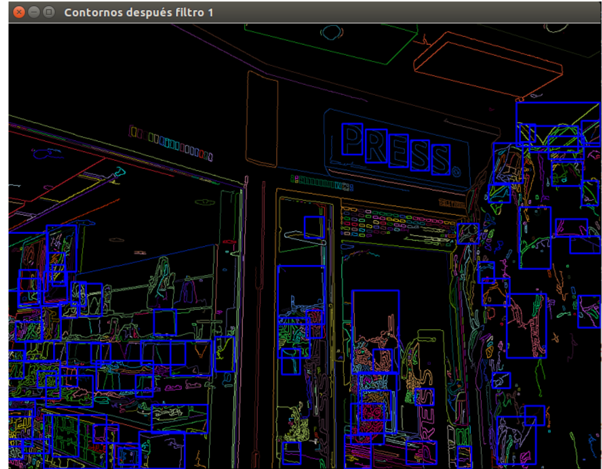


Figura 42: Bounding Boxes sobre la imagen de contornos después del primer filtro

Figura 43: Bounding Boxes sobre la imagen de entrada después del primer filtro

```
Contornos obtenidos: 1935
Número de contornos después del filtro 1: 81
```

Figura 44: Número de contornos antes y después del primer filtro

Como puede apreciarse tanto en la Figura 42 como en la Figura 43, el número de contornos se ve drásticamente reducido, pese a la simplicidad de las operaciones utilizadas. Gracias a esta reducción, podrán aplicarse operaciones computacionalmente más exigentes en los siguientes filtros. En la Figura 45 puede apreciarse una gráfica que muestra los resultados de la reducción del número de contornos de 16 imágenes utilizadas para testear el algoritmo, cuyo promedio de reducción es de 95.6%.

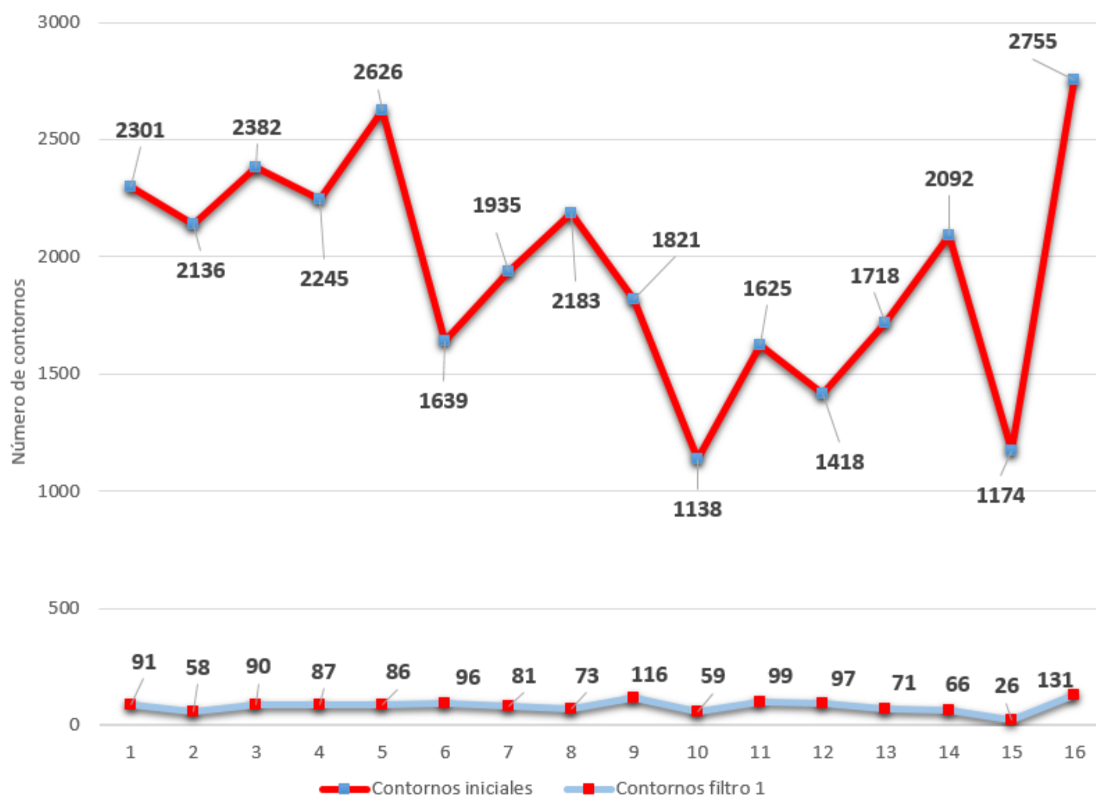


**Figura 45: Gráfica contornos iniciales vs contornos primer filtro**

#### *4.1.1.7 Segundo filtro de candidatos*

Una vez reducido el número de objetos en gran medida tras el primer filtrado es pertinente proceder a realizar el segundo filtrado, que va a consistir en crear un histograma de características tridimensional (altura, área y coordenada Y del centro de la *Bounding Box*). Este histograma va ser creado para agrupar objetos que tengan dichas características similares, debido a que los textos son, por definición, una agrupación de elementos de características similares (letras).

Se ha decidido escoger las características mencionadas en el apartado anterior debido a que las letras de una misma palabra suelen tener un área similar, presentan una altura parecida, y además están alineadas, es decir, la coordenada Y del centro de sus *Bounding Boxes* no varía en gran medida ente unas y otras.

Para crear el histograma tridimensional mencionado anteriormente se han decidido dividir las posibles alturas así como las coordenadas Y de los centros en 20 intervalos, y las áreas en 5, debido a que la variación en este parámetro es sensiblemente menor. Por tanto, los textos que se encuentren en la imagen deben generar picos en el histograma creado, de forma que, al hallar el máximo de dicho histograma estaremos hallando el grupo de objetos con mayor probabilidad de ser una palabra.

Una vez calculado el histograma tridimensional, y siguiendo la lógica del párrafo anterior, el siguiente paso debe ser encontrar el máximo de dicho histograma, para una vez hallado, clasificar los objetos que lo forman como posibles letras y descartar todos los demás.

Uno de los problemas de este método es que hay letras que pueden no ser descartadas debido a que sus características presenten variaciones suficientemente significativas a las demás para

hacerlas caer en otro grupo del histograma, para solucionar este problema ninguno de los grupos adyacentes al máximo es descartado, es decir, es clasificado como letra de igual forma.

Para incrementar en mayor medida la precisión de este método es posible seleccionar diversos máximos en lugar de solo uno, pero a costa de incrementar el coste computacional. Teniendo en cuenta que la velocidad es prioritaria en el algoritmo creado, se ha decidido sacrificar algo de precisión con el objetivo de incrementar la velocidad, es decir, solo se analizará un máximo. Los resultados al aplicar el segundo filtro a la imagen de ejemplo pueden verse en la Figura 47.

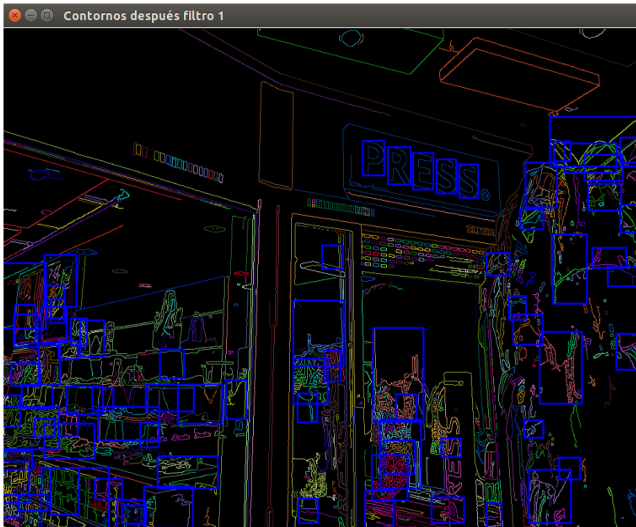


**Figura 46: : Bounding Boxes sobre la imagen de contornos después del primer filtro**

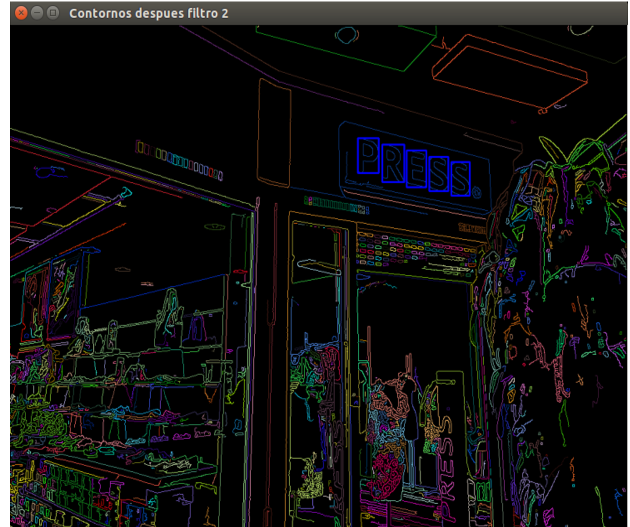


**Figura 47: : Bounding Boxes sobre la imagen de contornos después del primer filtro**

**Figura 48: Bounding Boxes sobre la imagen de entrada después del segundo filtro**

```
Número de contornos después del filtro 1: 81
Número de contornos después del filtro 2: 9
```

**Figura 49: Número de contornos antes y después del segundo filtro**

Como puede apreciarse tanto en la Figura 47 como el la Figura 48, el número de contornos ha vuelto a reducirse en gran medida, clasificando todas las letras de forma correcta. En la Figura 50 puede observarse una gráfica que refleja el grado de disminución del número de contornos después del segundo filtrado, cuyo promedio es del 85.41%.

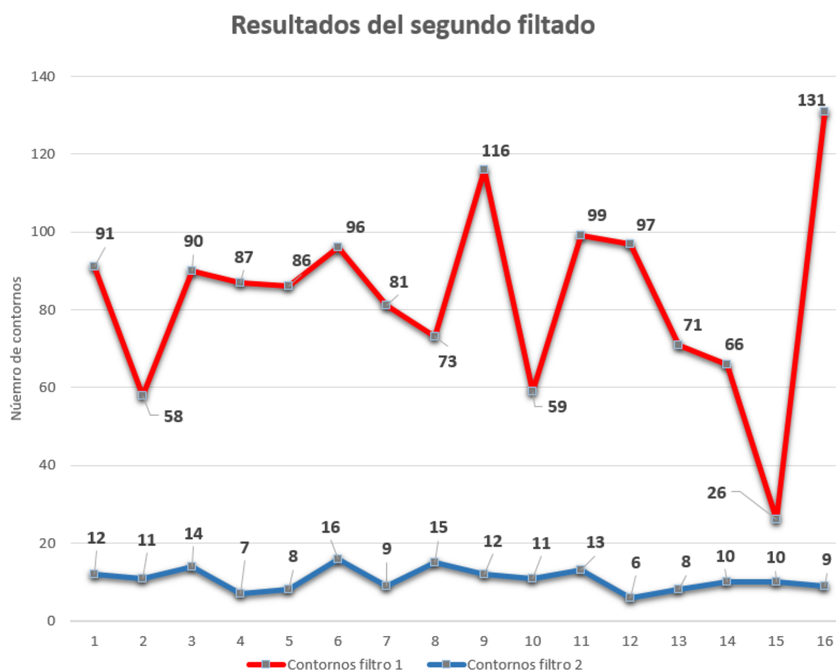


Figura 50: Gráfica contornos primer filtro vs contornos segundo filtro

Pese a que en la imagen de ejemplo se haya seleccionado el texto de manera correcta, existen casos en los que la selección no contiene ninguna letra, es decir, el algoritmo falla. Esta clase de errores pueden ser solucionados como se ha apuntado anteriormente con la selección de un mayor número de picos del histograma. Otro caso que sucede es que no todas las letras de la palabra sean detectadas, o que incluso que algún objeto espurio sea clasificado como letra junto a otros objetos que sí que lo son. En la imagen de la Figura 51 puede observarse una imagen en la que se producen algunas de estas inconveniencias.

Figura 51: Bounding Boxes sobre la imagen de entrada después del segundo filtro

```
Número de contornos después del filtro 1: 59
Número de contornos después del filtro 2: 11
```

**Figura 52: Número de contornos antes y después del segundo filtro en la imagen "Vodafone"**

Como puede apreciarse en la imagen 46 dos letras de la palabra "Vodafone" no son clasificadas como tal, y sin embargo un objeto espurio (en la parte superior derecha) es clasificado como letra. En los posteriores bloques del algoritmo se implementarán métodos para solventar los inconvenientes mencionados.

#### *4.1.1.8 Tercer filtro de candidato*

Al estudiar más en profundidad las imágenes de salida del segundo filtrado, se ha apreciado que hay más contornos clasificados de los que se muestran en la imagen, por tanto, será necesario eliminarlos. Un claro ejemplo de este problema puede apreciarse comparando las Figuras 51 y 52. En la Figura 51 podemos apreciar 7 *Bounding Boxes* dibujadas en la imagen, sin embargo, el número de contornos de salida del filtro 2 es de 11 según la Figura 52. Para estudiar a que se debe esta diferencia se va a pasar a estudiar el contenido del vector de *Bounding Boxes* que obtenemos a la salida del filtro 2, como se puede apreciar en la Figura 53.

```
[40 x 45 from (597, 127)]
[40 x 45 from (597, 127)]
[44 x 46 from (552, 123)]
[44 x 46 from (552, 123)]
[45 x 46 from (506, 120)]
[45 x 46 from (437, 114)]
[45 x 46 from (437, 114)]
[49 x 47 from (338, 105)]
[49 x 47 from (338, 105)]
[52 x 48 from (290, 100)]
[30 x 50 from (1004, 91)]
```

**Figura 53: Parámetros de las Bounding Boxes después del filtro 2**

Como se puede apreciar en la Figura 53, la descompensación entre el número de *Bounding Boxes* que se pueden ver en la Figura 51, y las que son realmente clasificadas (Figura 53), es debida a que algunas de las *Bounding Boxes* clasificadas están repetidas. Por tanto, el filtro 3 se encargará de eliminar dichas *Bounding Boxes* repetidas. La salida de dicho filtro puede verse en la Figura 54.

```
[40 x 45 from (597, 127)]
[44 x 46 from (552, 123)]
[45 x 46 from (506, 120)]
[45 x 46 from (437, 114)]
[49 x 47 from (338, 105)]
[52 x 48 from (290, 100)]
[30 x 50 from (1004, 91)]
```

**Figura 54: parámetros de las Bounding Boxes después del filtro 3**

```
Número de contornos después del filtro 2: 11
Número de contornos después del filtro 3: 7
```

**Figura 55: Número de contornos antes y después del tercer filtro en la imagen "Vodafone"**

Como se puede observar en las Figuras 54 y 55 los contornos repetidos han sido eliminados, por tanto, las *Bounding Boxes* que ahora vemos en la Figura 51 son las únicas que han sido clasificadas.

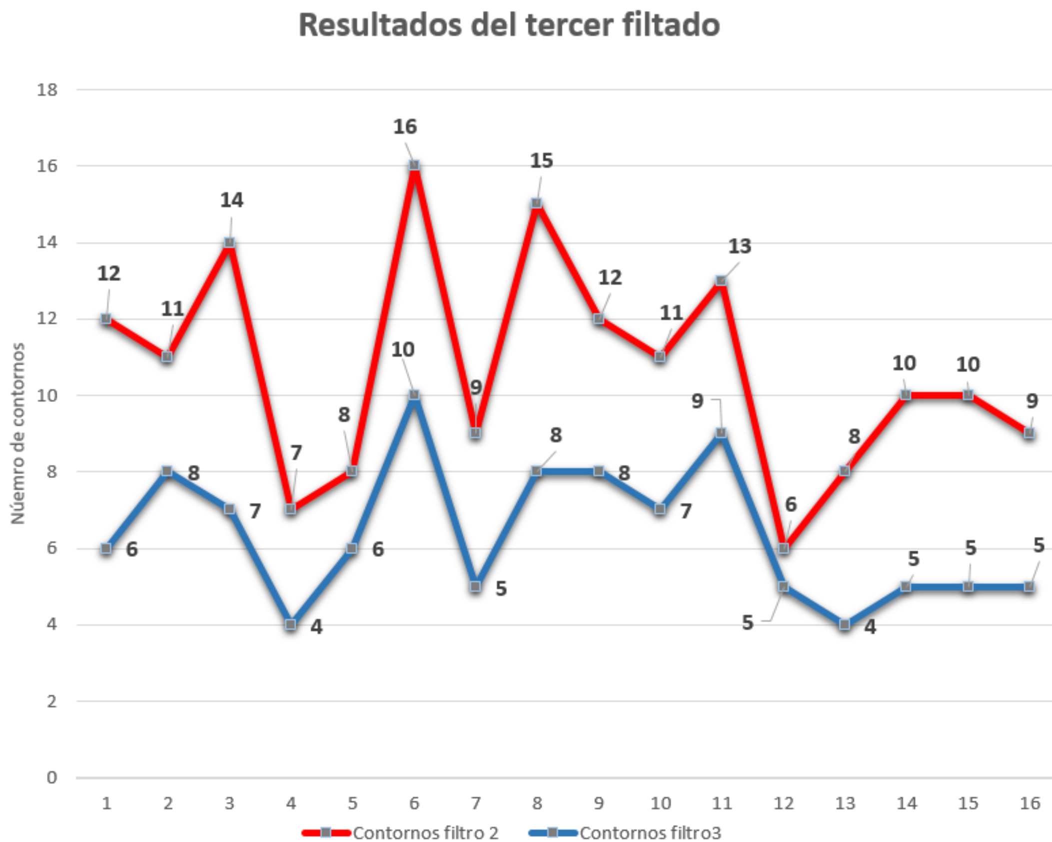


**Figura 56: Gráfica contornos segundo filtro vs contornos tercer filtro**

En la Figura 56 puede observarse la reducción del número de contornos clasificados antes y después del tercer filtrado, en este caso el promedio de reducción de objetos es del 39.71%. Es un porcentaje significativamente menor en comparación con el de los filtrados previos, pero no por ello es menos importante para el correcto funcionamiento del algoritmo desarrollado.

#### *4.1.1.9 Cuarto filtro de candidatos*

El cuarto filtrado va a ser el responsable de eliminar los objetos que son clasificados como letras pero que están desalineados o muy separados con respecto a los demás objetos clasificados. El objetivo de este filtro es pues, eliminar los objetos espurios que son clasificados como letras cuando en realidad no lo son, como el que podemos encontrar en la parte superior derecha de la Figura 58.

Para que este filtro funcione se compara cada uno de los objetos clasificados con los que tiene alrededor, y en caso de no tener ningún objeto cercano que esté alineado con él es descartado. Un ejemplo del funcionamiento de este algoritmo puede apreciarse al comparar la Figura 58 con la 57, donde se puede ver que el objeto espurio que aparece en la parte superior derecha de la Figura 58 no aparece en la Figura 57, porque ha sido descartado.

**Figura 58: Bunding Boxes sobre la imagen de entrada después del segundo filtro**

**Figura 57: Bounding Boxes sobre la imagen de entrada después del cuarto filtro**

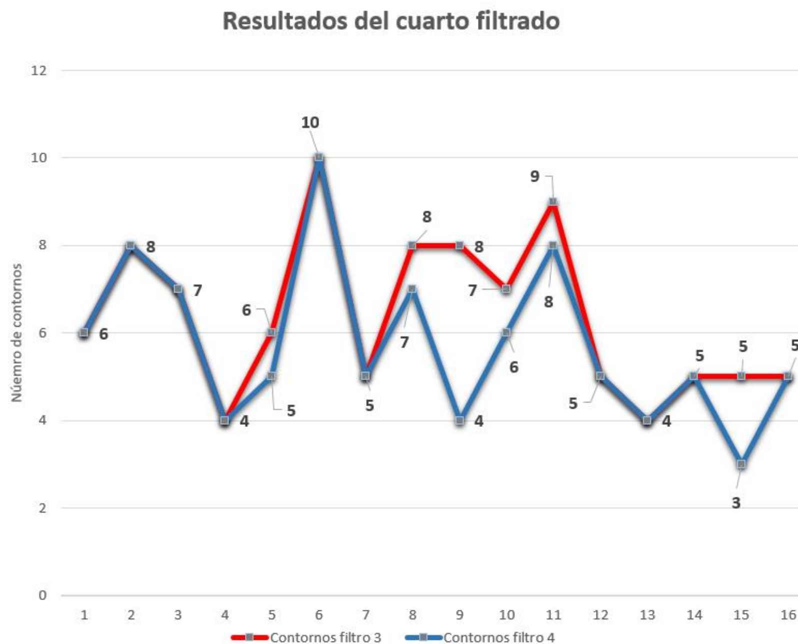


**Figura 59: Gráfica contornos tercer filtro vs contornos cuarto filtro**

En la Figura 59 puede verse una gráfica que compara el número de objetos en las 16 imágenes de prueba a la salida del tercer filtro, con los objetos a la salida del cuarto filtro. En este caso el promedio de reducción ha sido del 9.04% de reducción, debido a que el número de objetos que elimina este filtro es muy bajo, llegando a no eliminar ningún objeto en muchos casos. Pero, pese a su bajo porcentaje de reducción, este filtro es de suma importancia, debido a que reduce en gran medida el área de preselección en las imágenes en las que elimina objetos. En el siguiente subapartado se verá con más detalle porque este filtro es tan importante.

#### 4.1.1.10 *Agrupación de las Bounding Boxes clasificadas*

Este bloque del algoritmo se va a centrar en crea una *Bounding Box* que englobe todos los objetos que han sido clasificados como letras gracias a los filtrados explicados anteriormente. Para ellos será necesario utilizar los parámetros proporcionados por las *Bounding Boxes* de los objetos clasificados.

Para entender la forma en que se obtiene la *Bounding Box* global es necesario saber para la creación de cualquier *Bounding Box*, así como de cualquier rectángulo, son necesarios 4 parámetros, que son: las coordenadas X e Y del punto superior izquierdo del rectángulo y la altura y anchura del mismo. Una vez conocida esta información nuestro objetivo será obtener estos parámetros mediante las características de las *Bounding Boxes* individuales.

Para obtener la coordenada X del punto superior izquierdo de la *Bounding Box* global, deberemos de hallar la menor coordenada X (la que se encuentre más a la izquierda) de entre los diferentes puntos superiores izquierdos de las *Bounding Boxes* individuales de cada objeto. Del mismo modo, para obtener la coordenada Y del punto superior izquierdo de la *Bounding Box* global, será necesario encontrar la menor coordenada Y (más hacia arriba) de entre todos los puntos superiores izquierdos de las *Bounding Boxes* individuales.

Una vez obtenidas las coordenadas del punto superior izquierdo de la *Bounding Box* global, la anchura y altura de la misma, serán calculadas obteniendo la distancia entre dicho punto y los puntos de los objetos más alejados en X e Y respectivamente.

Los resultados de aplicar el método explicado en los párrafos anteriores pueden verse en la Figura 61. Como se puede observar la letra d no ha sido completamente incluida, debido a que no estaba entre los objetos clasificados como letras. En el siguiente subapartado se explicará cómo solucionar este tipo de problemas.

**Figura 60: Bounding Boxes sobre la imagen de entrada después del cuarto filtro**

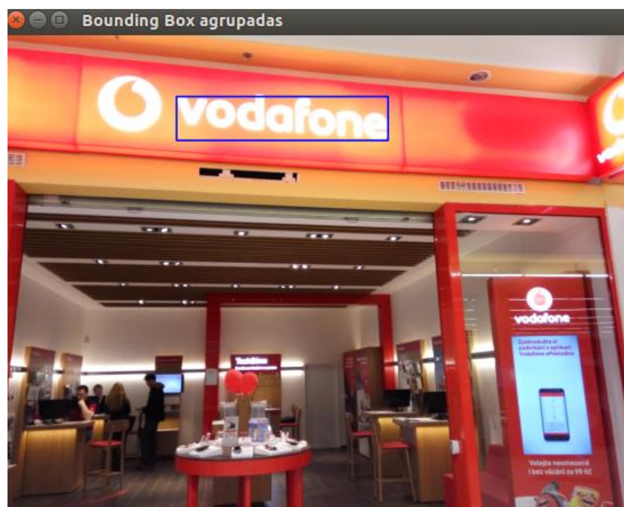


**Figura 61: Bounding Box global sobre la imagen de prueba "Vodafone"**

#### 4.1.1.11 *Agrandamiento de la Bounding Box global*

Como se ha comentado en el apartado anterior, cuando hay letras dentro de la palabra detectada que no se han clasificado correctamente, es muy posible que la *Bounding Box* global no las abarque apropiadamente. Para solucionar este problema, se ha decidido implementar un bloque de agrandamiento de la *Bounding Box* global.

Aumentar el tamaño de una *Bounding Box* es una tarea sencilla si se tiene que cuenta la información explicada anteriormente acerca de los parámetros que la forman, ya que lo único que se debe hacer es desplazar el punto superior izquierdo más a la derecha y hacia arriba cuanto más grande se quiera hacer el rectángulo, así como incrementar su anchura y altura de forma correspondiente.

En el presente algoritmo se ha decidido incrementar la anchura en un 50% y la altura en un 20%. Para que el crecimiento sea simétrico con respecto a los ejes de la *Bounding Box*, es decir, que crezca lo mismo hacia arriba que hacia abajo (10% en cada dirección), y por otra parte lo mismo hacia la izquierda que hacia la derecha (25% en cada dirección), ha sido necesario desplazar el punto superior izquierdo un 25% a la izquierda y un 10% hacia arriba, así como incrementar la anchura y la altura en un 50% y un 20% respectivamente. Los resultados de aplicar dicho incremento a la *Bounding Box* se pueden ver la Figura 63.

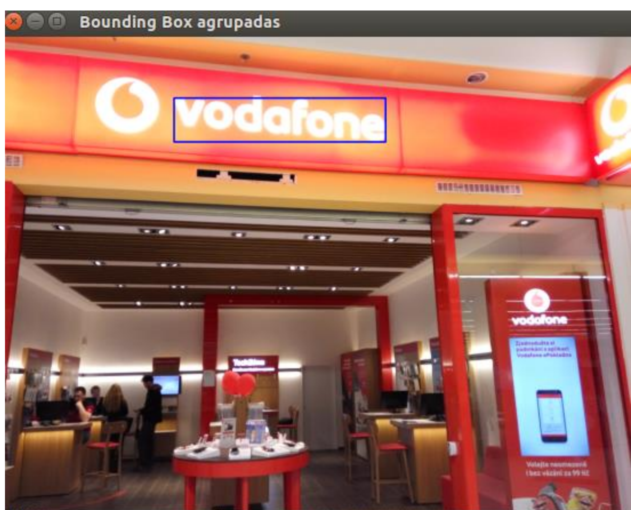


**Figura 62: Bounding Box global sobre la imagen de prueba "Vodafone"**

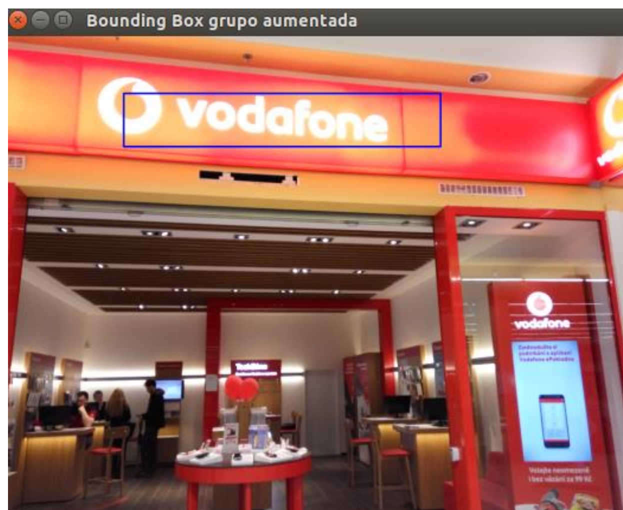


**Figura 63: Bounding Box global aumentada sobre la imagen de prueba "Vodafone"**

Como se puede apreciar en la Figura 63 ahora todas las letras de la palabra "Vodafone" están completamente incluidas en la *Bounding Box* global, y en el caso de que alguna letra de los extremos de la palabra no se hubiese clasificado correctamente, también se hubiese incluido, como consecuencia del mayor aumento en el eje X.

Con este bloque se da por concluido el algoritmo de predetección, debido a que el último paso del mismo, será pasarle al algoritmo de detección y reconocimiento un recorte de la información contenida dentro de la *Bounding Box* global aumentada, haciendo que dicho algoritmo de detección y reconocimiento solo tenga que procesar una pequeña parte de la imagen original, y, por tanto, acelerando la velocidad de obtención de respuesta.

**Figura 64: Output del algoritmo de predetección**

### 4.1.2 *Algoritmo de detección y reconocimiento de texto*

El algoritmo que va a ser descrito en el siguiente subapartado está formado por dos partes esenciales. El primer paso a realizar es la detección de candidatos a texto, los cuales serán encontrados en la imagen como regiones ER (Extremal Regions, explicadas en el punto 2.1.2 de esta memoria), para posteriormente utilizar un algoritmo OCR (Optical Character Recognition) para identificar que palabra o palabras se han detectado, en caso de que las regiones ER detectadas lo sean. A lo largo de los siguientes subapartados se van a detallar los pasos e este algoritmo con una mayor profundidad.

#### *4.1.2.1 Obtención de regiones ER (Extremal Regions)*

El primer paso del algoritmo de detección y reconocimiento de texto, como se ha comentado anteriormente será la obtención de las regiones ER, para ello la primera decisión que se debe tomar es cuantas características diferentes de la imagen se quieren analizar para hallar dichas regiones. Un mayor número de características analizadas proporcionará una mayor precisión en la detección, pero, por contrapartida, también conllevará un incremento significativo del coste computacional. Como el objetivo final del algoritmo que se está desarrollando es ser implementado en plataformas móviles, se debe intentar reducir al máximo el coste computacional del mismo, y por ello, se deberán reducir al máximo el número de características analizadas sin que la precisión se vea afectada sobremanera.

Teniendo en cuenta la información explicada en el párrafo anterior, y tras probar el estudio de diferentes características de la imagen para obtener regiones ER, se ha decidido que el algoritmo propuesto solo estudie 2 imágenes de características de la imagen de entrada, las cuales van a ser: la imagen original en escala de grises y el negativo de la misma, las cuales pueden ser observadas en las Figuras 65 y 66 respectivamente. Este método combina a la vez un buen desempeño a nivel temporal, así como a nivel de precisión en la detección.

**Figura 65: Imagen de grises de la imagen de entrada**

**Figura 66: Negativo de la imagen de grises de la imagen de entrada**

Una vez obtenidas estas dos imágenes de características, se va proceder a obtener las regiones ER contenidas en ellas mediante el método de umbralización explicado en el punto 2.1.2 de la presente memoria. Una vez obtenidas las regiones ER se van a utilizar 2 filtrados para descartar las regiones ER que no puedan ser letras.

El primer filtro será menos costoso computacionalmente, debido a que estudiará características de las regiones sencillas de calcular, ya que antes del primer filtro puede haber un gran número de regiones ER localizadas. El segundo filtro será computacionalmente más costoso debido a la complejidad de las características analizadas en el mismo. Los parámetros utilizados en los dos filtros van a ser detalladas a continuación.

Las características de las regiones ER analizadas en el primer filtro son las siguientes:

**Area:** Se analiza el área de las regiones (número de píxeles) para eliminar las que tengas un tamaño demasiado grande o pequeño.

**Bounding Box:** En este caso las *Bounding Boxes* de las áreas serán obtenidas a partir del punto superior izquierdo y del punto inferior derecho (en lugar del método utilizado anteriormente), debido a que en el algoritmo de Neumann y Matas para la obtención de regiones ER y su posterior clasificación como texto (el algoritmo utilizado para el bloque de detección) se utiliza dicho procedimiento.

**Perímetro:** Se estudia el número de píxeles del contorno exterior de la región.

**Número de Euler:** El número de Euler es una característica topológica de las imágenes binarias que compara la diferencia entre el número de componentes conexas y el número de huecos en dicha imagen.

**Cruces Horizontales c:** Para el cálculo de los cruces horizontales se crea un vector (de longitud h) con el número de transiciones entre pixeles pertenecientes y no pertenecientes a la región en una determinada fila i.

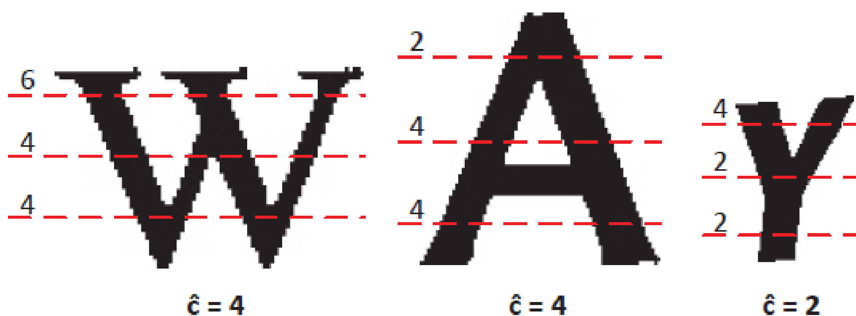


**Figura 67: Ejemplo de cruces horizontales en diferentes letras**

Las características de las regiones ER analizadas en el primer filtro son las siguientes:

**Relación área/hueco:** Se estudia la relación entre el número de píxeles de los huecos de región y el número de píxeles de la propia región.

**Relación de envolvente convexa:** En matemáticas se define la envolvente convexa de un conjunto de puntos X de dimensión n como la intersección de todos los conjuntos convexos que contienen a X. En el caso particular de puntos en un plano (como en las regiones ER que analizamos), si no todos los puntos están alineados, entonces su envolvente convexa corresponde a un polígono convexo cuyos vértices son algunos de los puntos del conjunto inicial de puntos. La relación de envolvente convexa es, por tanto, la relación entre el área de la envolvente convexa y el área de la región.

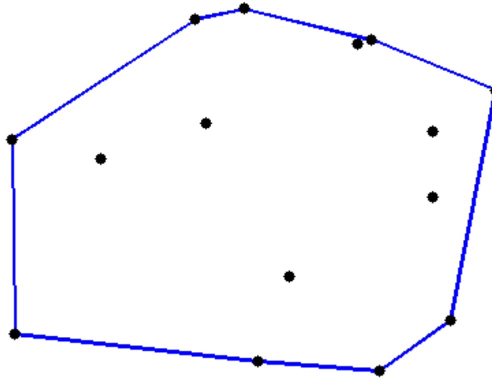

**Figura 68: Envolvente convexa de 15 puntos en un plano**

**Número de puntos de inflexión en el contorno exterior k:** Se analiza el número de cambios entre forma cóncava o convexa a lo largo del contorno exterior de la imagen. Los caracteres suelen tener un número limitado de puntos de inflexión, que suele ser menor a 10 ($k < 10$), mientras que las regiones que tienen un contenido no textual suelen presentar un número mayor de puntos de inflexión. En la Figura 69 podemos observar un ejemplo gráfico de dicho parámetro en diferentes tipos de regiones.

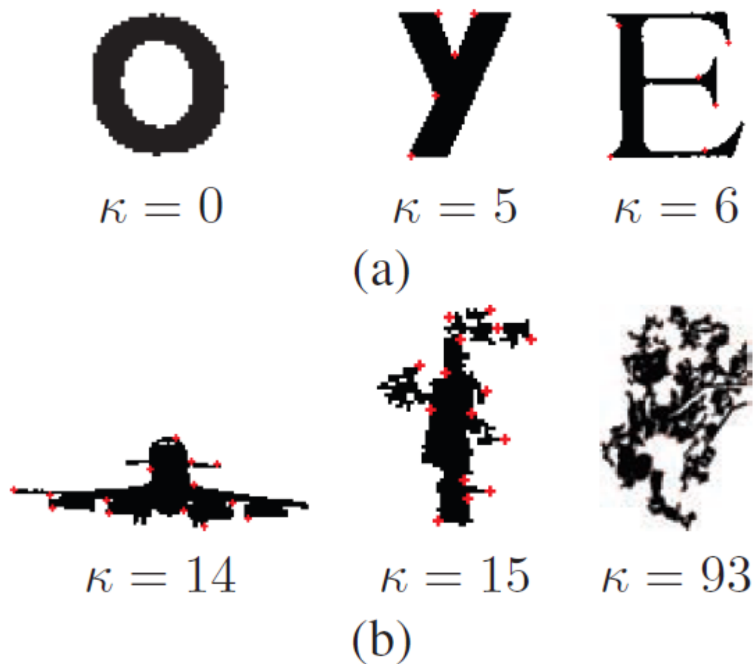


**Figura 69: Número de puntos de inflexión en el contorno letras y de otro tipo de regiones.**

A nivel de código, estos dos filtrados serán implementados a partir de dos clasificadores secuenciales entrenados que serán introducidos al algoritmo mediante dos ficheros en formato XML gracias a las funciones “loadClassifierNM1” y “loadClassifierNM2”. Una vez introducidos sendos clasificadores, serán aplicados individualmente a cada una de las imágenes de características que se hayan creado, en nuestro caso las imágenes representadas en las Figuras 65 y 66 mediante la función de la librería OpenCV “run”. Este proceso es programado para que su ejecución se realice en paralelo, debido a que el tiempo de ejecución de esta parte del algoritmo es determinante en la velocidad del mismo, debido a que uno de los procesos implementados con mayor coste computacional.

El resultado de aplicar el método de obtención de regiones ER con su posterior proceso de filtrado a las dos imágenes de características que se le pasan como entrada al algoritmo de detección de texto es el que se puede observar en la Figura 70.

**Figura 70: Regiones ER candidatas a texto detectadas en las diferentes imágenes de características.**

Como se puede observar hay tres niveles de intensidad de color en la imagen resultado de la Figura 70, esto es debido a que el nivel 0 (color negro en la imagen) representa la ausencia de regiones detectadas, el nivel 1 (color gris en la imagen) representa las regiones detectadas en la primera imagen de características, y por último el nivel 2 (color blanco en la imagen) representa las regiones detectadas en la segunda imagen de características. Estas regiones deberán ser analizadas y convenientemente agrupadas en el posterior bloque del algoritmo.

#### *4.1.2.2 Agrupación de regiones ER (Extremal Regions)*

Una vez la obtención de regiones ER y su posterior filtrado son llevados a cabo, es necesario realizar una agrupación de regiones candidatas en bloques de mayor nivel (palabras o líneas de texto). La librería OpenCV implementa 2 métodos para realizar dicha agrupación, uno de ellos está creado para agrupar regiones en horizontal, por otra parte, el otro método está diseñado para agruparlas en cualquier orientación. Como es de esperar, el método de agrupación omnidireccional requiere de un mayor coste computacional, por tanto, se ha decidido implementar el método de agrupación horizontal. En las Figuras 71 y 72 pueden verse los dos grupos de regiones encontrados por dicho método de agrupación de regiones cunado la imagen de entrada es la foto de prueba “Vodafone”.

**Figura 71: Primer grupo detectado**

**Figura 72: Segundo grupo detectado**

Como se puede observar en la Figura 71 el texto se ha detectado correctamente, sin embargo, en la figura 72, las regiones detectadas no son letras. El hecho de que se detecten regiones como la de la Figura 72 no supone ningún problema, ya que las regiones clasificadas serán procesadas por el algoritmo de reconocimiento de texto OCR, el cual se encargará de discernir si las regiones que tiene como entrada son letras o no.

#### *4.1.2.3 Reconocimiento OCR (Optical Character Recognition)*

Una vez agrupadas las regiones candidatas a texto nuestro objetivo será discernir si entre dichos grupos contienen texto o no, y que cuál es exactamente ese texto, en caso de que lo haya. Para llevar a cabo esta tarea, será necesario un motor OCR.

La libraría OpenCV nos brinda la posibilidad de utilizar el motor OCR de código libre Tesseract, el cual está actualmente desarrollado por Google y distribuido bajo la licencia Apache, versión 2.0. Tesseract está considera como uno de los motores OCR con mayor precisión disponibles actualmente, por tanto, se ha decidido implementarlo en el algoritmo que se está desarrollando.

Para obtener resultados más precisos con Tesseract será conveniente que se les añadan bordes negros a las regiones agrupadas, debido a que añadiendo dichos bordes se obtiene un mayor índice de reconocimientos correctos. Para añadir dichos bordes se utilizará la función OpenCV "copyMakeBorder", la cual nos permitirá añadir un borde del grosor deseado a las imágenes que contienen las regiones agrupadas. Para el presente algoritmo se ha decidido que un contorno de 15 píxeles es suficiente para alcanzar nuestro objetivo. En las Figuras 73 y 74 pueden verse los resultados de añadir bordes de 15 píxeles a las imágenes de entrada.

**Figura 73: Primer grupo detectado con bordes**

**Figura 74: Segundo grupo detectado con bordes**

Una vez se han agrandado las imágenes que contienen a los grupos de regiones ER candidatas a texto, el siguiente paso será utilizar Tesseract para reconocer el texto que se encuentre en las diferentes regiones, en caso de que lo haya.

Cuando Tesseract termina su ejecución se obtienen como parámetros de salida: un vector de strings con las palabras individuales que se han localizado en una región determinada, así como un valor de confianza de la detección para cada una de ellas, las *Bounding Boxes* de las mismas sobre la imagen de regiones con bordes añadidos, y finalmente, un string con el texto que han compuesto las palabras individuales.

De los parámetros mencionados anteriormente, el que va a resultar más interesante para el propósito de la aplicación que se pretende crear es el primero, debido a que la utilidad que se le pretende dar a la misma es leer nombres de tienda en un centro comercial, es decir grupos de palabras muy reducidos (entre 1 y 3 aproximadamente), por tanto, es más conveniente analizarlas de manera individual en lugar de como un texto conjunto.

Una vez obtenida la salida de Tesseract se hace pasar a las palabras detectadas por un filtro de características para descartar detecciones espurias. Este filtro estudia las siguientes características:

1. Si la palabra tiene menos de dos letras es descartada
2. Si el índice de confianza en la detección de la palabra es menor a 51 es descartada
3. Si la palabra tiene 2 letras y estas son iguales es descartada.
4. Si la palabra es menor de 4 letras y su nivel de confianza en la detección es menor de 60 es descartada
5. Si la palabra está repetida, es decir, si se ha detectado con anterioridad en un determinado grupo de regiones, es descartada.

La palabra o palabras que superen el filtro serán consideradas como válidas, siendo estas el resultado final de nuestro algoritmo. Para obtener el resultado de forma visual se mostrará por pantalla una imagen con cada una de las palabras detectadas recuadrada por su Bouding Box y se imprimirán por pantalla la interpretación de Tesseract de las mimas.

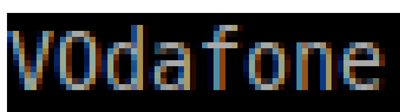


**Figura 75: Resultado del reconocimiento impresa en la consola de comandos**

**Figura 76: Bounding Box de la palabra detectada**

Como se puede apreciar en la Figura 76, la palabra se ha detectado correctamente, es decir, se ha encontrado su posición en la imagen, pero sin embargo el reconocimiento no ha sido perfecto, debido a que tanto la letra V como la O han sido detectados como letras mayúsculas en lugar de minúsculas, pero este error no afecta a nuestro propósito, es decir, que la persona invidente sea capaz de identificar el texto de la imagen, por tanto, el resultado del algoritmo es considerado correcto.

## 4.2 Tiempo de ejecución del algoritmo en C++

A lo largo de este apartado se van a comparar los tiempos de ejecución del algoritmo de detección y reconocimiento de texto que se ha analizado a lo largo de la presente memoria cuando se utiliza el algoritmo de predetección creado y cuando no se utiliza, con el objetivo de estimar si su implementación consigue reducir el tiempo de ejecución final de forma correcta, o si por el contrario es contraproducente utilizarlo.

Para estimar dicho efecto sobre un ejemplo concreto, se van a analizar los tiempos de ejecución de los bloques más importantes del algoritmo de detección y reconocimiento de texto, así como el tiempo de ejecución total del mismo sobre la imagen de prueba "Vodafone".

```
Tiempo lectura img = 124.438
Tiempo de detección de regiones = 206.082
Tiempo de agrupación = 36.5127
Tiempo de inicialización de OCR = 110.53
Tiempo algoritmo = 448.593
```

**Figura 77: Tiempos de ejecución sin predetección**

Es importante resaltar que los valores temporales están expresados en milisegundos, además, cabe destacar que el tiempo de lectura de imagen es tan abultado debido a que las imágenes de entrada del algoritmo son imágenes con muy alta resolución (4160x3120 píxeles), por este motivo no se ha tenido en cuenta para el cálculo del tiempo final del algoritmo, debido a que se utilice algoritmo de predetección o no, es te valor no variará.

Como se puede apreciar en la Figura 77, el bloque que más tiempo de ejecución necesita es la detección de regiones, por tanto, para reducir dicho tiempo será necesario disminuir el área de búsqueda en a que buscar dichas regiones, tarea que consigue con éxito el algoritmo de predetección. Sin embargo, se tendrá que estudiar si desempeña esta tarea en dentro de un límite de tiempo adecuado.

```
Tiempo lectura img = 106.378
Tiempo del algoritmo Canny detector = 7.72287
Tiempo de cálculo de contornos = 3.67185
Tiempo bboxes = 0.812139
Tiempo total = 13.5201
Tiempo de detección de regiones = 13.8847
Tiempo de agrupación = 10.4543
Tiempo de inicialización de OCR = 112.478
Tiempo algoritmo = 173.788
```

**Figura 78: Tiempos de ejecución con predetección**

Como puede observarse en la Figura 78 el tiempo total requerido por el algoritmo de predetección (expresa en el parámetro Tiempo total) es de 13.52 ms, un tiempo mucho más que aceptable teniendo en cuenta el coste computacional del algoritmo de detección y reconocimiento. La inclusión de este preprocesado de la imagen permite que el tiempo de detección de regiones se reduzca de 206.082 ms a 13.99 ms, es decir, acelera más de 10 veces el proceso más crítico del algoritmo de detección y reconocimiento, resultando como tiempo final del mismo 173.788 ms, que es una reducción muy significativa en comparación con los 448.593 ms iniciales (2.58 veces más rápido).

Al producirse una disminución del número de regiones encontradas, también se produce una reducción muy notable en el tiempo de agrupación de las mismas (más de 3 veces más rápido), sin embargo, el tiempo de inicialización el motor OCR se mantiene, debido a que este no depende del tamaño de la imagen de entrada.

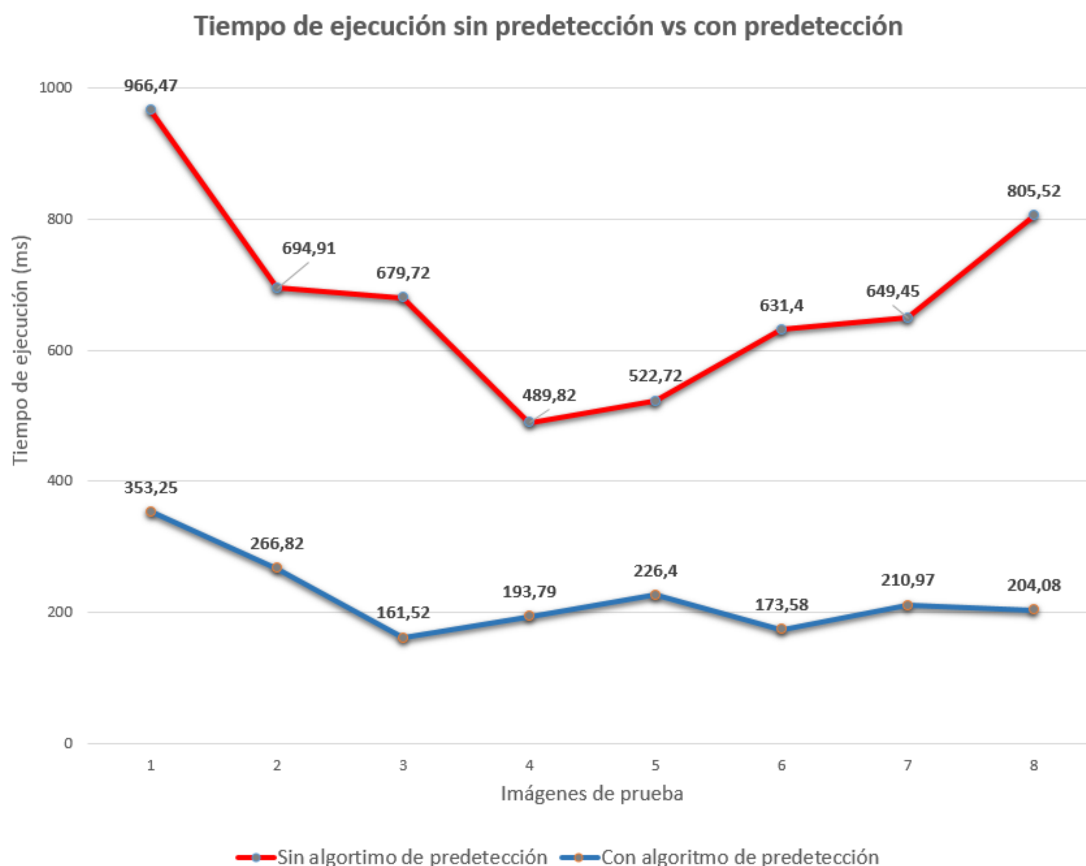


**Figura 79: Gráfica de tiempos de ejecución sin predetección vs con predetección**

Como se puede apreciar en la Figura 79 se han comparado las ejecuciones temporales de las 8 imágenes de prueba con mejor resultado en el reconocimiento de texto sin utilizar el algoritmo de detección y utilizándolo, y las conclusiones han sido que en todos los casos se produce una mejora muy significativa en la velocidad de ejecución del algoritmo al utilizar la predetección. En algunos casos se han llegado a alcanzar ejecuciones hasta 4 veces más rápidas, siendo el peor de los casos una multiplicación por 2 de la velocidad. El factor de aceleración promedio en las imágenes de prueba es de 3.13, obteniendo por tanto una media del 67% de reducción del tiempo de ejecución.

## 4.3 Desarrollo en Android

A lo largo de este apartado se va a detallar el estudio de la programación que se ha llevado a cabo para intentar desarrollar la aplicación móvil propuesta en la presente memoria. Es importante resaltar que el elemento más importante del sistema Android para implementar dicha aplicación es la compilación nativa, que como se ha detallado en secciones capítulos anteriores de esta memoria, es la posibilidad de implementar código en C y C++ en aplicaciones Android.

La compilación nativa tiene un papal tan significativo para el desarrollo de nuestra aplicación debido a que la totalidad del algoritmo desarrollado está escrito en C++, además, el código compilado en C++ para Android ofrece una velocidad de ejecución significativamente mayor que el código escrito en Java. Por tanto, la compilación nativa facilita la implementación del código desarrollado debido a que no es necesario traducirlo a Java y además permite que la ejecución del mismo sea más rápida. A lo largo del siguiente subapartado se detallarán los componentes más relevantes de la compilación nativa.

### *4.3.1 Compilación Nativa con Android NDK (Native Development Kit)*

El Android NDK (Native Development Kit) permite a los desarrolladores reutilizar código escrito en C/C++ introduciéndolo en las aplicaciones a través de JNI (Java Native Interface). El NDK hace que la ejecución de la aplicación sea en cierto modo más rápida, ya que pasará a ejecutarse directamente en el procesador y no es interpretado por una máquina virtual.

Las aplicaciones que suelen utilizar el NDK son aquellas que harán un uso intensivo de la CPU, como por ejemplo motores de videojuegos, procesado de señal y de imagen, como el caso que nos ocupa. En general, se trata de optimizar al máximo costosas operaciones matemáticas, como, por ejemplo, la detección ER.

#### *4.3.1.1 JNI (Java Native Interface)*

Java Native Interface (JNI) es un mecanismo que nos permite interactuar con aplicaciones nativas desde un programa escrito en Java. Las aplicaciones nativas son bibliotecas o funciones escritas en lenguajes como C, C++ y ensamblador para un sistema operativo donde se ejecuta la JVM (Java Virtual Machine).

Los métodos nativos se implementan por separado en archivos .c o .cpp, por los cuales se generará una biblioteca de enlace dinámico.

Una biblioteca de enlace dinámico es un fichero con código ejecutable el cual se carga en tiempo de ejecución bajo demanda de un programa por parte del sistema operativo. Aportan una gran ventaja con respecto al aprovechamiento de la memoria del sistema, dado que gran parte del código ejecutable puede encontrarse en bibliotecas, pudiendo así ser compartido entre distintas aplicaciones. Por supuesto dependen de cada sistema operativo, siendo su tratamiento y manejo distintos.

#### *4.3.1.2 Métodos Nativos*

Las clases, métodos y variables pueden ser declarados en Java como públicos, privados, entre otros tipos, pero gracias a JNI es posible trabajar con un modificador llamado "native", el cual indica a la aplicación que un determinado método está escrito en lenguaje nativo. Como por ejemplo el que se puede ver el Figura 80.

```
native int suma(int numero1, int numero2);
```

**Figura 80: Ejemplo de método nativo**

Con el modificador “native” indicamos que el método suma está escrito en C o C++, y que por tanto es necesario utilizar JNI para llegar a él.

#### *4.3.1.3 Librerías compartidas*

Las librerías compartidas son ficheros que se utilizan para enlazar ambos lenguajes: Java y C. Pueden ser de enlace estático o dinámico:

1. Enlace estático: extensión .o en sistemas UNIX y .lib en Windows.
2. Enlace dinámico: extensión .so en sistemas UNIX y .dll en Windows.

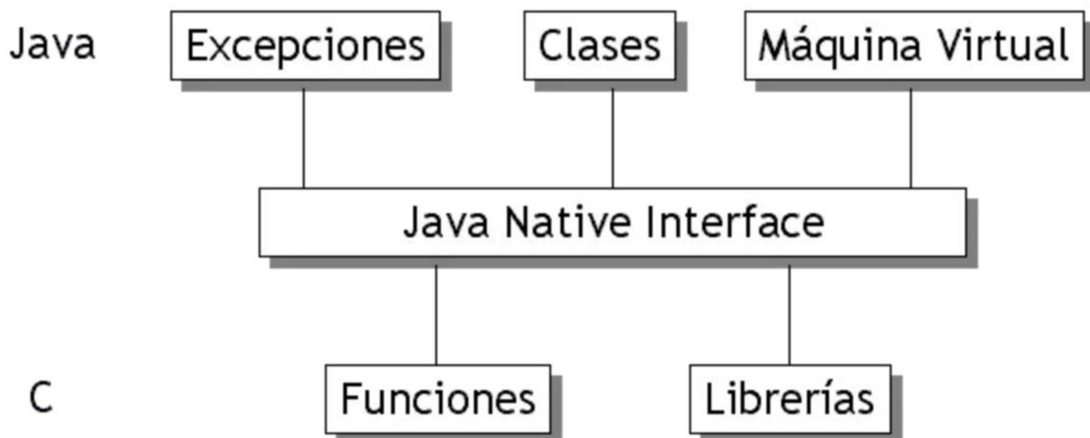


**Figura 81: Estructura entre Java y C al utilizar JNI**

#### *4.3.1.4 Importar librerías*

El proceso de importar una librería nativa al código Java es un proceso sencillo, que solo requiere del código mostrado en la siguiente Figura.

```
static {
	System.loadLibrary("miLibreria");
}
```

**Figura 82: Ejemplo del código necesario para importar una librería nativa**

Como se pude observar en la Figura 82, no es necesario indicar la extensión de la librería a la hora de importarla, solo será necesario su nombre.

#### *4.3.1.5 Código nativo*

El código nativo debe de ser compilado antes de ser ejecutado, es en este proceso donde el NDK tienen mayor relevancia. Las librerías escritas en C y C++ necesitan archivos llamados cabeceras, con extensión .h, los cuales deben ser generados utilizando la instrucción “javah” en la consola de comandos, debido a que las cabeceras utilizadas en el JNI tienen unos parámetros particulares. Una vez con las cabeceras y el código nativo terminados, será necesario compilarlo mediante la instrucción “ndk-build”. Es importante destacar que todo el código desarrollado en C o C++ debe estar almacenado en una carpeta llamada “jni”, que deberá ser creada en el interior de la aplicación que se quiere desarrollar.

#### *4.3.1.6 Carpeta jni*

La carpeta jni de la aplicación, como se ha comentado anteriormente, debe ser la que contenga todos los archivos relacionados con la compilación nativa, los cuales son los siguientes:

1. **Android.mk**: Este archivo describe los códigos fuente y librerías que van a ser necesarias a hora de compilar la aplicación. La sintaxis de este archivo permite agrupar las fuentes en módulos.
2. **Application.mk**: Es un pequeño GNU Makefile que define ciertas variables para la compilación, entre ellas, el tipo de procesador para el que se desea compilar el código nativo.
3. **Códigos fuente**: Estos archivos serán los que contengas las funciones de las librerías C o C++que se pretendan utilizar en la aplicación.
4. **Cabeceras**: Estos serán los archivos .h que contendrán las cabeceras de las funciones de las librerías utilizadas.

# Capítulo 5. Conclusiones

## 5.1 Conceptos aprendidos

A lo largo de la realización del trabajo de final de grado desarrollado a en la presente memoria se han aprendido una gran variedad de conceptos muy interesantes y útiles, predominantemente a cerca de Tratamiento de imágenes, entre los que destacan la instalación y utilización de la librería OpenCV, la cual proporciona unas herramientas fascinantes para trabajar con imágenes. En cuanto a los conocimientos de Tratamiento de imágenes aprendidos, también es muy importante destacar la recopilación, estudio y compresión de los diferentes de métodos de detección y localización de texto que hay en la actualidad, tarea que ha sido fundamental para llevar a cabo el presente proyecto.

Por otra parte también, para la realización del presente proyecto también se ha estudiado el lenguaje de programación Android, haciendo un especial hincapié en la compilación nativa mediante el uso del NDK (Native Development Kit) de dicha plataforma. El aprendizaje de estos conceptos ha supuesto un reto de especial dificultad debido a la total inexperiencia con Android y a que la utilización del NDK puede considerarse como una tarea de alta complejidad.

Por último, pero no menos importante, se considera que durante la realización de este trabajo se ha aprendido de forma práctica como gestionar un proyecto con una envergadura muy superior a los realizados anteriormente por el alumno, siendo muy importante para la consecución de dicha tarea fijar unos objetivos iniciales y realizar una planificación temporal a partir de los mismos.

## 5.2 Dificultades encontradas

A lo largo de este apartado se van a detallar las dificultades que se han encontrado a lo largo del desarrollo del presente proyecto y que no han podido ser subsanadas por falta de herramientas o por falta de tiempo.

### *5.2.1 Durante el testeo del algoritmo en C++*

A lo largo del presente subapartado se van a describir los problemas que se han encontrado a la hora de probar con una recopilación de 60 imágenes el algoritmo en C++ desarrollado para la consecución del presente trabajo.

#### *5.2.1.1 Elementos con características similares a letras*

Uno de los problemas que se han observado a la hora de analizar los outputs del algoritmo propuesto es que en ciertas existen ciertos elementos que presentan características espaciales similares a las letras. Esto va a tener como consecuencia que dichos objetos pasen los filtros de candidatos, pudiendo incluso ser clasificados en lugar del posible texto que se encuentre en la imagen. A continuación, se va a mostrar un ejemplo de dicho problema.

Figura 84: Entrada

Figura 83: Después del primer filtrado

Figura 85: Después del segundo filtrado

Figura 86: Salida

Como se puede apreciar en el ejemplo propuesto, el algoritmo de predetección ha seleccionado el elemento con propiedades de texto en lugar del texto que aparece en la imagen. Esta inconveniencia podría solucionarse si en el segundo filtrado se analiza más de un pico del histograma utilizado, pero como se ha explicado anteriormente esto tendría como consecuencia una ralentización del algoritmo final.

### *5.2.1.2 Imágenes tomadas con perspectiva*

Un problema que se ha producido en las imágenes más exigentes es que tanto el algoritmo de predetección como el de detección y reconocimiento presentan grandes dificultados a la hora de procesar correctamente las imágenes en las que aparece texto que no se encuentra en una perspectiva frontal. A continuación, se muestra un ejemplo de dicho fallo.

**Figura 87: Entrada**

**Figura 88: Después del primer filtrado**

**Figura 89: Después del segundo filtrado**

**Figura 90: Salida**

Como se puede observar en el ejemplo mostrado, cuando los textos aparecen en perspectiva en la imagen son deficientemente clasificadas, por tanto, trabajar para mejorar en este aspecto es muy importante, sobre todo para las personas invidentes que se pueden beneficia de esta clase de algoritmos, debido a que ellos no son capaces de discernir por ellos mismos si el texto está en perspectiva o no.

#### *5.2.1.3 Gran variedad tipográfica en los textos*

Cuando se pretende detectar y reconocer texto en escenas naturales, como, por ejemplo, en el caso particular de un centro comercial en el que se ha hecho hincapié a lo largo del presente proyecto, nos encontramos con una gran variedad de tipografías diferentes en los textos que aparecen en las imágenes. Esto es debido a que cada empresa utiliza su propio estilo para diferenciarse de la competencia.

Debido a esta gran variedad se producen dos problemas principales, el primero es que algunas de estas tipografías no son bien clasificadas por el algoritmo de predetección y el segundo es que muchas de estas tipografías no son reconocidas por el motor OCR. A continuación, se mostrarán 2 ejemplos de imágenes en las que se presentan estos problemas.

Figura 92: Entrada

Figura 91: Después del filtro 1

Figura 93: Después del filtro 2

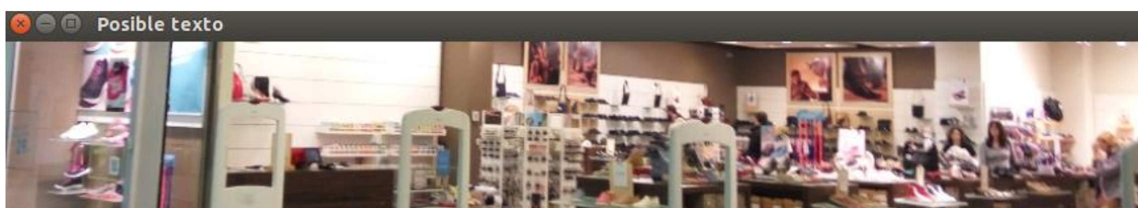


Figura 94: Salida

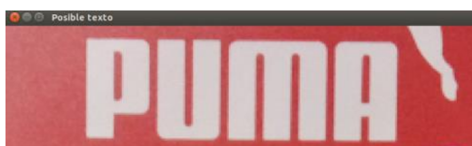


Figura 96: Entrada

Figura 95: Previo OCR

En el caso del primer ejemplo, la predetección se realiza de forma errónea debido a la inusual tipografía que podemos ver en la imagen, con todas las letras conectadas, pero en el segundo ejemplo, la palabra “PUMA” pasa con éxito el primer filtro de candidatos y las regiones ER de las letras que forman la palabra son agrupadas correctamente, el error en la detección se produce debido a que el motor OCR no reconoce ningún texto. Uno de los modos de mejorar este error es entrenando con redes neuronales un algoritmo de reconocimiento que sea capaz de reconocer todo tipo de tipografías.

#### *5.2.1.4 Destellos en las imágenes*

Por último, otro problema que se puede encontrar en la detección de textos en escenas naturales es la aparición de destellos en los caracteres que imposibiliten su detección. A continuación, se va a mostrar un ejemplo de dicho inconveniente.

**Figura 97: Entrada**

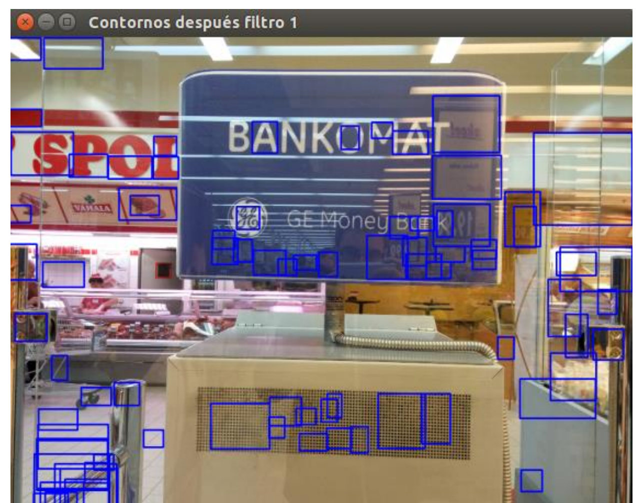


**Figura 98: Despues del filtro 1**

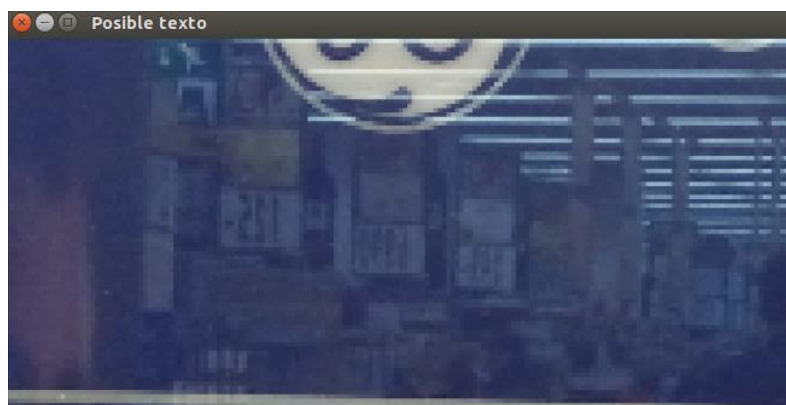


**Figura 99: Salida**

Como se puede apreciar en el ejemplo propuesto, los destellos han distorsionado un texto fácil de detectar (por sencillez tipográfica, perspectiva frontal y tamaño) haciendo que su detección sea imposible o muy complicada, en cualquier caso.

### 5.2.2 *Durante la implementación del algoritmo en Android*

#### *5.2.2.1 Problemas durante la compilación de los módulos extra de OpenCV*

Para conseguir el objetivo de implementar nuestro algoritmo en Android era necesario instalar la librería OpenCV para dispositivos Android, pero como se ha explicado anteriormente, las funciones de OpenCV para la detección de texto formar parte de los llamados módulos extra, por tanto, era necesario instalar estos módulos también.

Después de más de 2 semanas intentando compilar de forma correcta dichos módulos tanto en Ubuntu como en Windows no se consiguieron resultados satisfactorios, por lo que se decidió desistir en la instalación de los mismos por falta de tiempo, y por extensión no se alcanzó el objetivo de implementar el algoritmo propuesto en la plataforma Android.

## 5.3 Trabajo futuro

Como es lógico, el trabajo futuro que se va a plantear en el siguiente apartado está orientado a solucionar los problemas que se han presentado a lo largo del desarrollo del algoritmo y que no han podido ser solucionados por falta de herramientas o, principalmente, de tiempo.

Para empezar, la primera tarea en la que se trabajaría es en la implementación del algoritmo creado en la plataforma Android, debido a que era uno de los objetivos del presente proyecto y no se ha conseguido, por tanto, se considera muy importante la consecución de esta tarea. Para ello se debe investigar e insistir en la instalación de los módulos extra de la librería OpenCV para Android.

Una vez portado el algoritmo propuesto para Android sería conveniente intentar solucionar en la medida de lo posible los problemas comentados en los apartados anteriores. A continuación, se van a proponer posibles soluciones para algunos de ellos.

Para solucionar el problema de la predetección de texto cuando aparen en la imagen objetos con características similares, como se ha explicado anteriormente, es posible analizar diversos máximos del histograma de candidatos con su correspondiente aumento de coste computacional. También sería posible crear nuevos filtros de candidatos que fuesen más precisos a la hora de detectar textos.

En cuanto a la gran variedad de tipografías que aparecen en la detección de textos en imágenes naturales, sería conveniente entrenar un motor OCR mediante redes neuronales para que sea capaz de reconocer una mayor variedad de tipografías.

Para las imágenes tomadas en perspectiva podrían aplicarse transformaciones geométricas para obtener el texto en la perspectiva frontal que es conveniente para la detección, pero para ello sería necesario conocer la orientación del texto previamente.

# Capítulo 6. Bibliografía

[1] Moeslund, Thomas "Canny edge detection"

http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf [Online].

[2] Neumann, Lukas and Matas, Jiri "Real-Time Scene Text Localization and Recognition" http://cmp.felk.cvut.cz/~matas/papers/neumann-2012-rt_text-cvpr.pdf [Online].

[3] Neumann, Lukas and Matas, Jiri "A method for text localization and recognition in real-world images" http://cmp.felk.cvut.cz/~matas/papers/neumann-text-accv10.pdf [Online].

[4] Forss´en, P.; Lowe, D. and Wang, S-H "Region detectors" http://www.micc.unifi.it/delbimbo/wp-content/uploads/2011/03/slide_corso/A34%20MSER.pdf [Online].

[5] Adam Coates; Blake Carpenter and Carl Case "Text Detection and Character Recognition in Scene Images with Unsupervised Feature Learning"

http://www.iapr-tc11.org/archive/icdar2011/fileup/PDF/4520a440.pdf [Online].

[6] Neumann, Lukas and Matas, Jiri "Scene Text Localization and Recognition with Oriented Stroke Detection"

http://www.cv-foundation.org/openaccess/content_iccv_2013/papers/Neumann_Scene_Text_Localization_2013_ICCV_paper.pdf [Online].

[7] Xu-Cheng, Yi; Xuwang, Yin and Kaizhu, Huang "Robust Text Detection in Natural Scene Images" https://arxiv.org/pdf/1301.2628.pdf [Online].

[8] Grupo ADSLZONE "Ubuntu: descripción, descarga y características"

http://linuxzone.es/distribuciones-principales/ubuntu/ [Online].

[9] Google Inc. “Getting Started with the NDK”
https://developer.android.com/ndk/guides/index.html [Online].