Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Trabajo Fin de Máster		
Título	Implementación de una *Support Vector Machine* en RVC – CAL para imágenes hiperespectrales	
Autor	Daniel Madroñal Quintín	VºBº
Tutor	Eduardo Juárez Martínez	
Ponente		
Tribunal		
Presidente	César Sanz Álvaro	
Secretario	Juana María Gutiérrez Arriola	
Vocal	Matías Garrido González	
Fecha de lectura		
Calificación		

El Secretario:

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingeniería y Sistemas de Telecomunicación

TRABAJO DE INVESTIGACIÓN

Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Implementación de una *Support Vector Machine* en RVC – CAL para imágenes hiperespectrales

Daniel Madroñal Quintín

Junio de 2015

Agradecimientos

Tras un año me vuelvo a encontrar en la situación de agradecer a todas las personas que me han apoyado en todo momento. Y, de nuevo, gracias a ellos soy capaz de dar por finalizada otra etapa de mi vida con este Trabajo Fin de Máster.

Esta etapa que se cierra ahora ha sido completamente diferente a la anterior pero, gracias a ella, he conseguido conocer aspectos de la vida, del mundo y de mi mismo que me eran totalmente desconocidos. Por ello, quiero agradecer a todos y cada uno de los que han hecho posible que este año sea tan absolutamente inolvidable:

- *En primer lugar, agradecer a Eduardo Juárez, Maxime Pelcat y Luce Morin por hacer posible mi estancia de cuatro meses estudiando en Rennes y poder, de esta manera, ampliar mis conocimientos y seguir formándome como un verdadero ingeniero.*

- *A continuación, quiero agradecer a ese grupo de personas que hicieron que mi experiencia en Rennes fuera insuperable. Gracias a vosotros he conseguido formar una famille repartida por toda España. En especial, quiero destacar a la misu Lucía, a la sosia Ruiz, a los señores Artigas y Mateu y, sobre todo, a mi curry Laura, por demostrarme que no se necesita mucho tiempo para formar una amistad para toda la vida. Par chance, le monde est trop petit pour ne pas nous rencontrer, ¿no?*

- *Por supuesto, no puedo olvidarme de la familia con la que no comparto sangre: mis dos pequeñajas, Raquel y Mery. Gracias por demostrarme que, aunque estemos lejos, la amistad no entiende de distancias y que siempre podré contar con vosotras pase lo que pase y estemos donde estemos.*

- *Por último, no podía faltar mi eterno agradecimiento a mis padres y mi hermano por no dejar de confiar en mí ni dejar de apoyarme un segundo. En especial, quiero resaltar a esa mujercilla que no para de decir cosas como "pero yo de esto no sé…", "no sé si te será de ayuda…" o "no sé si estará bien o no…", sin ti absolutamente nada de esto habría sido posible. ¡Muchas gracias, Madre!*

Por último, quiero añadir que todos los nombrados aquí y los que no habéis sido nombrados pero sabéis que estáis ahí sois los que habéis hecho posible este trabajo, así que… ¡Va por vosotros! ¡Muchísimas gracias por todo!

Resumen

El análisis de imágenes hiperespectrales permite obtener información con una gran resolución espectral: cientos de bandas repartidas desde el espectro infrarrojo hasta el ultravioleta. El uso de dichas imágenes está teniendo un gran impacto en el campo de la medicina y, en concreto, destaca su utilización en la detección de distintos tipos de cáncer. Dentro de este campo, uno de los principales problemas que existen actualmente es el análisis de dichas imágenes en tiempo real ya que, debido al gran volumen de datos que componen estas imágenes, la capacidad de cómputo requerida es muy elevada. Una de las principales líneas de investigación acerca de la reducción de dicho tiempo de procesado se basa en la idea de repartir su análisis en diversos núcleos trabajando en paralelo.

En relación a esta línea de investigación, en el presente trabajo se desarrolla una librería para el lenguaje RVC – CAL – lenguaje que está especialmente pensado para aplicaciones multimedia y que permite realizar la paralelización de una manera intuitiva – donde se recogen las funciones necesarias para implementar el clasificador conocido como *Support Vector Machine* – SVM. Cabe mencionar que este trabajo complementa el realizado en [1] y [2] donde se desarrollaron las funciones necesarias para implementar una cadena de procesado que utiliza el método *unmixing* para procesar la imagen hiperespectral.

En concreto, este trabajo se encuentra dividido en varias partes. La primera de ellas expone razonadamente los motivos que han llevado a comenzar este Trabajo de Investigación y los objetivos que se pretenden conseguir con él. Tras esto, se hace un amplio estudio del estado del arte actual y, en él, se explican tanto las imágenes hiperespectrales como sus métodos de procesado y, en concreto, se detallará el método que utiliza el clasificador SVM. Una vez expuesta la base teórica, nos centraremos en la explicación del método seguido para convertir una versión en Matlab del clasificador SVM optimizado para analizar imágenes hiperespectrales; un punto importante en este apartado es que se desarrolla la versión secuencial del algoritmo y se asientan las bases para una futura paralelización del clasificador. Tras explicar el método utilizado, se exponen los resultados obtenidos primero comparando ambas versiones y, posteriormente, analizando por etapas la versión adaptada al lenguaje RVC – CAL. Por último, se aportan una serie de conclusiones obtenidas tras analizar las dos versiones del clasificador SVM en cuanto a bondad de resultados y tiempos de procesado y se proponen una serie de posibles líneas de actuación futuras relacionadas con dichos resultados.

Abstract

Hyperspectral imaging allows us to collect high resolution spectral information: hundred of bands covering from infrared to ultraviolet spectrum. These images have had strong repercussions in the medical field; in particular, we must highlight its use in cancer detection. In this field, the main problem we have to deal with is the real time analysis, because these images have a great data volume and they require a high computational power. One of the main research lines that deals with this problem is related with the analysis of these images using several cores working at the same time.

According to this investigation line, this document describes the development of a RVC – CAL library – this language has been widely used for working with multimedia applications and allows an optimized system parallelization –, which joins all the functions needed to implement the Support Vector Machine – SVM - classifier. This research complements the research conducted in [1] and [2] where the necessary functions to implement the unmixing method to analyze hyperspectral images were developed.

The document is divided in several chapters. The first of them introduces the motivation of the Master Thesis and the main objectives to achieve. After that, we study the state of the art of some technologies related with this work, like hyperspectral images, their processing methods and, concretely, the SVM classifier. Once we have exposed the theoretical bases, we will explain the followed methodology to translate a Matlab version of the SVM classifier optimized to process an hyperspectral image to RVC – CAL language; one of the most important issues in this chapter is that a sequential implementation is developed and the bases of a future parallelization of the SVM classifier are set. At this point, we will expose the results obtained in the comparative between versions and then, the results of the different steps that compose the SVM in its RVC – CAL version. Finally, we will extract some conclusions related with algorithm behavior and time processing. In the same way, we propose some future research lines according to the results obtained in this document.

Índice de contenidos

Índice de figuras

Índice de tablas

Índice de ecuaciones

Lista de acrónimos

- **ANN:** *Artificial Neural Network*
- **AMEE:** *Automatic Morphological Endmember Extraction*
- **CA:** *Cluster Analysis*
- **CITSEM:** *Centro de Investigación en Tecnologías de Software y Sistemas Multimedia*
- **DT:** *Decision Tree*
- **ERM:** *Empirical Risk Minimization*
- **FET:** *Future & Emerging Technologies*
- **FNR:** *False Negative Rate*
- **FPR:** *False Positive Rate*
- **GBF:** *Gaussian Radial Basis*
- **GDEM:** *Grupo de Diseño Electrónico y Microelectrónico*
- **HELICoiD:** *HypErspectraL Imaging Cancer Detection*
- **MPEG:** *Moving Pictures Experts Group*
- **ORCC:** *Open RVC – CAL Compiler*
- **PC:** *Precisión de Clases*
- **PG:** *Precisión Global*
- **PMC:** *Precisión Media de Clases*
- **PHMM:** *Pair Hidden Markov Moddel*
- **RBF:** *Radial Basis Function*
- **RGB:** *Red-Green-Blue*
- **RVC – CAL:** *Reconfigurable Video Coding – CAL Actor Language*
- **SRM:** *Structural Risk Minimization*
- **SVM:** *Support Vector Machine*
- **QP:** *Quadratic Programming*
- **ULPGC:** *Universidad de Las Palmas de Gran Canaria*

1 Introducción

1.1. Motivaciones

El estudio desarrollado en este Trabajo de Investigación se centra en el desarrollo de un nuevo método de implementación de cadenas de procesado de imágenes hiperespectrales. Este proyecto se enmarca dentro de una de las líneas de investigación del Grupo de Diseño Electrónico y Microelectrónico (GDEM) perteneciente al Centro de Investigación en Tecnologías de Software y Sistemas Multimedia para la Sostenibilidad (CITSEM) de la Universidad Politécnica de Madrid (UPM).

Esta línea de investigación se inscribe dentro del proyecto europeo HELICoiD[1] (de sus siglas en inglés, *HypErspectraL Imaging Cancer Detection*). Dicho proyecto comenzó en enero de 2014 y está enmarcado en el Séptimo Programa Marco de la Unión Europea, concretamente en el programa de Tecnologías Emergentes y Futuras (FET – *Future & Emerging Technologies*). En este proyecto de colaboración participan miembros de diversos países, tales como Francia, Reino Unido, Holanda y España y, a su vez, reúne dos hospitales, tres empresas y cuatro universidades, entre las que se incluye la UPM.

El objetivo del proyecto HELICoiD es estudiar la viabilidad de utilizar imágenes hiperespectrales para realizar una diferenciación entre tejido sano y tejido tumoroso en tiempo real durante una cirugía. El motivo por el que se utilizan dichas imágenes es la capacidad de extraer la firma espectral de cada tipo de tejido y, de esta manera, realizar la diferenciación.

En concreto, dentro de HELICoiD, el grupo GDEM lidera el paquete de trabajo relacionado con la selección de la plataforma hardware que se utilizará y la implementación de los algoritmos de la cadena de procesado en dicha plataforma; además, ayudará en el desarrollo y análisis de funcionamiento de los algoritmos. En este análisis, el principal trabajo del grupo será la detección de los posibles cuellos de botella que ralenticen el sistema y, en consecuencia, no permitan la consecución del análisis en tiempo real. En este campo el GDEM tiene una amplia experiencia, debido a su línea de investigación relacionada con la optimización del consumo de recursos mediante la monitorización de sistemas.

[1] http://helicoid.eu/

Por tanto, el presente Trabajo de Investigación se centrará en el desarrollo de un método que facilite la elaboración de distintos algoritmos relacionados con el procesado de imágenes hiperespectrales. En concreto, se pretende facilitar la monitorización de los distintos algoritmos y, a su vez, reducir la dificultad que supone la paralelización de los algoritmos desarrollados – ya que, presumiblemente, la plataforma hardware seleccionada será una plataforma multinúcleo.

En este sentido, cabe destacar que, para facilitar la realización de dicho paralelismo, se utilizará un lenguaje de flujo de datos denominado RVC – CAL. La principal característica de un lenguaje de este tipo es su estructura: bloques funcionales que intercambian datos entre sí.

1.2.Objetivos

El principal objetivo del presente trabajo es el desarrollo de un nuevo método que facilite la elaboración y posterior implementación de algoritmos propios del procesado de imágenes hiperespectrales en tiempo real. Para ello, se han estudiado los métodos de procesado de imágenes hiperespectrales y, posteriormente, se ha optado por incluir en la librería desarrollada en [1] y [2] las funciones necesarias para implementar el clasificador conocido como *Support Vector Machine* en RVC – CAL. Para alcanzar este objetivo global, se han fijado los siguientes objetivos específicos:

- ✓ Estudiar en detalle las imágenes hiperespectrales, sus métodos de análisis y los métodos utilizados actualmente para reducir el tiempo de procesado y aproximarse, en la medida de lo posible, al tiempo real.
- ✓ Estudiar en detalle el clasificador SVM y, en concreto, un modelo optimizado para el análisis de imágenes hiperespectrales codificado en Matlab proporcionado por la Universidad de las Palmas.
- ✓ Incluir en la librería desarrollada en [1] y [2] las funciones necesarias para implementar el clasificador SVM en lenguaje RVC – CAL.
- ✓ Replicar la implementación Matlab – secuencial – en lenguaje RVC – CAL para comprobar la viabilidad de desarrollar clasificadores SVM en dicho lenguaje.
- ✓ Realizar un estudio exhaustivo de la bondad del desarrollado en cuanto a lo que exactitud de resultados y tiempo de ejecución se refiere. Este estudio servirá para verificar el correcto funcionamiento de las nuevas funciones añadidas a la librería y comparar el rendimiento de ambas implementaciones.

2 Antecedentes

2.1. Estructura

A lo largo de este capítulo se introducirán los conceptos más importantes relacionados con el Trabajo de Investigación recogido en este documento. En este estudio, se describen detalladamente dichos conceptos y se estudiará su estado del arte resaltando los aspectos más significativos de cada uno de ellos.

En primer lugar, se definirá el concepto de imagen hiperespectral, aportando una definición formal y resaltando las características principales de este tipo de imágenes. Además, se introducirán sus dos métodos de procesamiento más extendidos y se explicarán las diferencias entre ambos.

Tras explicar los diferentes métodos existentes para analizar imágenes hiperespectrales, se define ampliamente el concepto de clasificador y, en concreto, el clasificador que se implementará durante este Trabajo de Investigación: el clasificador Máquina de Vectores Soporte o SVM (*Support Vector Machine*).

Una vez se han explicado tanto las imágenes hiperespectrales como las SVM, se realizará un pequeño estudio de las diferentes aplicaciones en las que se utilizan conjuntamente estas tecnologías. Teniendo en cuenta que, como se dijo en el capítulo 1, este proyecto se inscribe dentro de HELICoiD, las aplicaciones analizadas estarán principalmente relacionadas con el campo de la medicina y, en concreto, con la detección de cáncer.

Por último, para concluir este capítulo, se explicará el lenguaje de programación RVC – CAL. Este lenguaje será el utilizado durante el Trabajo de Investigación aquí reflejado y, durante esta explicación, se estudiarán sus principales características, su estructura, su sintaxis y, por último, se analizarán las distintas ventajas e inconvenientes de su utilización que afectan de manera directa a esta investigación.

Cabe destacar que este proyecto es una continuación de la investigación llevada a cabo en conjunto con Raquel Lazcano López, la cual está recogida en los documentos [1] y [2]. Debido a esto, no todos los apartados están descritos con el mismo nivel de detalle y, en consecuencia, esta investigación será referenciada con asiduidad a lo largo de este documento.

2.2.Imágenes hiperespectrales: definición y métodos de procesado

Durante el desarrollo de este Trabajo de Investigación, se pretende procesar una imagen hiperespectral. Por ello, a lo largo de este apartado, se definirá el concepto de imagen hiperespectral y, posteriormente, se explicarán los dos métodos más extendidos para realizar dicho procesado.

Una imagen hiperespectral se define como una imagen compuesta de alta resolución espacial y espectral. Lo que se traduce en la práctica como una imagen en la que, para un mismo píxel, no tenemos únicamente los tres valores de reflectancia (equivalentes a las bandas de color rojo, verde y azul) propios de una imagen habitual RGB (*Red-Green-Blue*).

En el caso de las imágenes hiperespectrales, cada píxel está formado por cientos de valores de reflectancia asociados a diferentes bandas repartidas por todo el espectro electromagnético; estas bandas pueden pertenecer al espectro visible, al infrarrojo o al ultravioleta – los dos últimos invisibles para el ojo humano. Esta idea queda plasmada gráficamente en la figura 2.1.

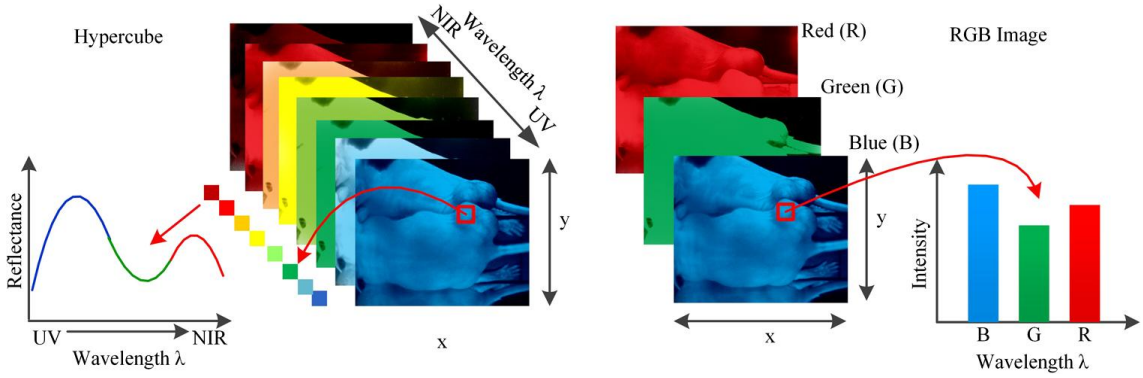

Figura 2.1. Comparativa: imagen hiperespectral (izq) e imagen RGB (der) [2]

La principal ventaja asociada a la utilización de este tipo de imágenes es la cantidad de información que obtenemos del espectro electromagnético invisible para el ojo humano. En contraposición, esta información extra conlleva que la cantidad de datos que tienen que ser procesados aumente drásticamente y, en consecuencia, el procesado de estas imágenes requiere un gran consumo de recursos.

Este tipo de imágenes, principalmente, se procesan utilizando dos caminos diferentes. Cada uno de estos caminos enfoca el análisis desde un punto de vista diferente pero ambos comparten la idea de sacar partido de la alta resolución espectral que éstas:

- En primer lugar, si tenemos en cuenta que cada imagen está formada por diferentes materiales mezclados entre sí, se analizará la imagen siguiendo la técnica denominada "desmezclado espectral" explicada detalladamente en [2]. En resumen, este procedimiento tiene por objetivo extraer las firmas espectrales propias de cada material – denominadas habitualmente *endmembers* – y estimar la distribución de cada *endmember* en la imagen. Este proceso está dividido en las siguientes cuatro etapas y puede verse gráficamente en la figura 2.2:

    1. *Estimación de endmembers*: etapa opcional en la que se estiman el número de *endmembers* que componen la imagen. Se considera opcional ya que es una etapa de configuración.

    2. *Reducción dimensional:* también es una etapa opcional cuyo objetivo es reducir el número de bandas de la imagen original reduciendo, de esta manera, el volumen de datos a procesar sin perder información relevante. Esta etapa es imprescindible cuando se quiere alcanzar el procesado en tiempo real.

    3. *Extracción de endmembers:* en esta etapa, se extraen las distintas firmas espectrales de cada elemento que compone la imagen. Estas firmas espectrales se corresponden con los valores de reflectancia frente a cada banda propios de cada elemento.

    4. *Estimación de abundancias:* durante esta etapa, se estima el grado de coincidencia de cada *endmember* con cada píxel de la imagen hiperespectral. Por tanto, se obtiene la distribución de cada elemento en la imagen.

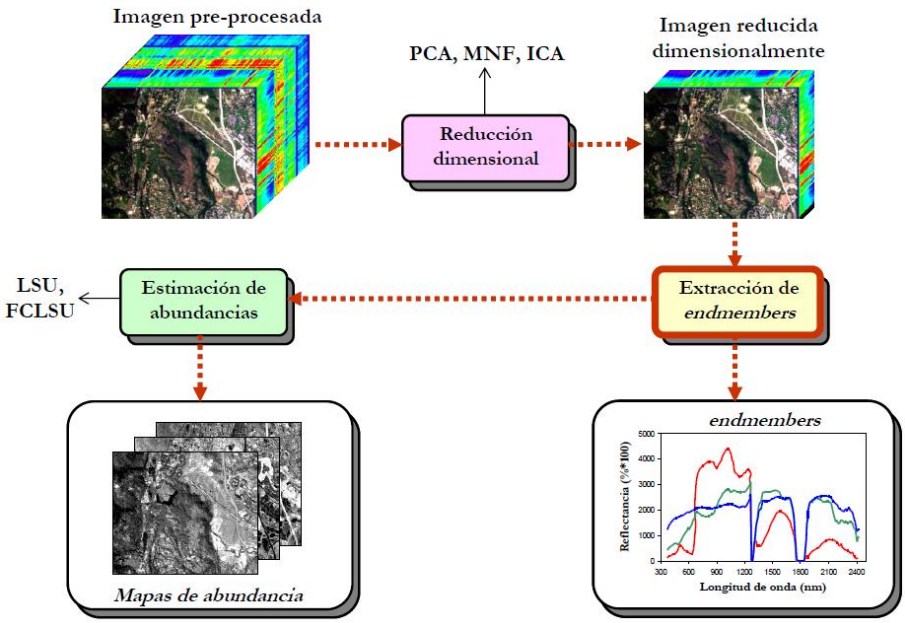

Figura 2.2. Cadena de procesado de imágenes hiperespectrales (Fuente: http://www.umbc.edu/)

- El segundo método de procesado de una imagen hiperespectral está basado en el uso de clasificadores. En este caso, se extraerán una serie de características diferenciadoras propias de cada material – en inglés, *features* – y, con ellas, se construirá un modelo mediante el cual el sistema será capaz de diferenciar (utilizando un mecanismo u otro, dependiendo del clasificador utilizado) un material de los demás [3]. En el caso de las imágenes hiperespectrales, las características extraídas serán las reflectancias de cada material para cada banda (longitud de onda).

  Hay que tener en cuenta que, si se requiere una alta velocidad de procesado, del mismo modo que ocurría en el método de procesado anterior, es necesario reducir el volumen de datos a analizar. Para ello, una solución habitualmente utilizada, es implementar la segunda de las etapas del primer procesamiento: *Reducción dimensional*. En este caso, se realizaría dicha reducción como etapa previa a la construcción del modelo de comparación del clasificador.

2.3.Clasificadores: *Support Vector Machine* (SVM)

Como se menciona en el capítulo 1 de este documento, el principal objetivo de esta investigación es implementar el clasificador conocido como Máquina de Vectores Soporte, del inglés, *Support Vector Machine* (SVM). Para ello, es necesario definir en primer lugar el concepto de clasificador, posteriormente, dar una definición formal de SVM y, finalmente, explicar su funcionamiento. Estos conceptos están recogidos a continuación en este apartado.

2.3.1. Concepto de clasificador

La idea de clasificar datos automáticamente pertenece a una rama relativamente actual de la ciencia de la computación relacionada con la capacidad de las máquinas de aprender (*machine learning*) y ser capaces de reconocer patrones (*pattern recognition*). Este concepto fue definido por Arthur Samuel en 1959 como "*el campo de estudio que otorga a los ordenadores la habilidad de aprender sin ser programados explícitamente*" [4].

El objetivo principal de estos clasificadores es ser capaz de generar un modelo de clasificación utilizando un set de muestras de prueba y realizando un entrenamiento y, tras esto, ser capaz de distinguir la clase a la que pertenece cada una de las muestras que se quieran clasificar.

Este modelo de clasificación se realiza mediante una etapa que se denomina entrenamiento supervisado, del inglés *supervised learning*. En esta fase se introduce al sistema una base de datos que ha sido previamente analizada y clasificada denominada set de entrenamiento. De esta manera, el clasificador será capaz de realizar una distinción entre clases utilizando los patrones extraídos durante dicho entrenamiento [5].

Una vez concluida la fase de entrenamiento, el clasificador debe ser capaz de clasificar correctamente nuevos datos de entrada denominados muestras de test, es decir, muestras que no se hayan utilizado durante el entrenamiento. En concreto, el objetivo es establecer una serie de parámetros de decisión que nos permitan realizar una distinción entre las distintas clases que se van a clasificar.

En la figura 2.3 se observa un ejemplo de clasificador básico, configurado para distinguir dos clases distintas. Dicha figura, está compuesta por una serie de puntos negros y puntos blancos, separados, en su mayoría, por una línea discontinua (la cual representa el parámetro de decisión establecido). También podemos observar que existen puntos mal clasificados, lo que demuestra que la aplicación de esta técnica tiene un margen de error en la clasificación.

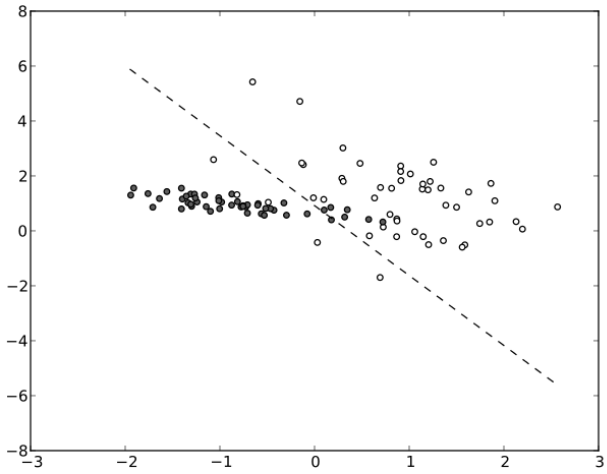

Figura 2.3. Ejemplo de clasificación lineal

La explicación de esta clasificación errónea es que los algoritmos utilizados para implementar dichos clasificadores tienen un margen de error ya que, como es evidente, las bases de datos de entrenamiento utilizadas son finitas y, en consecuencia, los patrones extraídos para cada clase son aproximados al ideal. Por otro lado, las muestras que se analizan con posterioridad, pueden estar compuestas por una mezcla de varias de las clases que se quieren analizar y, por tanto, no corresponderse directamente con uno de los modelos aprendidos[2].

A continuación, se ofrece un resumen de los principales métodos de clasificación utilizados actualmente y, para cada uno de ellos, se expone la idea básica en las que se basan.

- *Support Vector Machine* (SVM): este clasificador será explicado en profundidad en la próxima sección. Su entrenamiento consiste principalmente en la construcción de un hiperplano n-dimensional en el que sea posible realizar una separación lineal entre las distintas clases que se clasificarán posteriormente.

[2] http://en.wikipedia.org/wiki/Machine_learning

- *Decision Tree* (DT): este clasificador tiene en cuenta las entradas al sistema de una en una y, como observamos en la figura 2.4, avanza por el "árbol de decisión" hasta que alcanza un nodo que se corresponde con una de las clases aprendidas.

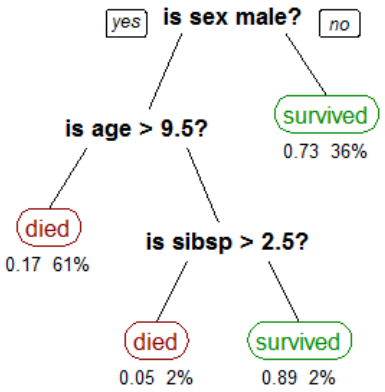

Figura 2.4. Ejemplo DT sobre los supervivientes del *Titanic*

- *Artificial Neural Network* (ANN): en este caso, a diferencia con el anterior, se tienen en cuenta todas las entradas en paralelo. Dichas entradas son introducidas en los distintos bloques *hidden* generadas durante la etapa de entrenamiento. Este procedimiento está reflejado en la figura 2.5 y, como se puede observar, se obtiene la clase asignada analizando las salidas de cada uno de los bloques *hidden*.

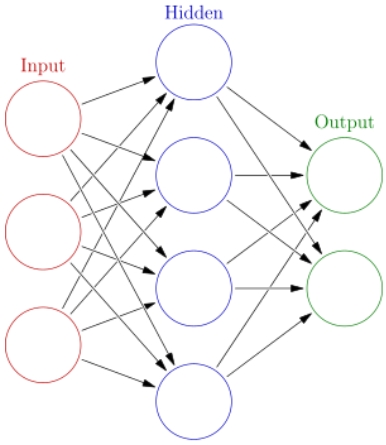

Figura 2.5. Esquema ANN

- *Cluster Analysis* (CA): el entrenamiento de este tipo de clasificadores está basado en el cálculo de una serie de *clusters.* Estos *clusters* son, en resumen, un conjunto de datos cercanos entre sí.

  Una vez representadas todas las muestras de entrenamiento, se calcula el *cluster* utilizando diferentes métodos. Los dos métodos más utilizados actualmente son: calcular el centro de gravedad de los puntos más cercanos de una zona de la representación o, en el segundo caso, tener en cuenta la densidad de muestras de entrenamiento en la representación y considerar como una clase cada conjunto de muestras agrupadas. Este último ejemplo puede observarse en la figura 2.6.

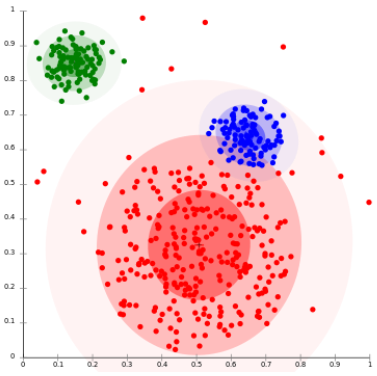

Figura 2.6. Ejemplo de CA

2.3.2. *Support Vector Machine* (SVM)

De cara a conocer a fondo el clasificador que se va a implementar durante el desarrollo de este Trabajo de Investigación, este apartado tratará en profundidad el clasificador SVM. Para ello, se aportará una definición formal de dicho clasificador y, tras esto, se expondrán su historia, su desarrollo y los principales parámetros implicados en su configuración.

Como se ha mencionado en la sección anterior, este clasificador se basa en la idea de separar de manera óptima un hiperplano n-dimensional (con n igual al número de *features* que hayamos extraído) en tantos subespacios como clases queramos diferenciar y, a su vez, otorgar a cada subespacio el máximo margen de seguridad posible, entendiendo como margen de seguridad la distancia existente entre la frontera de separación entre clases y el punto de entrenamiento más cercano a la misma [6]. Esta idea se recoge gráficamente en la figura 2.7 para el caso de un hiperplano de dos dimensiones y dos clases.

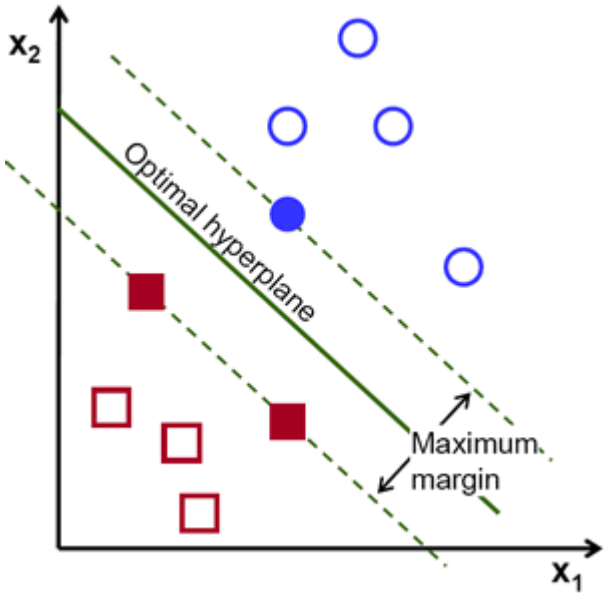

Figura 2.7. Ejemplo de SVM con márgenes de seguridad

El concepto lineal en el que se basa la máquina de vectores soporte fue elaborado por Vladimir N. Vapnik en el año 1963 mientras que, por otro lado, la versión no lineal de las SVM no surgió hasta 1992 cuando se sugirió la utilización de diferentes *kernels* a la hora de calcular el plano de separación entre clases.

Tal y como explica Marta Rojas en su Proyecto Fin de Carrera [7], el clasificador SVM surge de la combinación de cuatro ideas principales:

- La búsqueda de hiperplanos óptimos.
- La utilización de la convolución de producto escalar.
- La extensión de funciones lineales a no lineales.
- El concepto de margen de seguridad, del inglés *soft margin*, permite pequeños errores en los patrones de entrenamiento ignorando aquellos puntos que queden dentro de un rango de la frontera entre clases preestablecido.

En concreto, como se explica en [8], la búsqueda de hiperplanos óptimos y la posterior obtención de márgenes de seguridad, recibe el nombre de entrenamiento del clasificador.

En el caso del clasificador SVM para dos clases, dado un set de entrenamiento en el que para cada muestra se indica a cuál de las dos clases pertenece, el objetivo del entrenamiento es calcular un hiperplano que separe todos los puntos de cada clase con el máximo margen de separación posible. Este hiperplano dividirá un espacio con tantas dimensiones como características hayamos extraído y tendrá una forma característica dependiendo del *kernel* que se utilice. En la figura 2.8, se muestra un ejemplo gráfico del resultado que se obtiene al realizar el entrenamiento del clasificador SVM para el caso del *kernel* lineal para dos clases.

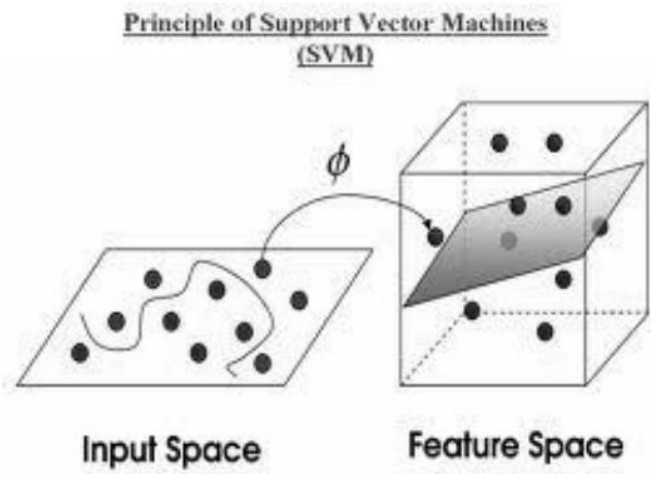

Figura 2.8. Ejemplo de hiperplano de un clasificador SVM [8]

En 1995, se introdujo el concepto de margen de seguridad. Esta idea proporciona la capacidad de considerar la región cercana al hiperplano de separación como una zona de incertidumbre. Los puntos de entrenamiento que se encuentren dentro de esta zona serán descartados y, posteriormente, si un punto a clasificar está incluido dentro de la misma, pasará a considerarse como no clasificable.

Por otro lado, a continuación se exponen las principales ventajas que se obtienen gracias a la utilización de este clasificador. Entre ellas, las tres más relevantes y que tienen un mayor impacto en el desarrollo de este Trabajo de Investigación son las siguientes:

- Gracias a la utilización del principio de Minimización de la Riqueza Estructural (*Structural Risk Minimization* – SRM), en lugar de utilizar el principio de Minimización de la Riqueza Empírica (*Empirical Risk Minimization* – ERM) [9], el clasificador SVM busca la consecución de una mejor generalización durante el entrenamiento y no la máxima reducción del número de errores durante dicha fase.

- La problemática de realizar la clasificación puede ser reducida a un problema de programación convexa (*convex quadratic programming* – QP) que, comparada con el método habitual, se resuelve de manera más sencilla.

- Por último, el clasificador SVM presenta una mayor robustez cuando utiliza grandes cantidades de datos tanto en entrenamiento como en clasificación.

Como se ha mencionado con anterioridad, este clasificador puede trabajar de modo lineal o no lineal. Para hacer esta diferenciación, el clasificador SVM se basa en la utilización de diferentes *kernel*, entendiendo por *kernel* la forma que tendrá la separación generada entre clases dentro del hiperplano. En la literatura referente a este concepto, destacan tres tipos de *kernel* cuyo uso está más extendido y son utilizados en la gran mayoría de los estudios realizados actualmente.

- El primer *kernel* que se desarrolló se denomina *kernel* lineal. Es el más básico y sencillo de todos y el que se utiliza habitualmente.
  Este *kernel* sigue la ecuación 2.1 y su funcionamiento se basa en el cálculo de la función que defina la línea, plano o hiperplano de grado n mediante el cual la separación entre clases del set de muestras sea óptima, es decir, que maximice la distancia entre la separación y los puntos de entrenamiento. Cabe

mencionar que durante el desarrollo de este Trabajo de Investigación se implementará una versión del clasificador SVM que utiliza este *kernel*.

$$\mathrm{k}(\mathrm{x},\mathrm{x}') = (x \cdot x')^d \qquad (2.1)$$

- En segundo lugar, nos encontramos con los *kernels* basados en *Radial Basis Function* o RBF. Éstos, de manera genérica, siguen la ecuación 2.2 y, en concreto, uno de los más utilizados es el GBF - *Gaussian Radial Basis function* – que sigue la variante 2.3 de la ecuación general. Un ejemplo gráfico de la diferencia que existe entre el *kernel* lineal y el presentado en este punto puede observarse en la figura 2.9.

$$k(x,x') = exp(-\gamma||x-x'||^2), \text{ para } \gamma > 0 \qquad (2.2)$$

$$k(x,x') = exp\left(\frac{||x-x'||^2}{2\sigma^2}\right) \qquad (2.3)$$

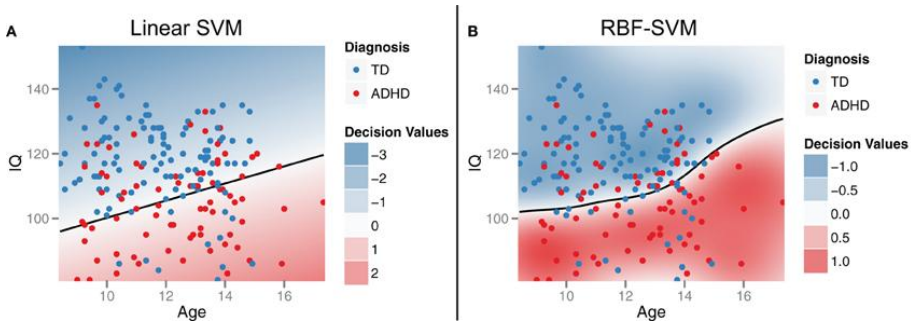

Figura 2.9. *Kernel* lineal (A) y *kernel* RBF (B)

- Por último, nos encontramos con un *kernel* denominado sigmoide o *Hyperbolic Tangent Kernel*. Dicho *kernel* sigue la ecuación 2.4 y, principalmente, se caracteriza por generar una separación con forma hiperbólica que, a su vez, es tangencial a los puntos de entrenamiento más críticos.

$$k(x,x') = tan(kx \cdot x' + c), \text{ para algunos k>0 y c<0} \qquad (2.4)$$

Para finalizar esta sección, cabe destacar que, como se menciona en [10], existen otros tipos de *kernel* que, para algunas aplicaciones, nos aportan una serie de ventajas sobre los demás. Estos *kernels* son, o bien de tipo PHMM (del inglés, *Pair Hidden Markov Moddel*) o bien utilizan la transformada de Fourier para clasificar los casos en las que la información está repartida a lo largo del espectro de frecuencias.

2.4. Aplicaciones de SVM en imágenes hiperespectrales

Ahora que los conceptos básicos relacionados con el clasificador SVM y las imágenes hiperespectrales han sido explicados, en este apartado se proporciona un resumen sobre los campos de aplicación en los que confluyen estas dos tecnologías. Cabe destacar que este estudio nos ofrece una visión global de los campos de utilización habituales pero, debido a que este documento se inscribe dentro de un proyecto europeo orientado a la detección del cáncer mediante la utilización de imágenes hiperespectrales, tiene como objetivo fundamental resaltar su utilización en el campo de la medicina.

Dado que las imágenes hiperespectrales surgieron para estudiar la superficie terrestre desde satélites – técnica denominada en la literatura como *remote sensing* –, las principales aplicaciones de SVM sobre imágenes hiperespectrales están desarrolladas para este campo. Este hecho se refleja en estudios como [10 - 13].

A continuación, se hará una breve descripción de la aplicación de estas tecnologías en conjunto en *remote sensing* y en detección de cáncer, explicando las ventajas de utilizar imágenes hiperespectrales y de clasificarlas utilizando SVM. Además, se aportarán, en la medida de lo posible, ejemplo relevantes de su utilización.

2.4.1. *Remote sensing*

Como se ha mencionado anteriormente, las primeras aplicaciones que utilizaron la tecnología de imagen hiperespectral estaban enfocadas a la observación terrestre. El objetivo con el que se desarrolló esta tecnología fue el de la identificación y localización de los materiales en la superficie terrestre, gracias a la utilización de sensores de imágenes hiperespectrales aerotransportados.

Al ser una aplicación enfocada a la identificación de materiales en una escena utilizando imágenes hiperespectrales, parece intuitiva la idea de utilizar un clasificador para distinguir entre los elementos que componen dicha imagen. En la literatura encontramos dos tipos de estudios con objetivos diferentes; en primer lugar, existen estudios enfocados a analizar el correcto funcionamiento del clasificador SVM frente a una imagen de alta dimensionalidad [6] y [11]. Mientras que, por otro lado, existen estudios dedicados a optimizar el funcionamiento del clasificador SVM al trabajar con imágenes hiperespectrales [12] y [13].

En la primera de estas vertientes, los estudios realizados corroboran que la utilización de SVM para clasificar los píxeles de una imagen hiperespectral arroja resultados satisfactorios debido a que no sufre el efecto Hughes [14] – incremento del error en la clasificación al aumentar el número de parámetros o *features* a tener en cuenta utilizando el mismo número de muestras de entrenamiento –. El clasificador SVM no sufre este efecto ya que directamente busca los parámetros óptimos para realizar la clasificación utilizando lo que se conoce como vectores soporte (del inglés, *support vector*). Estos vectores se caracterizan por ser las muestras más representativas del conjunto de entrenamiento.

Para corroborar el correcto funcionamiento del clasificador SVM aplicado a imágenes hiperespectrales, en [6] se aplica dicho clasificador a una imagen obtenida por el satélite AVIRIS (224 bandas) y otra capturada por el sensor AISA (entre 20 y 40 bandas). En el caso de la imagen obtenida por el satélite AVIRIS, se ha utilizado una captura de *Salinas Valley* en Carolina tomada el 9 de Octubre en 1998. En esta imagen, utilizando un 1% de los píxeles como muestras de entrenamiento y un 99% de test, se ha obtenido un 89% de efectividad en la clasificación. Por otro lado, se ha utilizado una imagen de *Delmarva Peninsula* capturada por el sensor AISA y, utilizando también un 1% de los píxeles totales para realizar el entrenamiento, se han conseguido los resultados mostrados en la tabla 2.I a la hora de clasificar dos tipos distintos de maíz y dos tipos de soja.

Corn A Pioneer 34KB2	Corn B Pioneer 33G26	Soybeans: C Asgrow AG4301	Soybeans: D Pioneer 9421WTW
99.9	99.9	94.6	74.3

Tabla 2.I. Rendimiento de clasificación para diferenciar cosechas [6]

Sin embargo, con el fin de comprobar la robustez del clasificador SVM frente a la inclusión de numerosas clases entre las que diferenciar, en [11] se procesa una imagen distinguiendo entre 4 clases en primer lugar y, posteriormente, entre 16. Para ello se utiliza una imagen obtenida por el satélite AVIRIS. Como se muestra en la tabla 2.II, se ha obtenido como resultado que, aunque la efectividad del clasificador se reduce cuando se incrementa el número de clases entre las que debe distinguir, el clasificador SVM sigue trabajando de manera efectiva.

PERFORMANCE	
Subset Scene	Full Scene
95.9%	87.3%

Tabla 2.II. Comparación de rendimientos para 4 y 16 clases [11]

En el caso de optimización del clasificador SVM analizaremos dos investigaciones. En la primera de ellas [12], se realizó un estudio para conocer el *kernel*, los parámetros y la estructura de SVM que mejor se adapta al análisis de imágenes hiperespectrales. Como conclusión, defienden que el SVM es mucho más robusto que otros clasificadores (*RBF neural network* o *K-nn classifier*) en términos de precisión, velocidad de cómputo y estabilidad ante el cambio de parámetros de configuración. Por otro lado, sacan a la luz la problemática de utilizar este clasificador como diferenciador binario – entre dos clases únicamente – y colocar varios clasificadores en cascada con el fin de distinguir entre un mayor número de clases. Esta configuración puede ser muy sencilla de implementar y de entrenar pero, en consecuencia, los errores que se generen en las primeras etapas, son arrastrados a través del resto de la estructura.

En segundo lugar, con el fin de optimizar la precisión del clasificador SVM, en [13], se analizan los diferentes modelos de clasificador y, a su vez, la determinación de las características o *features* más apropiadas para realizar correctamente la clasificación. Como resultado se observa que utilizando un modelo no lineal de SVM en lugar de un modelo lineal, se obtiene una mayor precisión con la desventaja de tardar más en realizar el entrenamiento. A su vez, este modelo no lineal es capaz de evitar la parte empírica basada en prueba y error que se utiliza normalmente al analizar imágenes hiperespectrales.

2.4.2. Detección de cáncer

Debido al gran impacto que ha tenido la utilización de imágenes hiperespectrales en el campo de la medicina, se han desarrollado un gran número de aplicaciones. Un amplio análisis de la literatura relacionada con aplicaciones desarrolladas entre el año 1988 y el 2013 está recogido en [15]. En este resumen, se repasan los logros conseguidos y las posibles futuras líneas de investigación.

Dentro de estos campos de aplicación podemos destacar la utilización de dichas imágenes en la detección, localización y posterior guía en la extirpación de tumores cancerígenos. En nuestro caso, nos centraremos en analizar aquellas aplicaciones en las que se utiliza un clasificador, principalmente el SVM, para realizar la diferenciación entre tejido sano y tejido tumoroso.

En primer lugar, un estudio recogido en el *Journal of Biomedical Optics* muestra la capacidad de detectar tejido prostático canceroso en ratones [15]. Durante este análisis, se ha demostrado que la utilización de imágenes hiperespectrales permite diferenciar de una manera no invasiva el tejido tumoroso del que no lo es y, utilizando un clasificador SVM, se obtiene un 92.8% de efectividad en la localización del mismo. Este estudio se ha realizado sobre 11 ratones y, el resultado definitivo, es que se podría utilizar esta tecnología en un quirófano ayudando al cirujano con una imagen como la mostrada en la figura 2.10.

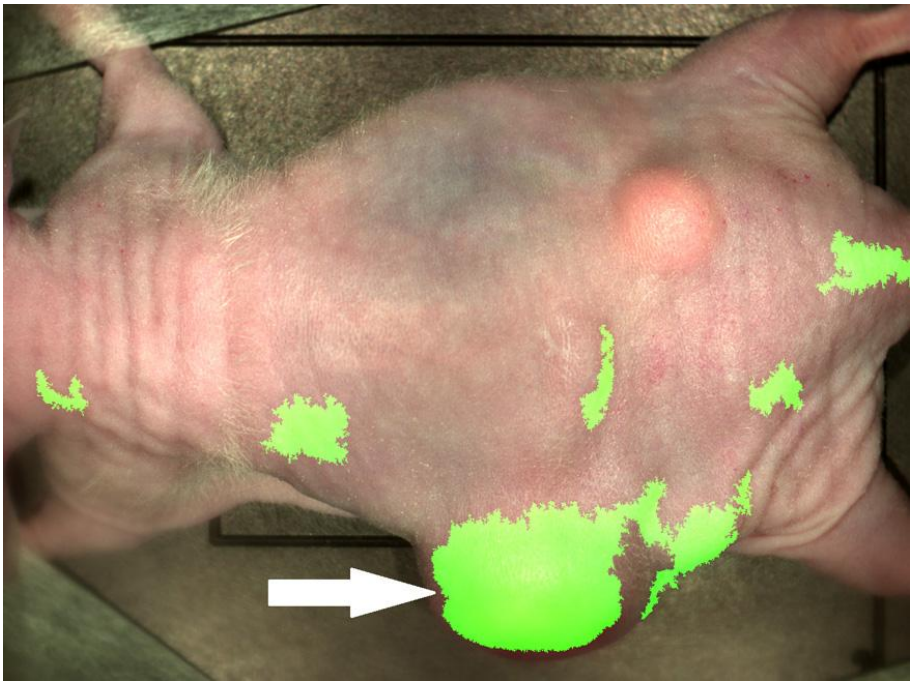

Figura 2.10. Localización de tejido tumoroso en ratones

Vistos los resultados obtenidos al localizar tejido tumoroso en animales, el siguiente paso es aplicar este procedimiento a muestras de tejido humano. Un ejemplo de esto es la diferenciación entre tejido sano y tejido tumoroso en muestras de colon humanas que fue llevado a cabo en 2004 por la Universidad de *Warwick* en colaboración con el Instituto GIK de Pakistán [16]. Para realizar dicho estudio, se utilizan 20 bandas pertenecientes al rango de 450-640 nm. El método de procesado se divide en tres etapas:

- Segmentación de la imagen hiperespectral en cuatro clases. Para ello, se ha realizado la diferenciación a nivel microscópico seleccionando, las diferentes clases e indicando la firma espectral asociada a cada una de ellas.

- En segundo lugar, se lleva a cabo una extracción de características discriminatorias que se asociarán a cada una de las clases. Por ejemplo, algunas de estas características son el área, el diámetro equivalente, la orientación, el número de Euler, etc.

- Por último, se realiza la clasificación de las muestras. En este paso, primero se realiza un entrenamiento del clasificador SVM utilizando un total de 30000 muestras, y, tras este entrenamiento, se lleva a cabo la clasificación. Cabe mencionar que, durante la realización de este estudio se han realizado pruebas utilizando diferentes *kernels* con el fin de optimizar dicha clasificación.

Tras este estudio se ha demostrado que la utilización del clasificador SVM para realizar la diferenciación entre tejido sano y tejido tumoroso de colon humano arroja resultados satisfactorios. Como conclusión se recogen en la tabla 2.III los resultados obtenidos en 4 pruebas diferentes utilizando los *kernels* más típicos. Como podemos observar, el *kernel* Gaussian es el que presenta un mayor nivel de precisión.

Trial	Linear	Gaussian	Polynomial
1	82.2	86.9	83.1
2	81.5	87.8	83.2
3	81.5	86.7	83.9
4	82.0	87.1	83.7

Tabla 2.III. Comparativa de precisión en la clasificación [16]

Por último, un estudio reciente (2013) probó la capacidad del clasificador SVM de distinguir entre tejido sano y tumoroso en muestras de pulmón y ganglios linfáticos. Debido a la capacidad de este clasificador para realizar la diferenciación píxel a píxel, los doctores no tienen que preocuparse de la forma que tenga el tumor ya que al analizar todos los píxeles por separado, el resultado arrojado no depende de los píxeles vecinos.

Para realizar este estudio se han utilizado 3 placas de cada tipo (tejido sano y tumoroso de pulmón y tejido sano y tumoroso de ganglios) haciendo un total de 12 muestras con las que realizar la diferenciación. Gracias a la utilización de imágenes hiperespectrales podemos analizar las muestras de tejido de una manera no invasiva y en menos tiempo que realizando una biopsia y analizando la muestra extraída.

Los resultados obtenidos están reflejados en la tabla 2.IV y se han obtenido utilizando un clasificador *Least Squares* SVM. Este clasificador se caracteriza por utilizar ecuaciones lineales en lugar de una programación cuadrática convexa lo que reduce aun más la complejidad de la computación. Como podemos observar, los resultados se dividen en cuatro valores diferenciados:

- Precisión (*Specificity*): este parámetro representa el número de negativos correctamente clasificados.
- Sensibilidad (*Sensitivity*): al contrario de la precisión, la sensibilidad representa los positivos correctamente clasificados.
- FPR: del inglés, *False Positive Rate*, representa el porcentaje de falsos positivos detectados.
- FNR: por el contrario, el *False Negative Rate*, representa el porcentaje de falsos negativos clasificados.

	Specifity (%)	Sensitivity (%)	FPR (%)	FNR (%)
Lymph nodes	98.3	96.2	1.7	3.8
Lungs	97.7	92.6	2.3	7.4

Tabla 2.IV. Resultados de la clasificación píxel a píxel

2.5.RVC – CAL

Con el fin de ampliar la librería implementada durante la investigación recogida en [1] y [2], este proyecto se desarrollará utilizando RVC – CAL (*Reconfigurable Video Coding – CAL*) [17]. Este lenguaje está basado en el lenguaje conocido como CAL (*CAL Actor Language* o Lenguaje de Actores CAL) [18]. Debido a la gran similitud que guardan ambos lenguajes de programación, es necesario explicar la estructura que utiliza CAL para entender el funcionamiento de RVC – CAL.

CAL es un lenguaje de programación de alto nivel desarrollado en 2011 en la Universidad de Berkeley basado en flujo de datos. Este tipo de lenguajes de flujo de datos tienen como objetivo facilitar la implementación de aplicaciones divididas en bloques de funcionalidad y, a su vez, establecer una comunicación entre dichos bloques. Esta idea de codificación por bloques nos posibilita la realización de un reparto de éstos entre procesadores y, de este modo, conseguir paralelizar la ejecución de la aplicación desarrollada.

El concepto de paralelización de procesadores resulta complejo de implementar cuando se utilizan lenguajes de programación secuencial tales como C, C++, Java, etc. En consecuencia, la utilización de un lenguaje basado en flujo de datos como CAL nos permite alcanzar un nivel de abstracción en el que la paralelización es transparente para el usuario.

Con el fin de facilitar la división del trabajo en bloques, CAL utiliza una estructura básica compuesta de actores, acciones y *networks* – o redes –. A continuación, se analizan en profundidad estos conceptos, con el objetivo de obtener una idea clara de los elementos principales que componen la estructura de CAL:

- El actor es la unidad más básica y característica de CAL. Este actor es lo que, con anterioridad, se ha denominado bloque y, tal y como se observa en la figura 2.11, se trata de una entidad compuesta de puertos de entrada, una o varias funcionalidades internas y un conjunto de salidas. Dichas entradas y salidas nos permiten ver claramente que cada actor puede comunicarse con uno o varios actores. El proceso de funcionamiento de un actor sería, en resumen, utilizar los datos recibidos – e incorporar nuevos de un fichero almacenado en disco si es necesario – para realizar una serie de operaciones cuyo resultado es enviado a otro actor. Dentro de RVC –

CAL, cabe destacar que los bloques de datos que se transmiten entre actores se denominan *tokens.*

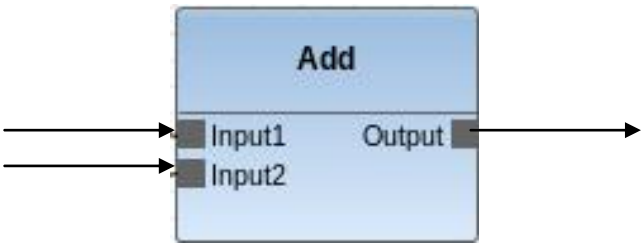

Figura 2.11. Gráfico de un actor en lenguaje CAL

- Una vez definido el concepto de actor como bloque funcional, es necesario definir el siguiente concepto característico presente en este lenguaje: la acción. Se entiende por acción cada una de las funcionalidades internas existentes dentro de un actor.

  Cada acción, a su vez, puede estar asociada o no con las distintas entradas y salidas declaradas durante la creación del actor y, en consecuencia, realizar una funcionalidad distinta al resto. Esta característica permite al usuario organizar el código de tal manera que se divida, dentro de un mismo actor, en diversas funcionalidades que dependan – o no – las unas de las otras.

  Otra de las características principales de las acciones es la capacidad que nos ofrece CAL para condicionar su ejecución ya que, aprovechando esta cualidad del lenguaje, se puede implementar la secuenciación de acciones que más se adapte a nuestras necesidades de manera rápida y eficiente.

- Tras definir la estructura interna de un actor, únicamente nos queda por definir el concepto de *network*. Entendemos por *network* lo siguiente: conjunto de varios bloques funcionales – en su mayoría actores – conectados entre sí y que componen parte o la totalidad de una aplicación. Cabe destacar que, en la configuración estándar de una *network*, existen, aparte de los actores explicados anteriormente, unas instancias específicas para implementar la interfaz de entrada y salida del sistema. Estas instancias se denominan dentro de este lenguaje *input and output ports* – puertos de entrada y salida –. Debido a la complejidad variable que puede tener la entrada o salida de un sistema, existen situaciones en las que la utilización de estos puertos no es viable; por ejemplo, cuando el número de

entradas o salidas es muy numeroso o cuando la entrada/salida del sistema es compleja. En estos casos, se utilizan actores denominados *source* – fuente – y *display* – visor – que realizan esta funcionalidad específica. Un ejemplo clásico en el que se da esta situación es cuando la entrada del sistema es un fichero almacenado en disco.

Una vez explicada la estructura básica de este tipo de lenguajes, somos capaces de dar una definición formal y explicar en profundidad RVC – CAL.

RVC – CAL es un lenguaje de flujo de datos – estructura heredada de CAL – cuya principal característica es, tal y como indica su nombre, su orientación al desarrollo de codificadores y decodificadores de vídeo reconfigurables. Este lenguaje está siendo desarrollado por MPEG (*Moving Pictures Experts Group* o Grupo de Expertos en Imágenes en Movimiento) y, con él, están desarrollando un nuevo estándar de codificadores y decodificadores de vídeo [19]. Este nuevo estándar se basa en desarrollar los codificadores, no como un todo, si no como bloques funcionales que, unidos de una forma concreta, conformen un decodificador completo. Este desarrollo por bloques permite que, si se quiere modificar una fase del decodificador – o actor –, únicamente es necesario modificar un actor concreto o sustituir dicho actor por otro cuyo funcionamiento sea el requerido.

Para analizar este lenguaje en profundidad, hay que tener en cuenta que su programación está dividida en dos interfaces independientes. Durante la primera de ellas, tenemos que implementar una interfaz gráfica en la que se instanciará el número de bloques y las conexiones – enlaces de transmisión de datos – que existirán entre ellos. En segundo lugar, implementaremos la interfaz funcional. En ella se definirán las distintas funciones o tareas que realizará cada uno de los bloques en los que se divide la aplicación – estos bloques, en RVC – CAL, reciben el nombre de actores –.

Con el fin de facilitar la implementación de la interfaz gráfica propia de las aplicaciones en lenguaje RVC – CAL se ha desarrollado un plugin de Eclipse denominado ORCC (*Open RVC – CAL Compiler* o Compilador de RVC – CAL abierto) con el que se pueden observar claramente los elementos característicos de CAL explicados anteriormente.

Desde ORCC se puede observar gráficamente el concepto de network como una entidad compuesta por una serie de instancias conectadas entre sí a través de puertos de entrada y de salida. En la figura 2.12 se muestra una *network* sencilla implementada en ORCC. En ella únicamente se utiliza un actor básico – denominado *Add* – compuesto por dos entradas y una salida.

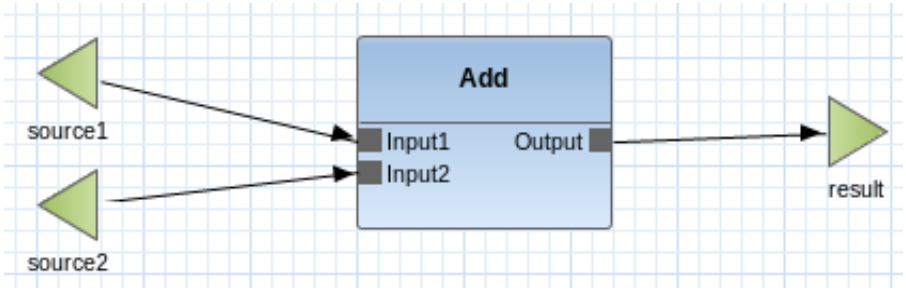

Figura 2.12. *Network* básica en ORCC

Como se ha mencionado con anterioridad, los puertos de entrada y salida del sistema pueden llegar a ser muy complejos y sustituirse por entidades denominadas *source* y *display*. Estas entidades tienen como característica principal o bien no tener entradas – en el caso del *source* –, o no tener salidas – en el caso del *display* –. Esto se debe a que estas entidades estarían leyendo o escribiendo los datos utilizados, por ejemplo, de disco. Una *network* en la que se utilizan estas entidades tendría una distribución similar a la que aparece en la figura 2.13. En este caso existen dos source distintos denominados *Actor1* y *Actor2* y un display denominado *Printer*.

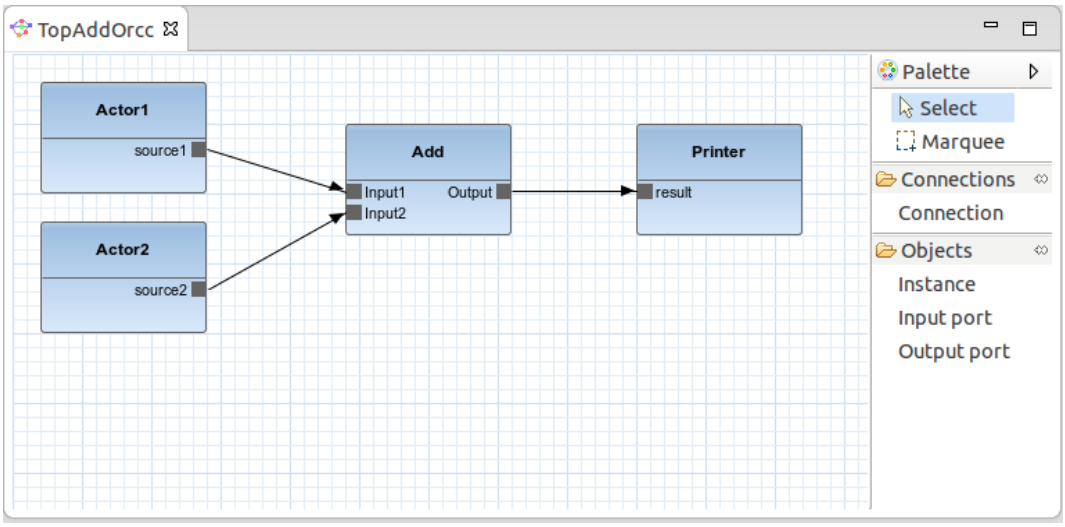

Figura 2.13. *Network* con entrada y salida implementada con actores (Fuente: *http://orcc.sourceforge.net/tutorials/a-very-simple-network/*)

En la figura 2.14 se puede observar un ejemplo de *network* compleja. En este caso, se puede observar la capacidad de realizar un sistema en el que exista un alto grado de comunicación entre actores o, incluso, implementar realimentación. A su vez, si tenemos en cuenta que cada instancia se puede ejecutar desde un procesador distinto, se puede comprobar de manera casi intuitiva que implementar la paralelización de una aplicación es una tarea relativamente sencilla.

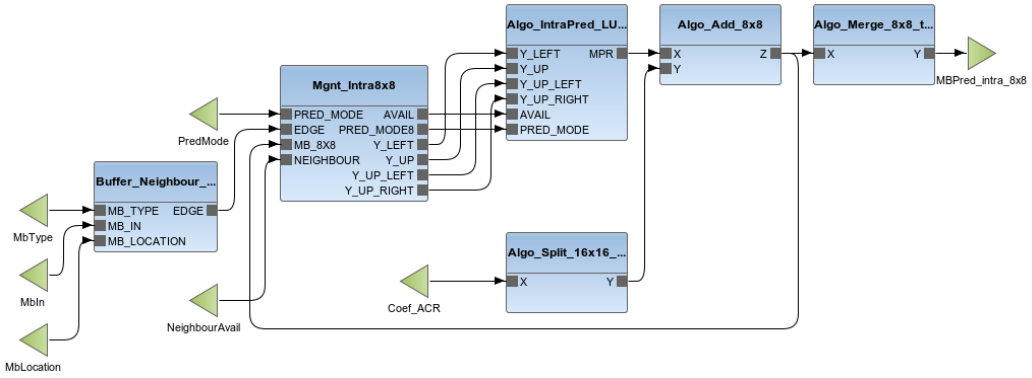

Figura 2.14. *Network* con paralelismo complejo en ORCC
(Fuente: *http://orcc.sourceforge.net/*)

Cabe destacar que, durante la explicación de ORCC, se ha utilizado en algunas ocasiones la palabra entidad en lugar de actor para denominar a los distintos bloques que componen una misma *network*. Esto se debe a que una entidad puede ser, a su vez, otra *network* más pequeña – o con una funcionalidad más específica –. Esto se observa claramente en la figura 2.15, en la que existe una instancia constituida por un decodificador completo denominado *HevcDecoder*. En ella se observa que, para diferenciar estos tipos de entidades, se utilizan diferentes colores: amarillo para las *networks* y azul para los actores.

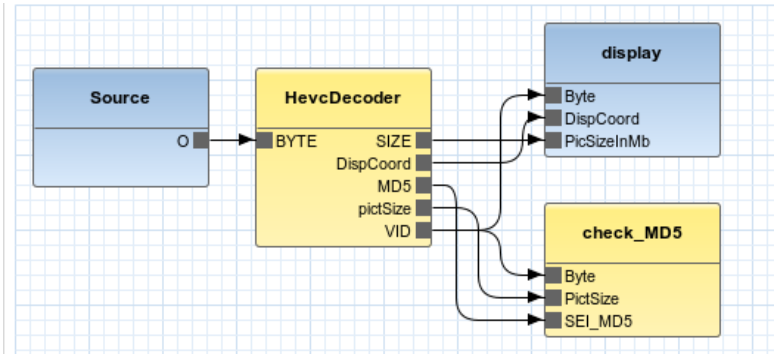

Figura 2.15. *Network* incluida en otra más compleja
(Fuente: *http://orcc.sourceforge.net/tutorials/make-an-hevc-decoder/*)

Para finalizar la explicación de ORCC, queda puntualizar su capacidad para generar código en distintos lenguajes. Como se explica en [20], [21] y [22] se puede generar código en lenguajes tales como C, PREESM, Jade, LLVM, etc. A modo de ejemplo, en la tabla 2.V se recogen los distintos lenguajes generables para las diferentes tecnologías desarrolladas con RVC – CAL. En ella se observan los códigos para los que están disponibles (OK), los que se encuentran en desarrollo (NOK) y los que aún no se han empezado a implementar (N/A).

	MPEG-4 Part 2 SP	MPEG-4 Part 10	MPEG-H Part 2	JPEG
C	OK	OK	OK	OK
HLS	OK	NOK	NOK	N/A
Jade	OK	OK	OK	N/A
LLVM	OK	OK	OK	N/A
Promela	OK	N/A	N/A	N/A
Simulator	OK	OK	NOK	OK
TTA	OK	NOK	OK	N/A
Xronos	OK	NOK	N/A	OK
DAL	N/A	N/A	N/A	N/A

Tabla 2.V. Lenguajes disponibles en ORCC

A continuación, tras explicar el funcionamiento del plugin ORCC y su capacidad para construir una *network* y establecer las comunicaciones apropiadas, citando, a su vez, algunos de los lenguajes de programación que permite generar en su compilación, queda centrarse en la sintaxis propia del lenguaje RVC – CAL mediante la cual se implementará la interfaz funcional de nuestra aplicación.

En primer lugar, para instanciar un actor se sigue la estructura expuesta en la figura 2.16. Como podemos observar en la imagen, se indican el nombre del actor y el nombre y tipo de dato de los distintos puertos de entrada (antes de la flecha) y salida (tras la flecha) – *int*, *double*, *float*, etc-.

```
actor Actor() int Input1, double Input2, bool config ==> float Output:
```

Figura 2.16. Sintaxis para la declaración de un actor

Estos actores pueden agruparse en paquetes y, a su vez, desde un actor, se puede importar el contenido de un paquete externo con el fin de incorporar las variables o las funciones implementadas en el mismo. El manejo de paquetes se realiza mediante las sentencias mostradas en la figura 2.17. Para indicar el paquete al que pertenece el actor que estamos codificando se utiliza el término *package*, mientras

que para importar variables o funciones incluidas en un actor que forma parte de un paquete externo se utiliza la sentencia *import*.

```
package example;

import other_packages.*;
```

Figura 2.17. Ejemplo de utilización de paquetes

Por otro lado, la declaración de una acción se realiza de la manera que se indica en la figura 2.18. En ella observamos dos entradas, que se corresponden con las entradas *Input1* e *Input2* declaradas en el actor ejemplo, y una salida, que se corresponde con la salida *Output* de ese mismo actor. Cabe destacar que en la declaración de la acción es donde, si procede, se indica el tamaño del array asociado al *token* de entrada o de salida. Este tamaño se indica utilizando el comando *repeat*. En este caso, *Input1* tiene un tamaño de 256 elementos.

```
tarea1: action Input1:[array] repeat 256, Input2:[extra_value] ==> Output:[result]
```

Figura 2.18. Declaración de una acción

Por tanto, la estructura completa de un actor sería la que se observa en la figura 2.19. En ella se muestra el actor utilizado anteriormente, el paquete al que pertenece, la utilización de otros paquetes y, por último, las diversas acciones en las que se divide dicho actor.

La acción denominada *tarea2* es un ejemplo de una tarea que únicamente consta de entradas. Esto demuestra que, como ocurría con los actores, no es necesario que todas las tareas consten tanto de entradas como de salidas.

```
package example;

import other_packages.*;

actor Actor() int Input1, double Input2, bool config ==> float Output:

    tarea1: action Input1:[array] repeat 256, Input2:[extra_value] ==> Output:[result]
    end

    tarea2: action Input2:[value], config:[option] ==>
    end

end
```

Figura 2.19. Estructura de un actor completo

Una vez explicada la estructura general de un actor en RVC – CAL se realiza un repaso de los principales recursos presentes en este lenguaje:

- En primer lugar, es necesario distinguir entre constantes y variables y, a su vez, entre las que son globales (utilizables durante todo el actor) y las que son locales y únicamente se utilizan dentro de una tarea. La única diferencia existente a la hora de declarar una constante o una variable reside en el elemento con el que se realiza la asignación. Concretamente, las constantes se asignan mediante el elemento "=" – signo de igualdad – y las variables con ":=" – dos puntos seguidos del signo de igualdad–. Por otro lado, para distinguir entre variable global y variable local de una tarea, hay que observar el lugar donde se realiza la declaración. La instanciación global se realiza entre la cabecera del actor y la primera de las tareas pero, al realizar una instanciación local de una tarea, hay que introducir el término *var*. Estos conceptos están reflejados claramente en la figura 2.20.

```
actor Actor() int Input1, double Input2, bool config ==> float Output:

    int(size = 32) constante_global = 5;
    int(size = 32) variable_global := 5;

    tarea: action Input2: [value], config:[option] ==>
    var
        int(size = 32) constante_local = 5,
        int(size = 32) variable_local := 5
    do

    end

end
```

Figura 2.20. Instanciación de constantes y variables

- A continuación, tal y como se observa en la figura 2.21 (A) existen, como en la mayoría de lenguajes habituales, unos elementos de condición. La principal diferencia con respecto a lenguajes como C es que esta comparación – sentencias *if-elsif-else* – se realizan con el símbolo "=" – signo de igualdad – y no con "= =" – dos signos de igualdad seguidos – como es habitual. Como caso característico de este lenguaje, en la figura 2.21 (B) observamos un ejemplo en el que la condición está impuesta al inicio de una acción, esto es, si la condición dentro del conjunto *guard-end* no se cumple, la acción a la que está asignada no se ejecutará.

A)

```
if(variable = 5)then
    //Condición 1
elsif (variable =7) then
    //Condición 2
else
    //Resto
end
```

B)

```
tarea: action Input2:[value], config:[option] ==>
guard
    variable = 5 and variable2 = 7
do
//Cuerpo de la tarea
end
```

Figura 2.21. Condición estándar (A) Condición de ejecución de tarea (B)

- Otro de los elementos clásicos presente en la mayoría de los lenguajes es el bucle. En este lenguaje existen dos formas de utilizarlo.

  En primer lugar está el bucle *while* común, el cual se ejecuta todas las veces necesarias hasta que se deja de cumplir la condición de inicio. Un ejemplo de uso es el que se observa en la figura 2.22.

```
while i < MAX do
    //Código a ejecutar mientras se cumple la acción
end
```

Figura 2.22. Bucle *while*

  En segundo lugar, encontramos el también clásico bucle *for* que, en nuestro caso, recibe el nombre de *foreach*. Dicho bucle se repite tantas veces como se indique en la declaración. La sintaxis necesaria para declarar un bucle de este tipo la podemos observar en la figura 2.23.

```
foreach int(size=32) i in 0 .. n do
    //Código a ejecutar n veces
end
```

Figura 2.23. Bucle *foreach*

- Por último, los principales operadores están divididos en dos grupos [18]. El primero de ellos está recogido en la tabla 2.VI y se refiere a los operadores unitarios.

Operator	Operand type	Meaning
not	Boolean	Logical negation
#	Collection [T]	Number of elements
	Map[K, V]	Number of mappings
dom	Map[K, V]	Domain of a map
rng	Map[K, V]	Range of a map
-	Number	Arithmetic negation

Tabla 2.VI. Tabla de operadores unitarios [18]

El segundo de ellos está recogido en la tabla 2.VII y se refiere a los operadores binarios.

P	Operator	Operand 1	Operand 2	Meaning
1	and	Boolean	Boolean	Logical conjunction
	or	Boolean	Boolean	Logical disjunction
2	=	Object	Object	Equality
	!=	Object	Object	Inequality
	<	Number	Number	Less than
		Set [T]	Set [T]	
		String	String	
		Character	Character	
	<=	Analogous to <		Less than or equal
	>	Analogous to <		Greater than
	>=	Analogous to <		Greater than or equal
3	in	T	Collection [T]	Membership
4	+	Number	Number	Addition
		Set [T]	Set [T]	Union
		List [T]	List [T]	Concatenation
		Map [K, V]	Map [K, V]	Map union
	-	Number	Number	Difference
		Set [T]	Set [T]	Set difference
5	div	Number	Number	Integral division
	Mod	Number	Number	Modulo
	*	Number	Number	Multiplication
		Set [T]	Set [T]	Intersection
	/	Number	Number	Division
6	^	Number	Number	Exponentiation

Tabla 2.VII. Tabla de operadores binarios [18]

Para finalizar la explicación de la sintaxis propia de RVC – CAL, únicamente queda por analizar la forma de programar la ejecución de las acciones que componen un mismo actor.

Esta organización se puede realizar de tres formas distintas:

- La primera de ellas es utilizar las sentencias *guard* explicadas anteriormente en este mismo documento. Un ejemplo de este tipo de sentencias puede verse en la figura 2.24 (A).

- La segunda es utilizar un elemento denominado *priority* que fija un orden de ejecución concreto, asignando a cada tarea un nivel de prioridad y ejecutando dichas tareas de mayor a menor prioridad. Esto se observa en la figura 2.24 (B).

- La última forma de realizar esta asignación es utilizar otro elemento propio del lenguaje RVC – CAL: el *scheduler* – o planificador de tareas. Este organizador funciona como un autómata, indicando, acción por acción, la tarea que se llevará a cabo tras ella. Un ejemplo de esta organización también puede observarse en la figura 2.24 (C).

A)
```
tarea1: action Input2:[value], config:[option] ==>
guard
    tarea1_done = false
do
    //Código de la tarea 1
    tarea1_done := true;
end
tarea2: action Input1:[value] ==> Output:[result]
guard
    tarea1_done = true and tarea2_done = false
do
    //Código de la tarea 2
    tarea2_done := true;
end
```

B)
```
priority
    tarea1 > tarea2;
end
```

C)
```
schedule fsm s_first:
    s_first (tarea1) --> s_second;
    s_second (tarea2) --> s_first;
end
```

Figura 2.24. Formas de organizar las tareas: mediante el uso del *guard* (A), mediante la utilización del comando *priority* (B) y mediante el uso del *scheduler* (C)

Tras analizar la sintaxis completa de este lenguaje, en la figura 2.25 se aporta un ejemplo de actor completo (obtenido de la página oficial de ORCC) con la mayoría de los elementos utilizados a lo largo de este estudio.

```
package org.mpeg4.part10.cbp.selectMacroblock;

import org.mpeg4.part10.cbp.MacroBlockInfo.BLOCK_TYPE_INTRA_MAX;

actor SelectMb()
    uint(size=6) MbType,
    uint(size=8) MbFromIntra,
    uint(size=8) MbFromInter
        ==>
    uint(size=8)  MbOut
        :
    uint(size=6) mbType;

    getMbType: action MbType:[valMbType] ==>
    do
        mbType := valMbType;
    end

    forwardMb.intra: action MbFromIntra:[x] repeat 64 ==>
        MbOut:[x] repeat 64
    guard
        mbType <= BLOCK_TYPE_INTRA_MAX
    end

    forwardMb.inter: action MbFromInter:[x] repeat 64 ==>
        MbOut:[x] repeat 64
    guard
        mbType > BLOCK_TYPE_INTRA_MAX
    end

    schedule fsm GetMbType:
        GetMbType (getMbType  ) --> ForwardMb;
        ForwardMb (forwardMb  ) --> GetMbType;
    end
end
```

Figura 2.25. Ejemplo de actor completo
(Fuente: *http://orcc.sourceforge.net/*)

Una vez realizado el análisis de este lenguaje, se recogen las principales ventajas e inconvenientes de la utilización del mismo durante el desarrollo de esta investigación.

2.5.1. Ventajas

- Implementación de código por actores – división de la aplicación en bloques.
- Comunicación sencilla entre actores.
- Posibilidad de realizar paralelización (mapeo) directamente desde ORCC.
- Generación del código realizado en RVC – CAL en distintos lenguajes, mejorando la interoperabilidad entre plataformas.
- Capacidad de importación de librerías externas en un lenguaje compatible con el generado.

2.5.2. Inconvenientes

- Escasez de librerías propias.
- Lenguaje en desarrollo (abundancia de nuevas versiones).
- Escasez de recursos propios del lenguaje: las distintas operaciones admiten únicamente tipos de datos muy concretos.
- No genera ejecutables directamente sino que, a partir de una descripción implementada en RVC – CAL, genera código en otros lenguajes tales como C, PREESM, Jade, etc. y, posteriormente, hay que realizar una compilación.

3 Descripción de la solución desarrollada

3.1. Estructura

Como se mencionó en el capítulo 1 de este mismo documento, el objetivo principal de este Trabajo de Investigación es implementar una versión del clasificador SVM para imágenes hiperespectrales en RVC – CAL para, de esta manera, facilitar, en la medida de lo posible, la clasificación en tiempo real. Por tanto, a lo largo del presente capítulo se realizará una explicación detallada del procedimiento seguido durante la investigación para alcanzar dicho objetivo.

Para realizar la explicación de una manera clara y sencilla, se ha dividido este capítulo en tres apartados que se corresponden con las tres etapas principales de la investigación:

- *Estudio de SVM versión Matlab*: durante esta fase se ha analizado en profundidad una versión del clasificador SVM codificada en Matlab haciendo especial hincapié en la comprensión de su estructura, su algoritmia y el funcionamiento general del algoritmo.

- *Análisis funcional*: tras entender la estructura y el camino seguido en la implementación Matlab del SVM, se extraen y analizan los tipos de función de los que está compuesto. Tras esto, se resumen las ventajas e inconvenientes que presenta la versión Matlab para alcanzar el procesado en tiempo real y se busca una alternativa.

- *Adaptación a RVC – CAL*: dadas las ventajas de interoperabilidad entre plataformas, la posibilidad de utilizar una librería externa en C o C++ y la capacidad de realizar una paralelización que nos ofrece RVC – CAL se decide realizar una adaptación del clasificador SVM a este lenguaje.

3.2. Estudio de SVM versión Matlab

Durante esta sección se analizará en profundidad el algoritmo de un clasificador SVM para imágenes hiperespectrales. Esta versión se encuentra codificada en lenguaje Matlab y ha sido proporcionada por los investigadores de la Universidad de Las Palmas de Gran Canaria (ULPGC).

Tras un primer análisis, se ha extraído la estructura general del algoritmo. Dicha estructura está compuesta por cuatro bloques en los que se utiliza un set de entrenamiento de un tamaño diferente (1%, 3%, 5% y 10% del número total de píxeles de la imagen original). A su vez, para cada porcentaje, se construyen diez sets de entrenamiento diferentes. Es decir, en total, se implementan un total de cuarenta modelos de clasificación diferentes – con sus respectivas clasificaciones.

Por otro lado, si observamos los pasos en los que se divide cada una de estas clasificaciones, observamos que está compuesto de cuatro etapas: selección del set de entrenamiento, entrenamiento del clasificador SVM, clasificación de la imagen y cálculo de estadísticas.

3.2.1. Selección del set de entrenamiento

El objetivo de esta fase es seleccionar los píxeles de la imagen original que servirán para realizar cada uno de los entrenamientos del clasificador SVM. Para ello, se calcula un total de 40 sets de entrenamiento – 10 sets para cada uno de los porcentajes implicados en el algoritmo desarrollado.

Las entradas de esta fase son tres: un vector con la clase con la que se corresponde cada píxel de la imagen de entrenamiento, el número ideal de píxeles de entrenamiento de cada clase y el número total de píxeles con los que realizaremos el entrenamiento. A continuación, se explica paso a paso el proceso seguido para calcular cada uno de los sets:

1. Se calcula el número de clases que hay en la imagen buscando el valor máximo del vector de clases.

2. Se extrae la posición que ocupa en la imagen cada píxel que corresponde a cada clase utilizando el vector de clases y, posteriormente, se escogen aleatoriamente tantas muestras como indique la entrada de número ideal de píxeles.

3. Si en alguna de las clases no se alcanza el número ideal, se completa el vector de salida con píxeles aleatorios de cualquiera de las clases – evitando aquellos que ya hayan sido añadidos al set de entrenamiento.

4. Se construye el vector de salida que contendrá el set de entrenamiento de cada modelo de clasificación.

Como resultado de esta fase tenemos las posiciones del conjunto de píxeles para realizar el entrenamiento de cada uno de los 40 modelos de clasificación que implementaremos. Durante la construcción de estos sets de píxeles, se ha buscado que estuvieran compuestos por un número equilibrado de muestras de cada clase.

3.2.2. Entrenamiento del clasificador SVM

Esta fase del algoritmo se encarga de realizar el entrenamiento del clasificador y obtener cada uno de los modelos de clasificación que serán utilizados posteriormente.

En este caso, las entradas disponibles son: el vector con los índices que indican los píxeles escogidos en la etapa anterior y los diferentes parámetros con los que queremos configurar el clasificador SVM. Como en la fase anterior, se explican paso a paso las fases que componen esta etapa:

1. En primer lugar, se obtienen y ordenan los píxeles completos de la imagen original a partir del set de entrenamiento calculado en la etapa anterior.

2. Una vez se han cargado los valores de los píxeles, se comprueba si existe alguna característica – en nuestro caso banda – constante y, en caso de que la haya, se elimina. Este paso se realiza porque, desde el punto de vista del modelo de clasificación, una banda constante no aporta información útil.

3. Tras eliminar las características constantes, se comprueba que no existe ningún punto de entrenamiento repetido, es decir, dos píxeles con exactamente la misma composición. Esto se debe a que dos puntos de entrenamiento iguales tampoco aportan información útil al modelo de clasificación.

4. Se seleccionan los distintos parámetros con los que se realizará el entrenamiento, realizando la configuración del modelo que queremos utilizar – tipo de SVM, tipo de *kernel*, grado, coste, gamma, etc. En este caso, el sistema está configurado para el cálculo de un clasificador SVM con *kernel* lineal y coeficiente igual a 1.

5. Se realiza el entrenamiento del clasificador SVM y se obtiene el modelo de clasificación. Para realizar dicho entrenamiento se recurre a la librería *libSVM* codificada en lenguaje C++ y, en concreto, a una función denominada *svm_train* que, introduciendo las muestras del set de entrenamiento, la clase con la que se corresponde cada una y los parámetros con los que queremos configurar el clasificador, devuelve un modelo entrenado de SVM. El uso de esta librería tiene tres pasos básicos: convertir los datos de entrada de Matlab a C++, obtener el modelo de clasificación y traducir el resultado de C++ a Matlab.

Como resultado de esta fase obtenemos un modelo de SVM configurado para realizar la clasificación con un *kernel* lineal y un coeficiente igual a 1. Este procedimiento es válido para los cuatro porcentajes implicados en este algoritmo (1%, 3%, 5% y 10%).

3.2.3. Clasificación de la imagen

Una vez se ha obtenido el modelo de clasificación, el siguiente paso de nuestro algoritmo es realizar la clasificación de la imagen de entrada. Para ello, utilizando el modelo generado en la etapa anterior, se clasifican todos los píxeles de la imagen de entrada, realizando dicha clasificación de la imagen completa como un bloque.

Las entradas implicadas en esta fase son: la imagen original de entrada y el modelo entrenado de SVM. Dado que esta fase está compuesta por un único paso – llamar a la función *svm_predict* de la librería *libSVM* – el algoritmo es el más sencillo de todos y se basa en los mismos pasos que el entrenamiento anterior: transformar los datos de entrada (el modelo de clasificación) de Matlab a C++, realizar la predicción – o clasificación – de cada uno de los píxeles de la imagen y, por último, convertir el resultado del formato C++ al formato Matlab.

3.2.4. Cálculo de estadísticas

Tras finalizar la ejecución de cada uno de los cuarenta modelos de clasificación, se procede a analizar los resultados obtenidos en cada uno de los clasificadores por separado y comparar las diferencias entre uno y otro.

Las entradas de esta fase son: el vector con las clases originales a cada píxel de la imagen de entrada y el vector de clases calculado en la etapa anterior. En esta etapa se calculan las siguientes estadísticas (los valores obtenidos de dichas estadísticas serán los asociados a cada modelo de clasificación y, para cada porcentaje de entrenamiento, la media y la desviación estándar de las diez repeticiones):

1. *Precisión*: porcentaje de píxeles tumorosos correctamente clasificados sobre el total clasificado como tejido tumoroso.

2. *Sensibilidad*: porcentaje de píxeles sanos correctamente clasificados sobre el total clasificado como tejido sano.

3. *Precisión global* (*PG*): precisión de los píxeles correctamente clasificados sobre el total de píxeles.

4. *Precisión de clases* (*PC*): precisión de los píxeles correctamente clasificados para cada una de las clases.

5. *Precisión media de clases* (*PMC*): media del conjunto de precisiones de los píxeles correctamente clasificados para cada clase en cada iteración.

6. *Coeficiente Kappa de Cohen*[3]*(K)*: parámetro que tiene en cuenta la posible aleatoriedad de la clasificación y que, por tanto, nos ofrece un porcentaje estadístico más preciso.

A su vez, en paralelo al cálculo de los parámetros estadísticos, se almacenan tanto la matriz de confusión (matriz que resume la comparativa realizada entre el vector de entrada original y el resultado de la clasificación) como los tiempos de procesado para cada uno de los clasificadores, la media de dicho tiempo y su desviación estándar.

[3] Coeficiente Kappa de Cohen: http://en.wikipedia.org/wiki/Cohen%27s_kappa

3.3. Análisis funcional

Tras analizar en profundidad el algoritmo del clasificador SVM en la versión base de Matlab, podemos realizar un resumen de los distintos tipos de operaciones que nos encontramos durante la ejecución del mismo. Tras esto, se analizarán los diferentes puntos fuertes y débiles que presenta la implementación en Matlab para, de esta manera, poder proponer una solución que incremente la velocidad de procesado y poder alcanzar, de esta manera, el análisis en tiempo real.

3.3.1. Tipos de operación

Como conclusión del análisis de las cuatro etapas que componen el algoritmo, se puede extraer que las operaciones que lo conforman se agrupan en tres grupos principales:

- *Tratamiento de vectores y matrices*: uno de los pilares principales en los que se basa este algoritmo es el manejo de vectores y matrices. Este tipo de funciones predominan en las dos primeras etapas de procesado dado que, en ellas, se lleva a cabo el preprocesamiento de los datos de entrada: selección de los píxeles de entrenamiento (permutaciones y búsquedas dentro de un vector) y adecuación del set de entrenamiento (eliminación de bandas constantes y de muestras repetidas).

- *Utilización de la librería* libSVM: tanto en la segunda como en la tercera fase se utiliza esta librería codificada en C++. Para ello, es necesario utilizar funciones de conversión de datos de Matlab a C++ y viceversa.

- *Procesado estadístico de datos*: en la última etapa del algoritmo, la totalidad de la fase está dedicada a realizar un procesado estadístico de los resultados. Por tanto, el pilar fundamental de esta fase son funciones estadísticas sencillas: cálculo de una media o cálculo de una desviación estándar.

Una vez observado el tipo de funciones que componen la versión Matlab del algoritmo SVM, cabe destacar que está desarrollado para una ejecución secuencial. Esto es, una instrucción detrás de otra, pero, si observamos con atención el algoritmo, observamos que existen ciertas funcionalidades (como la clasificación píxel a píxel o la utilización de los diferentes modelos de clasificación) en las que, intuitivamente, se observa que existen un alto grado de paralelismo.

3.3.2. Ventajas e inconvenientes

Finalizado el análisis del algoritmo SVM codificado en Matlab – punto de partida del presente Trabajo de Investigación – desde un punto de vista estructural y funcional, se extraerán durante esta sección las principales ventajas e inconvenientes que presenta esta implementación.

En primer lugar, la principal ventaja que presenta la utilización de Matlab es que está optimizado para trabajar con algoritmos de procesado matemático y tratamiento de matrices y, por tanto, alcanza una velocidad de procesamiento muy alta.

En segundo lugar, si observamos el algoritmo, vemos que se ejecuta de manera secuencial sin embargo, intuitivamente, podemos extraer un gran paralelismo en ciertas fases:

- Durante la ejecución del algoritmo, se utilizan cuatro porcentajes de entrenamiento diferentes por lo que se podría realizar en paralelo la clasificación con cada uno de ellos.

- De cada porcentaje se calculan diez modelos de clasificación y, por tanto, se podría paralelizar el entrenamiento de cada modelo y la posterior clasificación.

- Una vez calculado el modelo de clasificación, se realiza la clasificación de toda la imagen transfiriendo la imagen completa y clasificándola como un bloque completo. Esta operación podría realizarse en paralelo si se dividiera la imagen en bloques y cada bloque se clasificase por separado.

Por último, cabe resaltar la utilización de una librería externa al entorno de programación Matlab, esta librería codificada en lenguaje C++ requiere la elaboración de un conversor de datos Matlab a C++ y viceversa. La utilización de una librería externa en el entorno Matlab y su correspondiente adaptación de entradas/salidas supone añadir un retardo importante si la cantidad de datos implicados con dicha librería es alta.

3.3.3. Alternativa

A lo largo de esta sección se propondrá una alternativa a la versión Matlab teniendo como objetivo el procesado de una imagen en tiempo real. Esta meta se corresponde con el objetivo principal del proyecto europeo HELICoiD: analizar una imagen hiperespectral en tiempo real. Dado que este Trabajo de Investigación se inscribe dentro de este proyecto, el procesado en tiempo real es uno de los principales objetivos de esta investigación.

Si tenemos en cuenta las conclusiones extraídas durante el análisis de la sección anterior, podemos extraer dos posibles puntos de mejoría para la velocidad de análisis: aumentar la velocidad de computación del algoritmo actual e implementar el algoritmo paralelizado.

Desde el punto de vista de aumentar la velocidad de computación, existen dos formas de incrementar este aspecto: incrementar la plataforma en la que se ejecuta el algoritmo o, en segundo lugar, realizar una implementación utilizando un nivel de programación menos abstracto que Matlab. En este proyecto se tiene como base la librería en C desarrollada en [1] y [2] por lo que, se podrían implementar funciones optimizadas para nuestro caso concreto.

Por otro lado, desde el punto de vista de la paralelización del algoritmo, existe una gran capacidad de mejoría. Sin embargo, la utilización de lenguajes de ejecución secuencial – pensados para ejecutar las aplicaciones desarrolladas en un único procesador – tales como C, C++, Java y, en este caso, Matlab, para implementar una aplicación multinúcleo requiere altos conocimientos sobre uso de hilos y comunicación entre procesadores.

A este respecto, existen lenguajes de programación denominados lenguajes basados en flujo de datos en los que, como se explica en el capítulo 2 de este mismo documento, se divide la implementación de una aplicación en dos vertientes: una parte estructural en la que se indican las dependencias de datos entre los bloques que componen el programa y, por otro lado, la parte funcional en la que se lleva a cabo la implementación del código que ejecutará cada uno de estos bloques.

Debido a que este proyecto busca una ejecución en tiempo real, se ha decidido realizar una implementación en un lenguaje que reduzca drásticamente la dificultad de realizar dicha paralelización. Para ello, se ha tomado la decisión de, en una primera aproximación, realizar una réplica del algoritmo base en el lenguaje de flujo de datos conocido como RVC – CAL.

La utilización de este lenguaje, como se explica en el capítulo 2 de este documento, nos aporta un nivel de abstracción superior a la implementación de la paralelización de la aplicación. En este lenguaje, la paralelización se realiza de manera gráfica indicando, a través de una interfaz gráfica, la dependencia de datos existente entre los distintos bloques del programa – denominados actores.

A su vez, otro de los puntos fuertes de este lenguaje es su capacidad de generar automáticamente en C con la comunicación entre actores ya implementada el código implementado, y, tras esto, ejecutar cada bloque en un procesador distinto.

Por último, como se ha dicho anteriormente en esta misma sección, gracias al estudio realizado en [1] y [2], ya existe una librería externa en la que se implementan funciones para procesado de imágenes hiperespectrales codificadas en C y que, incluso, utilizan a su vez funciones contenidas en una librería especializada codificada en C++. Estas dos ideas nos permiten incluir a dicha librería las funciones necesarias para implementar el clasificador SVM y, a su vez, utilizar la librería *libSVM* implicada en el proceso de análisis de la versión Matlab.

3.4. Adaptación a RVC – CAL

Como resultado del análisis estructural y funcional del algoritmo implementado en lenguaje Matlab, se ha llegado a la conclusión de que, con el fin de alcanzar el procesado en tiempo real de una imagen hiperespectral, es necesario realizar una paralelización del algoritmo de partida. Para ello, en la sección anterior se ha decidido realizar una adaptación del algoritmo al lenguaje RVC – CAL. Este nuevo lenguaje facilita la paralelización ya que dicha paralelización se realiza de manera explícita – se lleva a cabo gráficamente creando una *network* en la que se interconectan diferentes actores –. Esta adaptación se realizará en tres pasos:

1. *Conversión de funciones*: durante esta etapa se realizará la codificación en C de las distintas funciones que componen el algoritmo del clasificador SVM.

2. *Implementación secuencial*: con el fin de comprobar cómo afecta la adaptación al comportamiento del algoritmo se realiza una implementación secuencial del algoritmo. En esta fase, a su vez, se analizará el rendimiento del algoritmo cuando se implementa fuera de un entorno software especializado en procesado matemático.

3. *Paralelización del algoritmo*: finalmente, una vez se haya comprobado que la adaptación al lenguaje RVC – CAL es viable y se tenga una versión base en dicho lenguaje, se realizará la paralelización del algoritmo y se testearán las diferentes configuraciones con el fin de optimizar al máximo el tiempo de procesado y alcanzar, en la medida de lo posible, el procesado en tiempo real.

Durante este Trabajo de Investigación se han llevado a cabo las dos primeras etapas de la adaptación al lenguaje RVC – CAL solventando los problemas de traducción entre lenguajes y desarrollando todas las funciones necesarias para realizar implementaciones secuenciales o paralelizadas. De esta manera, se plantea como línea de trabajo futura la paralelización del clasificador. Esta paralelización se realizaría de manera totalmente independiente a la versión base de este Trabajo de Investigación – Matlab – y, por tanto, se afrontaría como un problema de paralelización y no como uno de conversión entre lenguajes.

3.4.1. Conversión de funciones

Para esta fase de la conversión entre lenguajes, se ha seguido secuencialmente la implementación Matlab y se ha replicado el funcionamiento de cada una de las funciones encontradas en lenguaje C. Durante esta sección, como se ha procedido con anterioridad, se explican por separado las funciones desarrolladas para cada etapa del algoritmo.

Lectura de los datos de entrada del sistema

Durante este paso, se incorpora al sistema la imagen de entrada – en forma de matriz – y el vector de clases. Para ello se utiliza una función ya incluida en la librería utilizada en RVC – CAL. Esta función se denomina *read_txt_c* y lee un fichero de texto almacenado en el disco duro.

Selección del set de entrenamiento

Una vez tenemos las entradas disponibles, se calcula el set de entrenamiento para cada uno de los modelos de clasificación. Como ya hemos explicado con anterioridad, este cálculo está compuesto de cuatro pasos. A continuación, se muestran las funciones que se han desarrollado para cada uno de ellos:

1. Cálculo del número de clases: durante este paso, se ha implementado una función que busque el valor máximo contenido en un vector. Esta función se ha denominado *vector_find_max*.

2. Obtención aleatoria del número ideal de píxeles de cada clase: en este caso, el proceso se ha dividido en tres etapas:

    a. Extracción de los píxeles de cada clase: se implementa una función que, indicándole el valor asociado a una clase (en nuestro caso 1 para tejido sano, 2 para tejido tumoroso y 3 para tejido necrosado) devuelve un vector que contiene los índices que ocupan cada uno de los píxeles de dicha clase en la imagen original. Esta función se ha denominado *vector_find_equals*.

    b. Permutación de los píxeles de cada clase: el objetivo es ordenar de manera aleatoria los índices de cada clase y, por tanto, ser capaces de obtener un número concreto de índices de cada clase también de manera aleatoria. Para ello, se construye la función denominada *vector_permutation*.

c. En último lugar, siguiendo el orden que presenta el vector permutado, es decir, orden aleatorio con respecto al original, se extraen tantos índices como indique la entrada utilizada para señalar el número ideal de muestras de cada clase.

3. Búsqueda de píxeles no repetidos: en el caso de que no se alcance el número óptimo de índices para cada clase, se realiza una búsqueda aleatoria de los índices restantes para completar el set de entrenamiento. Para realizar esta operación, en primer lugar se ha desarrollado una función que elimina los índices ya seleccionados del vector de clases de entrada: *vector_remove_indexes*; posteriormente, se realiza una permutación del vector resultante utilizando de nuevo la función *vector_permutation*.

4. Construcción del set de entrenamiento: para este último paso, se ha implementado una función para añadir valores a un vector a partir de una posición determinada: *vector_concatenation.* La salida se compone utilizando los índices calculados durante los pasos anteriores.

Entrenamiento del clasificador SVM

Tras obtener el set de entrenamiento, el siguiente paso que hay que llevar a cabo es el entrenamiento del clasificador SVM. En este caso, se exponen las funciones desarrolladas para completar los cincos pasos que componen esta fase:

1. Composición de la matriz de entrenamiento: en primer lugar, para comenzar el entrenamiento, se compone el set de entrenamiento utilizando el vector de índices calculado en la etapa anterior. Este proceso se ha dividido en dos etapas:

   a. En primer lugar, se ordena de menor a mayor el vector con los índices de los píxeles que compondrán el set de entrenamiento utilizando la función *vector_sort*.

   b. Tras esto, se compone el set de entrenamiento extrayendo cada uno de los píxeles de la imagen original y almacenándolos en una matriz de manera traspuesta. Para realizar este paso, se han utilizado las funciones *matrix_get_row* y *matrix_get_column* ya codificadas en la librería utilizada.

2. Eliminación de las bandas constantes: en este caso, la adaptación se realiza en dos pasos. El primer paso es realizar una búsqueda de las filas constantes y devolver un vector con las posiciones de las filas constantes utilizando la función *matrix_find_constant_rows*. Una vez encontradas dichas filas, se procede a la eliminación de las bandas en el set de entrenamiento y en la matriz original. Para ello, se utiliza la función *matrix_remove_rows_by_indexes* para la matriz del set de entrenamiento y la función *matrix_remove_columns_by_indexes* para la imagen original ya que, para la primera, las bandas se corresponden con las filas y, para la imagen original, con las columnas.

3. Sustracción de las muestras repetidas: durante este paso se realizan varias operaciones. En primer lugar se traspone el set de entrenamiento para adaptarlo a la entrada necesaria para realizar el entrenamiento utilizando la función *transpose* ya implementada en la librería. A continuación, se ordena por filas de menor a mayor el set de entrenamiento utilizando dos funciones: *matrix_get_sorted_row_indexes* para obtener un vector con las posiciones ordenadas de dicha forma y, tras esto, se aplica la función *matrix_reorder_by_row_indexes* para reordenar el set de entrenamiento de esta manera. Para finalizar, se aplica la función *matrix_find_equal_rows* para eliminar las filas repetidas. El reordenamiento se realiza para facilitar tanto la búsqueda de filas repetidas como el entrenamiento del clasificador.

4. Selección de los parámetros de configuración del clasificador: este paso ha sido obviado en la adaptación del algoritmo ya que, como se explicará a continuación, la función implementada para realizar el entrenamiento del clasificador permite elegir los parámetros de configuración del clasificador.

5. Entrenamiento del clasificador: para realizar el entrenamiento del clasificador con *kernel* lineal, se ha implementado la función *svm_train_linear*. Esta función adapta el set de entrenamiento a la estructura de *libSVM*, configura los parámetros del *kernel*, realiza el entrenamiento del clasificador SVM y, por último, convierte el modelo de clasificación generado al lenguaje RVC – CAL.

Clasificación de la imagen

Partiendo del modelo de clasificación generado anteriormente, se realiza la clasificación de la imagen original utilizando la función *svm_predict_linear*. Esta función, está implementada de tal manera que, en primer lugar convierte el modelo de clasificación del formato RVC – CAL a la estructura de la librería *libSVM* y, posteriormente, realiza la clasificación de los píxeles que se le indiquen en la entrada uno por uno. Como salida de la función tenemos un vector que indica la clase que se le ha asignado a cada píxel.

Cálculo de estadísticas

Una vez se ha realizado la clasificación, se procede a realizar el cálculo de las estadísticas para comprobar la bondad de los resultados de dicha clasificación. En esta fase se han utilizado tres funciones:

1. *get_confusion*: con esta función obtenemos la matriz de confusión y, aprovechando dicha matriz, se calculan las diferentes precisiones de cada clasificación y, a su vez, el parámetro Kappa de Cohen.

2. *mean_vector*: esta función ya estaba implementada en la librería y su función es calcular la media de los valores de un vector de entrada.

3. *std_vector*: el objetivo de esta función es calcular la desviación estándar de los valores contenidos en un vector de entrada.

Escritura de los resultados

Finalmente, tras finalizar el algoritmo, se realiza la escritura de los resultados. Los resultados se almacenan en formato de matriz y, dicha matriz, está formada por los datos obtenidos con los diez modelos de clasificación correspondientes a cada porcentaje de entrenamiento. Se almacenan cinco matrices de resultados: los píxeles con los que se realiza el entrenamiento de cada modelo de clasificación, las clases obtenidos durante la clasificación, las matrices de confusión, las estadísticas medias y las desviaciones estándar, las estadísticas de cada clasificación y la precisión con la que se ha clasificado cada clase en cada iteración. Por otro lado, se realiza la escritura de los tiempos de cada entrenamiento, cada clasificación y el tiempo total invertido en preprocesar los datos, entrenar y clasificar. Para realizar las escrituras se utilizará una función ya codificada en la librería denominada *write_txt_c*.

A modo de resumen, en la tabla 3.I se recogen las funciones explicadas durante esta sección indicando si ya existían antes del desarrollo de este Trabajo de Investigación o no, y recordando su funcionalidad:

Nombre	Nueva	Funcionalidad
read_txt_c	No	Lee un fichero txt almacenado en disco
vector_find_max	Sí	Busca el valor máximo de un vector
vector_find_equals	Sí	Busca los índices en los que se encuentra un valor indicado dentro de un vector
vector_permutation	Sí	Reordena aleatoriamente las posiciones de cada elemento de un vector
vector_remove_indexes	Sí	Se eliminan de un vector las posiciones que se indiquen
vector_concatenation	Sí	Concatena dos vectores
vector_sort	Sí	Ordena de menor a mayor un vector
matrix_get_row	No	Devuelve una fila de una matriz
matrix_get_column	No	Devuelve una columna de una matriz
matrix_find_constant_row	Sí	Busca en una matriz filas constantes
matrix_remove_rows_by_indexes	Sí	Elimina las filas de una matriz que le indiquen
matrix_remove_columns_by_indexes	Sí	Elimina las columnas de una matriz que le indiquen
transpose	No	Traspone una matriz
matrix_get_sorted_row_indexes	Sí	Ordena una matriz por filas de menor a mayor
matrix_reorder_by_row_indexes	Sí	Reordena una matriz siguiendo un vector patrón de orden de filas
matrix_find_equal_rows	Sí	Busca en una matriz filas repetidas
svm_train_linear	Sí	Realiza el entrenamiento con *kernel* lineal de un clasificador SVM
svm_predict_linear	Sí	Realiza la clasificación con *kernel* lineal de un clasificador SVM
get_confusion	Sí	Calcula la matriz de confusión y las precisiones de un conjunto clasificado
mean_vector	No	Obtiene la media de los elementos que componen un vector
std_vector	Sí	Obtiene la desviación estándar de los elementos que componen un vector
write_txt_c	No	Escribe un fichero txt y lo almacena en disco

Tabla 3.I. Resumen de las funciones utilizadas

Para concluir, cabe destacar que se han codificado tres funciones extra (*vector_build_by_position*, *vector_fill_by_position* y *vector_fill_with_value*) para completar la adaptación. Tras finalizar esta etapa se han añadido un total de 19 funciones a la librería.

3.4.2. Implementación secuencial

Durante esta sección se explicarán las decisiones tomadas para replicar la implementación secuencial original del algoritmo Matlab en su versión RVC – CAL. Para completar este proceso se han utilizado las funciones desarrolladas en el apartado anterior y se ha adaptado paso a paso el algoritmo base.

En primer lugar, como se observa en la figura 3.1, se ha dividido el algoritmo en cuatro acciones. La primera de ellas es la lectura de datos, la segunda el cálculo de los índices con los que se compondrán los diferentes sets de entrenamiento, en tercer lugar se realizará el entrenamiento, la clasificación y el cálculo de estadísticas para cada uno de los modelos de clasificación y, por último, se guardan los datos obtenidos en cada una de las cuarenta clasificaciones.

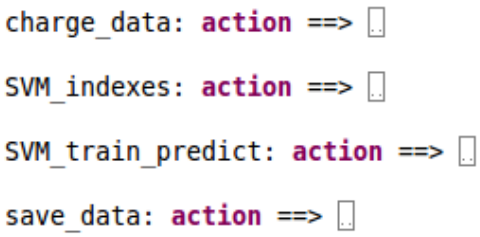

Figura 3.1.- Secuencia de acciones

Estas acciones se ejecutan de manera secuencial, es decir, una acción se ejecuta cuando la anterior finaliza. A continuación, se explica cómo se ha implementado cada una de las fases:

- *charge_data*: durante esta etapa se realiza la lectura del fichero de texto que contiene la imagen original y el vector que contiene la clase con la que se corresponde cada uno de los píxeles.

- *SVM_indexes*: en primer lugar, se realiza la extracción de los índices de cada una de las clases (operación común a todos los sets de entrenamiento que se utilizarán posteriormente). Tras esto, se repite en cuatro bloques consecutivos el código necesario para calcular los clasificadores de cada porcentaje (primero el 1%, después el 3%, posteriormente el 5% y, por último, el 10%). Para cada uno de estos bloques, se calculan diez sets de entrenamiento y, dado que el código es el mismo para cada repetición, se implementa utilizando una sentencia *foreach*.

- *SVM_train_predict*: en esta acción, como ocurría en la anterior, se repiten cuatro bloques que contienen la misma secuencia de acciones pero modificadas de acuerdo al tamaño concreto de cada uno de los sets de entrenamiento. Dentro de estos bloques, se utiliza, también como en la acción *SVM_indexes*, una sentencia *foreach* para realizar cada uno de los diez procesados correspondientes a cada porcentaje. Dentro de este *foreach* se implementa el algoritmo correspondiente al preprocesado del set de entrenamiento, el entrenamiento del clasificador SVM, la clasificación de la imagen utilizando un clasificador con *kernel* lineal y coeficiente igual a 1 y el cálculo individual de estadísticas. Tras llevar a cabo el procesamiento de los diez modelos de clasificación de cada porcentaje, se calculan los valores medios y las desviaciones estándar de las estadísticas obtenidas para cada bloque.

- *save_data*: por último, también para cada bloque de diez modelos de clasificación, se guardan los datos obtenidos durante la ejecución del sistema completo (estadísticas, índices utilizados, resultados de clasificación y tiempos de ejecución).

Tras completar esta etapa de la adaptación, nos encontramos en la situación de tener el algoritmo base replicado en RVC – CAL. Para probar que este algoritmo ha sido correctamente traducido, durante el capítulo siguiente se realizará un estudio que compruebe si los resultados obtenidos con ambas versiones coinciden y, tras realizar dicho análisis, podremos concluir si se ha alcanzado el objetivo de adaptar correctamente el algoritmo a dicho lenguaje. Por otro lado, se realizará un test de velocidad de procesado para comprobar el rendimiento del clasificador SVM con *kernel* lineal fuera de un entorno de procesado matemático.

4 ANÁLISIS DE RESULTADOS

4.1. Estructura

A lo largo de este capítulo, se resaltarán los principales resultados obtenidos durante el desarrollo del presente Trabajo de Investigación. Este análisis, como se mencionó en el capítulo anterior, se centra en comprobar si, al realizar la conversión entre lenguajes, existe alguna incompatibilidad y comparar el rendimiento entre ambas versiones.

Con el fin de facilitar la comprensión de los resultados arrojados por la aplicación desarrollada, se ha decidido dividir el presente capítulo en tres secciones:

- *Estudio de la conversión*: en esta sección, se ha replicado paso a paso el algoritmo implementado en Matlab. Para ello, se ha dividido el procedimiento en tres pasos: en primer lugar, se construye la parte del cálculo aleatorio del set de entrenamiento, posteriormente, se implementa el algoritmo utilizando un único modelo de clasificación (con un 1% de set de entrenamiento – el primero utilizado por Matlab) y, una vez se ha conseguido que éste funcione, se codifica el modelo secuencial con los cuarenta clasificadores incluidos.

- *Estudio de la precisión*: tras implementar el algoritmo completo – con sus cuarenta modelos de clasificación –, se estudian las distintas estadísticas arrojadas por cada uno de ellos y se estudian las diferencias existentes entre la versión Matlab y la versión RVC – CAL.

- *Estudio de tiempos de ejecución*: una vez hemos visto si la adaptación al lenguaje RVC – CAL se ha realizado correctamente, se lleva a cabo un estudio del impacto en el tiempo de ejecución que conlleva el paso de un lenguaje optimizado para cálculo matemático a un lenguaje optimizado para facilitar la paralelización. En consecuencia, se estudiará el impacto del cambio de lenguajes en el tiempo de procesado.

Cabe mencionar que las pruebas contenidas en el presente capítulo se han recogido utilizando un ordenador con un procesador *Intel Core i7* funcionando a 2.6 GHz; para la versión Matlab se ha utilizado Windows 8.1 de 64 bits y la versión R2014a de Matlab y, para la versión RVC – CAL, se ha utilizado Ubuntu 12.04 de 64 bits. Asimismo, la imagen utilizada para realizar el análisis tiene un total de 18580 píxeles y 1040 bandas.

4.2. Estudio de la conversión

Como se ha comentado en la introducción de este capítulo, durante esta sección se explicará cómo se ha ido construyendo paso a paso el algoritmo secuencial del clasificador SVM implementado en Matlab en su versión RVC – CAL.

El primer paso de este proceso ha sido la extracción de los diferentes sets de entrenamiento. Para ello, se ha comprobado que los vectores que contienen las posiciones de cada una de las clases se permutan correctamente. Tras observar que esta etapa funciona como se espera, se han extraído los píxeles utilizados en el primer caso del entrenamiento con 1% de la versión Matlab y se han incluido en la versión RVC – CAL.

Utilizando el primer modelo de clasificación como base, se han implementado paso a paso las etapas necesarias para completar el algoritmo: preprocesado, entrenamiento del modelo de clasificación y clasificación de la imagen utilizando el modelo desarrollado.

Para comprobar el funcionamiento de cada una de las funciones desarrolladas, se ha ejecutado paso a paso tanto la versión Matlab como la versión RVC – CAL y se ha realizado una comparativa de resultados. De esta manera, se ha observado que únicamente en un paso se obtienen resultados ligeramente diferentes en la versión desarrollada durante este Trabajo de Investigación.

El paso con el que obtenemos resultados diferentes es el de reordenamiento de menor a mayor las filas de la matriz de entrenamiento en combinación con la eliminación de filas repetidas. Se ha comprobado que en ambas versiones las filas que se mantienen son las mismas pero, la versión Matlab, no devuelve la matriz totalmente ordenada de menor a mayor. Por otro lado, al continuar con la ejecución del algoritmo y realizar el entrenamiento, el modelo de entrenamiento generado en ambas versiones es exactamente igual. Por tanto, se ha llegado a la conclusión de que la repercusión que tiene esta diferencia no alterará en gran medida los modelos de clasificación generados por la librería *libSVM*.

Para finalizar, tras obtener exactamente los mismos resultados para el primer modelo de clasificación del conjunto de 1% de entrenamiento, se han replicado los algoritmos para cada porcentaje y se han incluido las sentencias *foreach* necesarias para generar los cuarenta modelos de clasificación.

A continuación, se analizarán la precisión (bondad de resultados) de este tipo de clasificador (en ambas versiones) y el tiempo de procesado necesario para realizar las cuarenta clasificaciones en ambos casos.

4.3. Estudio de la precisión

Durante este estudio, se realizará una comparativa entre ambas versiones para comprobar si, como se dijo en la sección anterior, la diferencia en la función descrita anteriormente para cada versión tiene alguna repercusión en el comportamiento de la nueva implementación.

Para realizar dicho estudio, se compararán los cuarenta modelos de clasificación que se generan durante la ejecución del algoritmo. Dicha comparación se basa en observar la variación de los diferentes parámetros estadísticos que se han calculado para cada una de las versiones. A su vez, se comprobará si el modelo de clasificación generado procesa la imagen hiperespectral de manera satisfactoria.

Durante este estudio se aportarán una serie de tablas con un código de colores que indicará si RVC – CAL está por encima, iguala o está por debajo – en verde, azul y rojo respectivamente – de la versión implementada en Matlab.

En primer lugar, si analizamos únicamente las muestras de tejido sano y de tejido tumoroso y, en consecuencia, obviamos los píxeles clasificados como tejido necrosado, podemos extraer dos parámetros estadísticos:

- Por un lado, se obtiene la precisión del algoritmo (porcentaje de píxeles tumorosos bien clasificados sobre el total de píxeles clasificados como tumorosos). En este caso, podemos observar en la tabla 4.I que los resultados en ambas versiones son equivalentes y, a su vez, que siempre se mantiene por encima del 99% sobre el total. Por esto, podemos concluir que el algoritmo ha sido correctamente replicado en cuanto a precisión se refiere y que mantiene un alto porcentaje de tejido tumoroso bien clasificado.
- Por otro lado, se encuentra la sensibilidad del algoritmo (porcentaje de píxeles sanos bien clasificados sobre el total clasificado como sano). En la tabla 4.II se muestran los resultados obtenidos para este parámetro. Como podemos observar, prácticamente la totalidad de los casos se encuentran por encima de 95%. También se observa que los resultados obtenidos para ambas versiones son prácticamente idénticos y, por tanto, como en el caso anterior, podemos concluir que el algoritmo funciona correctamente y que se mantiene un alto porcentaje de tejido sano bien clasificado.

Por último, cabe destacar que existen varios casos (con un porcentaje de entrenamiento del 10%) en el que se clasifica correctamente la totalidad de los píxeles tumorosos o de los píxeles sanos y, a su vez, para el 10% de entrenamiento, todas las clasificaciones tienen un porcentaje superior al 99% de precisión y sensibilidad.

	1%		3%		5%		10%	
	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL
1	0.9961	0.9961	0.9950	0.9950	0.9998	0.9998	0.9996	0.9996
2	0.9972	0.9972	0.9997	0.9997	0.9998	0.9997	0.9996	0.9996
3	0.9924	0.9924	0.9996	0.9996	0.9992	0.9992	0.9999	0.9999
4	0.9936	0.9980	0.9995	0.9995	0.9998	0.9998	0.9994	0.9994
5	0.9979	0.9979	0.9991	0.9991	0.9999	0.9999	0.9998	0.9998
6	0.9994	0.9992	0.9999	0.9999	0.9996	0.9996	0.9995	0.9995
7	0.9991	0.9997	0.9969	0.9969	0.9999	0.9999	0.9993	0.9993
8	0.9909	0.9918	0.9989	0.9989	0.9995	0.9995	0.9998	0.9998
9	0.9953	0.9953	0.9975	0.9975	0.9994	0.9996	1	1
10	0.9880	0.9880	0.9983	0.9983	0.9999	0.9999	1	1
MAX	0.9994	0.9997	0.9999	0.9999	0.9999	0.9999	1	1
Media	0.9950	0.9956	0.9984	0.9984	0.9997	0.9997	0.9997	0.9997
STD	0.0037	0.0036	0.0015	0.0015	0.0002	0.0002	0.0002	0.0002
MIN	0.9880	0.9880	0.9950	0.9950	0.9992	0.9992	0.9993	0.9993

Tabla 4.I. Comparativa de precisión

	1%		3%		5%		10%	
	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL
1	0.9840	0.9840	0.9853	0.9840	0.9909	0.9909	0.9966	1
2	0.9850	0.9848	0.9896	0.9916	0.9950	0.9990	0.9971	0.9982
3	0.9792	0.9792	0.9825	0.9825	0.9853	0.9853	0.9979	0.9992
4	0.9864	0.9799	0.9794	0.9794	0.9881	0.9881	0.9964	0.9982
5	0.9579	0.9579	0.9868	0.9868	0.9827	0.9827	0.9946	0.9984
6	0.9771	0.9939	0.9853	0.9928	0.9931	0.9931	0.9982	0.9961
7	0.9472	0.9472	0.9939	0.9939	0.9844	0.9844	0.9997	0.9894
8	0.9928	0.9890	0.9900	0.9900	0.9851	0.9851	0.9987	0.9987
9	0.9648	0.9648	0.9914	0.9914	0.9978	0.9952	0.9974	0.9974
10	0.9491	0.9491	0.9909	0.9909	0.9804	0.9804	0.9959	0.9987
MAX	0.9928	0.9939	0.9939	0.9939	0.9978	0.9990	0.9997	1
Media	0.9724	0.9730	0.9875	0.9883	0.9883	0.9884	0.9973	0.9974
STD	0.0164	0.0160	0.0045	0.0046	0.0057	0.0057	0.0015	0.0028
MIN	0.9472	0.9472	0.9794	0.9794	0.9804	0.9804	0.9946	0.9894

Tabla 4.II. Comparativa de sensibilidad

Por otro lado, si tenemos en cuenta las tres clases implicadas en la clasificación (tejido sano, tumoroso y necrosado), obtenemos los siguientes dos parámetros:

- En la tabla 4.III se puede observar la precisión global (porcentaje de píxeles bien clasificados dentro del total de píxeles). En dicha tabla, de nuevo, basándonos en la cercanía de los resultados de ambas versiones, se corrobora que la adaptación de lenguajes es correcta. A su vez, si observamos los resultados obtenidos, comprobamos que para casi la totalidad de los casos, se ha clasificado correctamente como mínimo el 90% de los píxeles de la imagen.
- En segundo lugar, si observamos el coeficiente Kappa de Cohen mostrado en la tabla 4.IV (el cual elimina la parte aleatoria que puede existir durante la clasificación), podemos concluir que la concordancia entre ambas versiones se mantiene para todos los parámetros que hemos calculado y, por tanto, que ambas versiones son equivalentes. A su vez, este parámetro es el más fiable para saber si la clasificación se realiza correctamente y si observamos los resultados obtenidos, vemos que, en el peor de los casos, obtenemos un valor superior a un 81%.

	1%		3%		5%		10%	
	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL
1	0.9294	0.9294	0.9504	0.9504	0.9592	0.9592	0.9781	0.9781
2	0.9248	0.9246	0.9519	0.9520	0.9649	0.9658	0.9752	0.9752
3	0.9341	0.9340	0.9566	0.9566	0.9563	0.9563	0.9749	0.9752
4	0.9177	0.9185	0.9533	0.9527	0.9609	0.9609	0.9727	0.9731
5	0.9105	0.9077	0.9551	0.9551	0.9570	0.9560	0.9773	0.9773
6	0.8901	0.8940	0.9576	0.9575	0.9658	0.9658	0.9724	0.9724
7	0.9136	0.9102	0.9607	0.9607	0.9665	0.9665	0.9738	0.9738
8	0.9073	0.9068	0.9592	0.9592	0.9666	0.9666	0.9809	0.9809
9	0.9195	0.9195	0.9946	0.9446	0.9616	0.9611	0.9727	0.9727
10	0.9212	0.9212	0.9516	0.9516	0.9646	0.9646	0.9768	0.9774
MAX	0.9341	0.9340	0.9946	0.9607	0.9666	0.9666	0.9809	0.9809
Media	0.9168	0.9166	0.9541	0.9540	0.9623	0.9624	0.9755	0.9756
STD	0.0125	0.0114	0.0048	0.0045	0.0039	0.0038	0.0028	0.0026
MIN	0.8901	0.8940	0.9504	0.9446	0.9570	0.9560	0.9724	0.9727

Tabla 4.III. Comparativa de la precisión global

	1%		3%		5%		10%	
	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL
1	0.8795	0.8795	0.9140	0.9141	0.9283	0.9283	0.9605	0.9605
2	0.8726	0.8723	0.9170	0.9171	0.9385	0.9401	0.9553	0.9552
3	0.8868	0.8868	0.9248	0.9248	0.9238	0.9238	0.9548	0.9553
4	0.8588	0.8605	0.9190	0.9180	0.9315	0.9315	0.9508	0.9515
5	0.8487	0.8443	0.9223	0.9223	0.9250	0.9250	0.9590	0.9590
6	0.8185	0.8246	0.9263	0.9262	0.9397	0.9397	0.9505	0.9504
7	0.8547	0.8495	0.9315	0.9315	0.9411	0.9411	0.9529	0.9529
8	0.8429	0.8422	0.9289	0.9289	0.9409	0.9410	0.9655	0.9655
9	0.8630	0.8630	0.9044	0.9044	0.9326	0.9318	0.9510	0.9510
10	0.8641	0.8641	0.9161	0.9161	0.9379	0.9379	0.9582	0.9592
MAX	0.8868	0.8868	0.9315	0.9315	0.9411	0.9411	0.9655	0.9655
Media	0.8590	0.8587	0.9204	0.9203	0.9339	0.9340	0.9558	0.9560
STD	0.0196	0.0179	0.0080	0.0076	0.0066	0.0064	0.0049	0.0047
MIN	0.8185	0.8246	0.9044	0.9044	0.9238	0.9250	0.9505	0.9504

Tabla 4.IV. Comparativa del Coeficiente Kappa de Cohen

Tras finalizar el análisis de la precisión de los resultados obtenidos, como conclusiones finales de este análisis, se extraen las siguientes ideas:

- Con respecto a los porcentajes de entrenamiento, se observa que un aumento en el número de píxeles utilizados para realizar el entrenamiento del modelo de clasificación conlleva un aumento en las prestaciones del clasificador utilizado.

- En el mejor de los casos, utilizando un 10% de píxeles para realizar el entrenamiento, se ha llegado a obtener una clasificación totalmente correcta para los píxeles tumorosos y para los píxeles sanos y, a su vez, se ha obtenido un coeficiente Kappa de Cohen superior al 96%. Estos resultados demuestran, que la capacidad que se había predicho del clasificador SVM para realizar el procesamiento de las imágenes hiperespectrales se mantiene para nuestro caso.

- La versión secuencial implementada en lenguaje RVC – CAL tiene un funcionamiento equivalente a la implementada en lenguaje Matlab. Esta idea queda claramente reflejada de manera gráfica en las figuras 4.1, 4.2, 4.3 y 4.4 que se adjuntan al final del capítulo.

4.4. Estudio de tiempos de ejecución

Tras verificar el correcto funcionamiento de la versión traducida a RVC – CAL del clasificador SVM, durante este apartado se llevará a cabo un análisis del tiempo de ejecución de dicho algoritmo. Durante este estudio, se realizará una comparativa del tiempo total que tarda cada modelo de clasificación en realizar todo el procesamiento para cada una de las versiones. Con esto, se pretende estudiar el impacto en el tiempo de procesado que tiene convertir el algoritmo SVM de la versión Matlab a la versión RVC – CAL.

Tras realizar este primer análisis, se estudian por separado los tiempos implicados en cada una de las etapas de la clasificación (preprocesado, entrenamiento y clasificación) con el fin de comprobar si existe algún cuello de botella y, en el caso de que exista, si se puede realizar una paralelización que acelere dicha etapa.

En primer lugar, en la tabla 4.V, se puede observar la comparativa del tiempo total realizada para ambas versiones. De esta comparativa extraemos que, tras realizar la conversión de lenguajes, se ha logrado un *speedup* (incremento en la velocidad de procesado) de casi el 30% excepto para el 1% que ha sido del 20%.

	1%		3%		5%		10%	
	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL	Matlab	RVC – CAL
1	10.9886	8.9109	34.4053	21.4040	40.7761	28.0997	63.6823	46.1352
2	13.2888	8.7177	31.1133	19.9360	43.4852	27.9248	65.4445	46.2531
3	12.1217	9.2051	30.9133	19.4105	42.0487	29.0429	61.2963	44.2227
4	11.2704	8.3204	32.1833	20.7405	40.9590	28.9222	63.3844	45.8483
5	11.4526	8.1411	26.6817	17.6797	41.8262	27.9263	66.9320	46.5651
6	10.4951	8.8151	28.6893	19.3407	45.9752	28.4350	70.0252	46.7550
7	10.0551	9.0235	30.5081	20.4607	41.3368	27.9367	62.4512	44.5417
8	9.4497	8.4625	30.3199	20.2049	45.8116	28.8755	66.4315	48.0717
9	10.4995	9.1904	27.9321	18.4422	43.2684	28.1968	59.4825	43.0002
10	10.1298	8.7424	31.3803	20.5499	37.5908	25.8208	64.4702	46.6358
TOTAL	109.7514	87.5307	304.1266	198.1707	423.1171	281.1821	643.5999	458.0305
MAX	12.1217	9.2051	34.4053	21.4040	45.9752	29.0429	70.0252	48.0717
Media	10.9751	8.7530	30.4127	19.8170	42.3117	28.1182	64.3600	45.8030
STD	1.1219	0.3384	2.2064	1.0648	2.4872	0.8699	3.0262	1.3995
MIN	9.4497	8.1411	26.6817	17.6797	37.5908	25.8208	59.4825	43.0002

Tabla 4.V. Comparativa de tiempos de procesado total (en segundos)

Una vez se ha comparado el tiempo total y se ha demostrado que la conversión a este nuevo lenguaje aporta una mejora en el tiempo de procesado, se realiza un análisis del tiempo de ejecución de cada etapa del procesado. Este análisis se llevará a cabo utilizando la versión traducida a RVC – CAL desarrollada durante el presente Trabajo de Investigación.

Para realizar este análisis se ha dividido el algoritmo de cada modelo de clasificación en tres etapas: preprocesado (F1), entrenamiento (F2) y clasificación (F3). Este análisis está recogido en la tabla 4.VI y, de ella, podemos extraer las siguientes tres ideas:

- La etapa de preprocesado implicada en la generación del modelo de clasificación tiene un tiempo de ejecución despreciable comparado con las otras dos etapas.

- La fase de entrenamiento incrementa su tiempo de ejecución conforme aumenta el tamaño del set de entrenamiento. Este incremento es exponencial y, dado que recurrimos a la librería especializada *libSVM* para realizar dicho entrenamiento, no es posible realizar una paralelización del algoritmo que entrena al clasificador y genera el modelo de clasificación. Por esto, esta etapa únicamente se puede realizar de manera secuencial ya que, por el momento, es necesario utilizar el set de entrenamiento como un único bloque para generar el modelo de clasificación. Pero, si nos basamos en los datos de la tabla 4.VI, observamos que esta etapa consume desde el 5% en el caso del 1% hasta el 20% en el caso del 10% de entrenamiento.

- Por último, la etapa de clasificación es la que más tiempo consume de todas – desde el 95% en el caso del 1% de entrenamiento hasta el 80% en el caso del 10%–. Como se ha mencionado con anterioridad, en la implementación base la clasificación de la imagen se realiza píxel a píxel y utilizando un único bloque que engloba la imagen completa. Esta etapa puede ser fácilmente paralelizable si, utilizando el mismo modelo de clasificación, dividimos la imagen original en bloques más pequeños y clasificamos cada bloque utilizando un núcleo diferente. Con esta idea, se podría reducir el tiempo de procesado de esta etapa y, en consecuencia, reducir el tiempo de ejecución del principal cuello de botella que nos encontramos al utilizar este método de procesado de imágenes hiperespectrales.

	1%			3%			5%			10%		
	F1	F2	F3	F1	F2	F3	F1	F2	F3	F1	F2	F3
1	0.1843	0.3008	8.4254	0.2037	1.8516	19.3483	0.2250	3.0855	24.7899	0.3027	8.9531	36.8791
2	0.1144	0.2968	8.3062	0.1315	1.9085	17.8957	0.1483	2.6948	25.0814	0.2252	8.8905	37.1371
3	0.1153	0.3011	8.7884	0.1303	1.3317	17.9481	0.1493	3.3337	25.5596	0.2262	7.9522	36.0439
4	0.1153	0.2232	7.9815	0.1305	1.7956	18.8141	0.1487	3.4041	25.3691	0.2234	8.1283	37.4962
5	0.1156	0.2509	7.7743	0.1301	1.2615	16.2877	0.1482	2.8695	24.9083	0.2221	8.6842	37.6584
6	0.1152	0.2722	8.4276	0.1309	1.4208	17.7887	0.1492	2.8602	25.4253	0.2248	8.4986	38.0313
7	0.1151	0.3672	8.5408	0.1302	1.4084	18.9218	0.1486	2.6703	25.1174	0.2549	8.0200	36.2666
8	0.1160	0.3874	7.9588	0.1304	1.3900	18.6842	0.1490	2.8237	25.9024	0.2187	8.8217	39.0312
9	0.1165	0.2790	8.7946	0.1302	1.4381	16.8736	0.1502	2.9837	25.0626	0.2281	7.4753	35.2964
10	0.1161	0.2661	8.3599	0.1309	1.5441	18.8746	0.1500	2.4584	23.2121	0.2241	8.1400	38.2714
MAX	0.1843	0.3874	8.7946	0.2037	1.9085	19.3483	0.2250	3.4041	25.9024	0.3027	8.9531	39.2312
Media	0.1224	0.2945	8.3358	0.1379	1.5350	18.1437	0.1567	2.9184	25.0428	0.2350	8.3564	37.2116
STD	0.0206	0.0475	0.3253	0.0219	0.2199	0.9282	0.0228	0.2790	0.6855	0.0245	0.4623	1.0690
MIN	0.1144	0.2232	7.7743	0.1301	1.2615	16.2877	0.1483	2.4584	23.2121	0.2187	7.4753	35.2964

Tabla 4.VI. Tiempos de procesado por etapas (en segundos)

Para finalizar el apartado de resultados, a continuación se aportan una serie de gráficas a modo de resumen comparativo. Dichas gráficas recopilan la media de cada una de las tablas obtenidas a lo largo de este apartado:

- En las figuras 4.1, 4.2, 4.3 y 4.4 se recogen las comparativas medias de la precisión, la sensibilidad, la precisión global y el coeficiente Kappa de Cohen respectivamente. Estas medias están calculadas utilizando los cuarenta modelos de clasificación y, a la hora de realizar las gráficas comparativas, se han agrupando por porcentajes y se han comparando los resultados de ambas versiones.

- En la figura 4.5 se realiza la comparativa de los tiempos de ejecución totales medios para cada uno de los porcentajes de entrenamiento.

- Por último, en la figura 4.6, se recoge la media del tiempo de ejecución de cada una de las fases que están implicadas en la construcción y posterior utilización de cada uno de los modelos de clasificación. Se utiliza una escala logarítmica para facilitar la visualización.

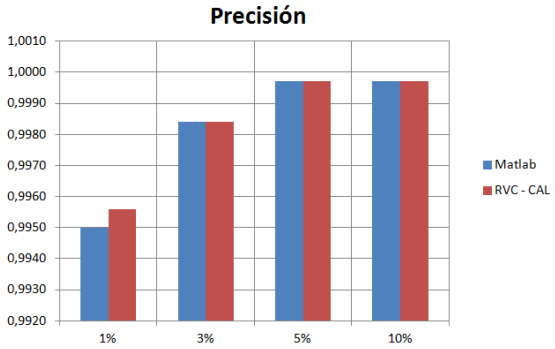

Figura 4.1. Comparativa de precisión

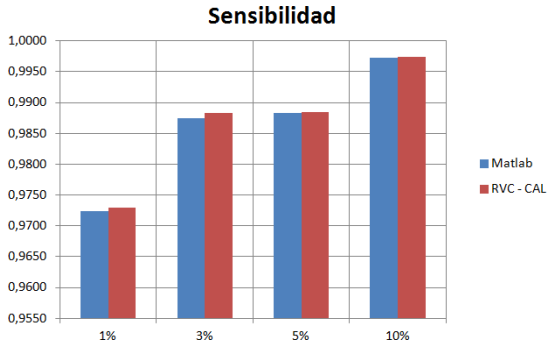

Figura 4.2. Comparativa de sensibilidad

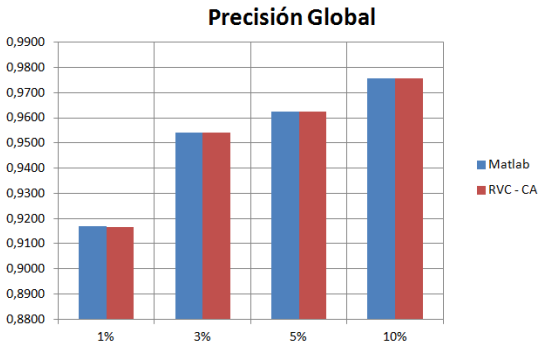

Figura 4.3. Comparativa de PG

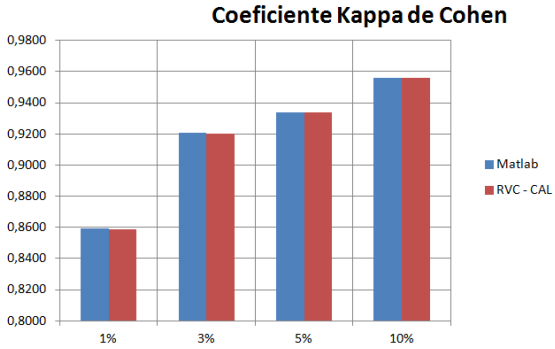

Figura 4.4. Comparativa de Kappa

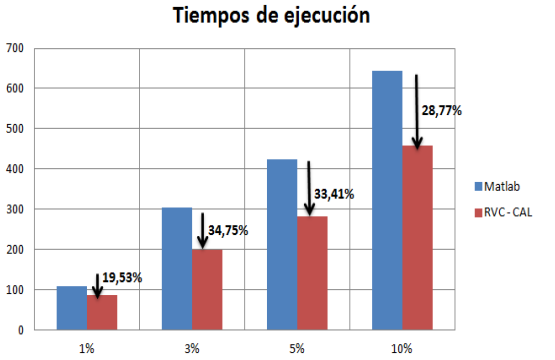

Figura 4.5. Comparativa de tiempos de ejecución

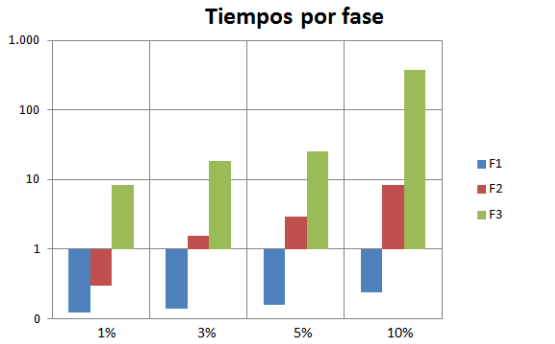

Figura 4.6. Comparativa de tiempos por fase

5 CONCLUSIONES

5.1. Conclusiones

Tras realizar el análisis de los resultados obtenidos al utilizar RVC – CAL para implementar el algoritmo del clasificador SVM optimizado para imágenes hiperespectrales, se han observado ciertos aspectos que se han considerado especialmente relevantes:

- Tras realizar una comparativa de los resultados arrojados por Matlab y de los obtenidos en la versión RVC – CAL, se ha observado que el comportamiento en cuanto a precisión de resultados es prácticamente idéntico.

- En cuanto a los resultados obtenidos, se extrae que conforme aumenta el número de píxeles utilizados para entrenar el modelo de clasificación, aumenta la precisión con la que se clasifica la imagen.

- Tras realizar una comparativa de los tiempos de ejecución entre ambas versiones, se corrobora que, en media, se ha conseguido un *speedup* de casi un 30%. Y, a su vez, que conforme aumenta el número de píxeles utilizados durante el entrenamiento, incrementa el tiempo de procesado requerido para analizar la imagen completa.

- Al analizar paso a paso las distintas etapas del algoritmo, concluimos que la fase que más tiempo implica y, por tanto, que supone un cuello de botella al realizar el análisis de imágenes hiperespectrales, es la propia clasificación de la imagen, ya que, en esta implementación, se clasifica píxel a píxel la imagen completa como un único bloque.

Analizando en conjunto los aspectos anteriormente mencionados, se extraen las siguientes conclusiones finales:

- Se ha logrado realizar satisfactoriamente la conversión entre lenguajes.

- El tiempo de ejecución de la versión secuencial se ha reducido en un 30% – excepto para el caso del 1% que se ha reducido en un 20% – y, además, el principal cuello de botella es fácilmente paralelizable.

- Al aumentar el tamaño del set de entrenamiento obtenemos resultados más precisos pero, a su vez, la clasificación requiere más tiempo.

Por último, queda demostrada la potencialidad de RVC – CAL para implementar el clasificador SVM y realizar una paralelización que nos permita reducir el tiempo de procesado. Como conclusión final, extraemos que, optimizando dicha paralelización, se podría alcanzar el objetivo fundamental del proyecto HELICoiD: el análisis de una imagen hiperespectral en tiempo real – entendiendo por tiempo real realizar el procesamiento antes de que la siguiente imagen sea capturada por la cámara hiperespectral.

5.2. Líneas futuras

Tras finalizar el presente Trabajo de Investigación y analizar los resultados obtenidos y las conclusiones alcanzadas, han surgido una serie de líneas de trabajo futuras relacionadas con esta línea de investigación. Las más importantes se citan a continuación:

- Desarrollar nuevos clasificadores SVM con diferentes *kernels* con el fin de completar la librería desarrollada y estudiar el comportamiento de dichos *kernels* a la hora de procesar imágenes hiperespectrales.

- Realizar una implementación paralelizada en lenguaje RVC – CAL para aumentar la velocidad de procesamiento. En concreto, implementar la clasificación por bloques de la imagen con el fin de reducir el tiempo de ejecución del principal cuello de botella.

- Implementar el clasificador SVM en una plataforma multinúcleo para, de esta manera, aportar una idea más clara del comportamiento del modelo de clasificación al proyecto HELICoiD.

- Incluir, como se dijo en el capítulo 2 de este mismo documento, una etapa previa de reducción dimensional para incrementar la velocidad de procesado de la imagen hiperespectral.

- Por último, una vez desarrollado el clasificador SVM con una paralelización eficiente y tras llevar a cabo la implementación en una plataforma multinúcleo, se puede estudiar el consumo de energía de dicho clasificador con el fin de desarrollar analizadores hiperespectrales portátiles.

6 REFERENCIAS

[1] D. Madroñal, "Generación de una librería RVC – CAL para la etapa de determinación de endmembers en el proceso de análisis de imágenes hiperespectrales", Proyecto Fin de Grado, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, Universidad Politécnica de Madrid, 2014.

[2] R. Lazcano, "Generación de una librería RVC-CAL para la etapa de estimación de abundancias en el proceso de análisis de imágenes hiperespectrales ", Proyecto Fin de Grado, Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, Universidad Politécnica de Madrid, 2014.

[3] S. Theodoridis, *"*Introduction to Pattern Recognition: A Matlab Approach: A Matlab Approach*" Academic Press*, 2010.

[4] P. Simon, "Too Big to Ignore: The Business Case for Big Data". *John Wiley & Sons*, 2013.

[5] P. Langley, "The changing science of machine learning." *Machine Learning* 82.3 (2011): 275-279.

[6] J.A. Gualtieri and S. Chettri. "Support vector machines for classification of hyperspectral data." *Geoscience and Remote Sensing Symposium*, 2000. Proceedings. IGARSS 2000. IEEE 2000 International. Vol. 2. IEEE, 2000.

[7] M. Rojas, "Caracterización de imágenes hiperespectrales utilizando Support Vector Machine y técnicas de extracción de características", Proyecto Fin de Carrera, Escuela Politécnica de Cáceres, Universidad de Extremadura, 2009.

[8] N. Kumar, P. Saurabh, and B. Verma. "An efficient approach parallel support vector machine for classification of diabetes dataset." *J.Computer Applications* 36.6 (2011): 19-24.

[9] V. Vapnik, "Principles of risk minimization for learning theory." *Advances in neural information processing systems*. 1992

[10] S. Stefan. *"*SVM kernels for time series analysis". No. 2001, 43*. Technical Report, SFB 475: Komplexitätsreduktion in Multivariaten Datenstrukturen*, Universität Dortmund, 2001.

[11] J.A. Gualtieri and R. Cromp. "Support vector machines for hyperspectral remote sensing classification." *The 27th AIPR Workshop: Advances in Computer-Assisted Recognition. International Society for Optics and Photonics*, 1999.

[12] F. Melgani and L. Bruzzone. "Classification of hyperspectral remote sensing images with support vector machines." *Geoscience and Remote Sensing*, IEEE Transactions on 42.8 (2004): 1778-1790.

[13] Y. Bazi and F. Melgani. "Toward an optimal SVM classification system for hyperspectral remote sensing images." *Geoscience and Remote Sensing*, IEEE Transactions on 44.11 (2006): 3374-3385.

[14] G. F. Hughes. “On the mean accuracy of statistical pattern recognizers”. *IEEE Transactions on Information Theory*, 14(1), 1968.

[15] G. Lu, B. Fei, "Medical hyperspectral imaging: a review*", Journal of biomedical optics*, vol. 19, no. 1, 2014.

[16] K. Rajpoot and N. Rajpoot. "SVM optimization for hyperspectral colon tissue cell classification." *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2004*. Springer Berlin Heidelberg, 2004. 829-837.

[17] S. Bhattacharyya, E. Johan, J. W. Janneck, C. Lucarz, M. Mattavelli, M. Raulet, “Overview of the MPEG Reconfigurable Video Coding Framework”, *Journal Of Signal Processing Systems 63*, pp 251-263, © 2009 Springer Science + Business Media, LLC. Manufactured in The United States. DOI: 10.1007/s11265-009-0399-3.

[18] J. Eker and J. W. Janneck, "Cal language report", *Tech. Rep. UCB/ERL M03/48*, University of California at Berkeley, December 2003.

[19] C. Lucarz, I. Amer, M. Mattavelli, “Reconfigurable video coding: Objectives and technologies”, *16th IEEE International Conference on Image Processing (ICIP)*, pp. 749-752, Nov. 2009.

[20] M. Wipliez, G. Roquier, J.-F. Nezan, “Software code generation for the RVC-CAL language”, *Journal of Signal Processing Systems 63,* vol 2, pp 203-213, © 2009 Springer Science + Business Media, LLC. Manufactured in The United States. DOI: 10.1007/s11265-009-0390-z.

[21] M. Wipliez, G.Roquier, M. Raulet, J.-F. Nezan, O. Deforges, “Code generation for the MPEG Reconfigurable Video Coding framework: From CAL actions to C functions”, *IEEE International Conference on Multimedia and Expo 2008*, IEEE, Hannover, Germany, DOI: 10.1109/ICME.2008.4607618.

[22] E. Bezati, M. Mattavelli, M. Raulet, "RVC-CAL dataflow implementations of MPEG AVC/H. 264 CABAC decoding", *Design and Architectures for Signal and Image Processing (DASIP)*, 2010 Conference on. IEEE, 2010.