UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE TEORÍA DE LA SEÑAL Y COMUNICACIONES

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN: SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN EFICIENTE DE CLASIFICADORES PRIOR-SVM PARA MATLAB

AUTOR: FERNANDO VALLE PADILLA
TUTOR: EMILIO PARRADO HERNÁNDEZ

ABRIL, 2010

El destino es el que baraja las cartas, pero nosotros los que las jugamos.

Arthur Schopenhauer

Más vale arrepentirse de lo que has hecho que arrepentirse de lo que no has hecho.

Agradecimientos

Muchas son las personas que han contribuido a que por fin sea ingeniero y a que, a pesar de los malos momentos, haya vivido los que probablemente hayan sido los mejores años de mi vida.

Mi etapa universitaria quedará para siempre ligada a la Residencia y a la increíble experiencia que mi estancia allí supuso. En ella he llorado, reído y vivido momentos magníficos que no cambiaría por nada. Hoy puedo presumir de haber conocido allí a muchísima gente interesante y de haber hecho grandes amigos para toda la vida.

Además de mi gente de la residencia, un grupo de amigos que han sido una pieza fundamental en mi vida universitaria han sido mis Booleanos, miembros (casi) todos de aquel equipo cuyas victorias se cuentan más por los buenos momentos vividos que por los partidos ganados... Junto a ellos, las largas tardes de biblioteca se hacían más amenas y los malos momentos más llevaderos. Tampoco puedo olvidar a mis chavales de Residual, con quienes viví grandes tardes de gloria tanto dentro como fuera de los terrenos de juego. Gracias también a mis compañeros de *Informadores* por las maravillosas experiencias vividas en el mejor trabajo que nunca tendré.

Gracias también a mi gente de Córdoba, que siempre me han dado ánimos y sé que están orgullosos de mí, al igual que yo lo estoy de ellos.

Gracias a mi tutor de proyecto, Emilio, por su dedicación y extrema paciencia.

Y cómo no, a mi familia, que siempre me ha apoyado y la he tenido cerca a pesar de la distancia.

Por último, a ti, con quien más me río y a quien más quiero, por apoyarme en todo momento y quererme como soy.

Resumen

El presente proyecto pretende dar un paso más en el desarrollo del Aprendizaje Máquina mediante la programación eficiente de una variación de la Máquina de Vectores Soporte, perteneciente a la familia de los clasificadores semi lineales. Esta variación consiste en introducir conocimiento *a priori* en el entrenamiento y recibe el nombre de Prior-SVM. El entrenamiento es llevado a cabo mediante el método de *Sequential Minimal Optimization*.

El punto de partida del proyecto es un código en MATLAB, el cual ha sido reprogramado, mejorado y completado con el fin de ganar en velocidad y poder así trabajar con un mayor número de muestras. Para ello, el programa se ha realizado en un lenguaje de medio nivel, C, y se ha integrado en una completa *toolbox* en MATLAB mediante el uso de la API MEX.

Índice General

Índice de Figuras

Índice de Tablas

Acrónimos

ACRÓNIMO	DESCRIPCIÓN
SVM	*Support Vector Machine*
SMO	*Sequential Minimal Optimization*
ML	*Machine Learning*
MATLAB	*MATrix LABoratory*
LIBSVM	*Library for Support Vector Machines*
OSH	*Optimal Separating Hyperplane*
KKT	*Karush-Kuhn-Tucker*
QP	*Quadratic Programming*
RAM	*Random Acces Memory*

Capítulo 1

INTRODUCCIÓN

En este primer capítulo de introducción comenzaremos presentando el marco bajo el que se desarrolla el presente trabajo, el Aprendizaje Máquina, así como las aplicaciones que su uso puede tener. Tras esto, se definirán los objetivos generales del proyecto y se explicará la organización del documento.

1.1 El Aprendizaje Máquina

El Aprendizaje de Máquina o Automático (*Machine Learning* en inglés) es una rama de la Inteligencia Artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras *aprender* y cambiar su comportamiento de manera autónoma basándose en su experiencia [1]. Se trata de crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos. Es, por lo tanto, un proceso de inducción del conocimiento.

En muchas ocasiones el campo de actuación del Aprendizaje Automático se solapa con el de la Estadística, ya que las dos disciplinas se basan en el análisis de datos. Sin embargo, el Aprendizaje Automático se centra más en el estudio de la

Complejidad Computacional de los problemas. El Aprendizaje Máquina puede ser visto como un intento de automatizar algunas partes del Método Científico mediante métodos matemáticos.

El Aprendizaje Automático ha jugado un rol fundamental en áreas tales como la bioinformática, la recuperación de información en la web, la inteligencia de negocios y el desarrollo de vehículos autónomos.

Las **Máquinas de Vectores Soporte** (SVM de sus siglas en inglés) son una familia de algoritmos de aprendizaje automatizado que, a partir de datos previamente discriminados, aprenden automáticamente y son capaces de realizar clasificaciones binarias. Las grandes ventajas que tiene la SVM son:

- Gran potencial en tareas de clasificación, principalmente por su excelente capacidad de generalización, debido a que están fundamentadas en la teoría de aprendizaje estadístico[1] [2].
- Existen pocos parámetros a ajustar; el modelo solo depende de los datos con mayor información.
- La solución de SVM es *sparse*, esto es que la mayoría de las variables son cero en la solución de SVM, lo que quiere decir que el modelo final puede ser escrito como una combinación de un número muy pequeño de vectores de entrada, los llamados **Vectores Soporte**.

La SVM se verá más en detalle en el apartado 2.5.

[1] Los métodos estadísticos de aprendizaje van desde el simple cálculo de medias hasta la construcción de modelos complejos como las redes bayesianas o las redes neuronales. Tienen aplicación en Informática, ingeniería, neurobiología, psicología, física... El aprendizaje estadístico es un área de investigación muy activa. Se han hecho enormes avances tanto en la teoría como en la práctica, hasta el punto que es posible aprender casi cualquier modelo para el cual sea posible inferencia aproximada o exacta.

1.2 Aplicaciones

El Aprendizaje Automático tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica.

El aprendizaje mediante Vectores Soporte se ha generalizado enormemente, teniendo en la actualidad numerosas aplicaciones prácticas [3] tales como:

- Reconocimiento faciales y de objetos en tres dimensiones
- Clasificación en procesos industriales
- Detección remota como matrículas
- Reconocimiento de caracteres escritos a mano
- Aplicaciones biomédicas como reconocimiento de enfermedades o clasificación de genes
- Estimación de precios dados unas características
- Mejora de los motores de búsqueda en Internet

1.3 Objetivos Generales

El presente proyecto se enmarca dentro del Aprendizaje Máquina y surge con la necesidad de hacer más eficiente un software para el entrenamiento de una variación de la SVM. Dicha variación consiste principalmente en introducir conocimiento *a priori* en el entrenamiento y recibe el nombre de **Prior-SVM**. El entrenamiento se lleva a cabo mediante el método de *Sequential Minimal Optimization* (**SMO**), el cual habrá que adaptar al nuevo problema.

El punto de partida es un software programado en MATLAB que entrena la Prior-SVM mediante SMO. Al tratarse de un lenguaje de alto nivel, el tiempo de

procesamiento es muy elevado, lo que limita el número de muestras con las que poder trabajar.

El objetivo del proyecto es mejorar ese código y crear en MATLAB una completa *toolbox*[2] que englobe tanto el entrenamiento de la máquina como la clasificación binaria de muestras.

La mejora en el código se traduce en una mayor velocidad de ejecución, lo que conlleva un aumento del número de muestras que se pueden entrenar/clasificar. Para lograr esto se ha optado por reprogramar el código en un lenguaje de más bajo nivel. El lenguaje elegido ha sido **C**, que aunque es más complejo, presenta mejores prestaciones que MATLAB. Además, se ha creado una caché para hacer más eficientes las operaciones con las matrices, especialmente cuando se trabaja con muchos datos.

Este código en C se ha introducido en la *toolbox* mediante el empleo de funciones MEX, que permiten que una función en C sea llamada desde un entorno MATLAB. Además, se han creado una serie de scripts y funciones en MATLAB que complementan la *toolbox* y que se describirán en los capítulos posteriores.

Para probar la eficiencia o no de esta nueva implementación, se procederá a comparar el tiempo que ambos programas tardan en entrenar la Prior-SVM y en la posterior clasificación de unas muestras.

Además de esta comparación, será interesante comparar los resultados de clasificación de una SVM, sin conocimiento *a priori*, frente a Prior-SVM, que sí tiene conocimiento *a priori*. Para ello, se utilizará la librería LIBSVM, que contiene una SVM entrenada con SMO.

[2] Conjunto de rutinas ya existentes para usarlas en la escritura de nuevos programas.

1.4 Estructura del Documento

El documento está dividido en cinco capítulos. En este primero de introducción, se recogen tanto el planteamiento del problema como los objetivos del proyecto. Además, se define el Aprendizaje Máquina, marco en el que se engloba este trabajo, y se explica cómo se va a organizar la presente memoria.

En el capítulo segundo, veremos el Marco Teórico en el que se desarrolla el problema y cuyo conocimiento resultará fundamental para la posterior comprensión de la solución propuesta.

En el tercer capítulo se explicará cómo se ha planteado la resolución del problema así como los medios técnicos utilizados. Veremos la arquitectura empleada y analizaremos el código del programa.

El capítulo cuarto desglosa los resultados de los experimentos realizados, comparando el programa presentado con su equivalente en MATLAB. Además se compararán las soluciones obtenidas por la Prior-SVM frente a la SVM.

En el quinto capítulo se expondrán las conclusiones extraídas en la realización del presente proyecto.

A modo de notación, comentar que para distinguir variables de vectores en la formulación, estos últimos se han escrito en **negrita**.

Capítulo 2

MARCO TEÓRICO

En este capítulo se introducen muchos de los conceptos teóricos sobre los que se apoya el presente proyecto. Comenzaremos estudiando los elementos que componen un sistema de clasificación y los tipos de aprendizaje existentes, centrándonos en el aprendizaje supervisado, que es el desarrollado en el proyecto. Posteriormente veremos en qué consiste la clasificación multiclase y compararemos la SVM con otra máquina de aprendizaje, las Redes Neuronales. Tras esto, pasaremos a explicar en qué consiste la Máquina de Vectores Soporte, estudiando tanto el caso separable como el no separable. Para este último caso veremos como la SVM no lineal es una buena alternativa para abordar el problema. Para acabar, se explicará en qué consiste el entrenamiento SMO y se detallarán las modificaciones introducidas por Prior-SVM.

2.1 Introducción

Muchos de los problemas de tratamiento estadístico de información se pueden agrupar en dos conjuntos: los problemas de clasificación o decisión y los problemas de regresión.

En los de clasificación, el interés se centra en saber si una determinada muestra, en función de las características que presenta, pertenece a una u otra clase. En regresión, por el contrario, lo que se pretende es estimar una variable desconocida a partir de observaciones que guardan algún tipo de relación con ella. Tanto la decisión como la regresión pueden verse como casos particulares de un problema de aproximación de funciones.

La clasificación, en el ámbito del Aprendizaje Máquina, consiste en encontrar una regla de decisión tal que dada una muestra externa, ésta sea asignada a su clase correspondiente. El caso más sencillo, y el que se va a desarrollar en el presente proyecto, es el binario, en el que únicamente tenemos dos clases, las cuales llamaremos +1 y -1.

La búsqueda de una regla de decisión adecuada puede interpretarse como la estimación de una función *g*, la cual asigna a cada punto del espacio de observación, un punto en el espacio de las clases (-1 y +1 en el caso binario). Esta búsqueda se lleva a cabo usando un conjunto de *datos de entrenamiento* formado por *N* pares de muestras etiquetadas, distribuidas generalmente mediante una distribución de probabilidad desconocida $P(\boldsymbol{x}, y)$.

Lo que se busca con la función *g* es que generalice, es decir, que clasifique correctamente un conjunto de datos que no pertenezcan al conjunto de entrenamiento. Esta función suele tomar la forma:

$$g(\boldsymbol{x}) = sign\big(f(\boldsymbol{x})\big) \tag{2.1}$$

Así pues, suponiendo el caso binario, las muestras nuevas se asignarán a la clase +1 si $f(\boldsymbol{x}) > 0$ o a la clase -1 si $f(\boldsymbol{x}) < 0$. Al valor de $f(\boldsymbol{x})$ se le denomina *salida blanda*, mientras que a $sign\big(f(\boldsymbol{x})\big)$, *salida dura*.

Para un resultado óptimo, tanto las muestras de entrenamiento como las nuevas deben tener una misma distribución de probabilidad $P(\boldsymbol{x}, y)$. La mejor función $f(\boldsymbol{x})$ será aquella que tenga la esperanza del error de clasificación menor, la que conlleve un menor coste:

$$C(f) = \int l(f(\boldsymbol{x}), y)\partial P(\boldsymbol{x}, y) \quad (2.2)$$

donde *l* es la función de coste que expresa la penalización que se produce al tomar, a partir de la observación de ***x***, una decisión basada en el valor de ***f(x)*** siendo el valor correcto *y*.

Tanto los datos de entrenamiento como los de clasificación son vectores en los que cada dimensión constituye una *característica* o atributo de la muestra. Así por ejemplo, si el conjunto a clasificar es *coches*, cada ejemplo (***x***) será un vector que contenga las características del coche (longitud, anchura, velocidad máxima…).

2.2 Componentes de un sistema de clasificación

En la resolución de problemas de toma de decisiones, la primera parte de la tarea consiste en clasificar el problema o la situación, para después aplicar la metodología correspondiente, que en buena medida dependerá de esa clasificación.

Cuando se diseña un sistema de clasificación es necesario definir cada uno de sus componentes. Los principales componentes de un sistema de clasificación general son [4]:

- **Clase**: Conjunto de entidades que comparten alguna característica que las diferencia de otras
- **Clase de rechazo**: Conjunto de entidades que no se pueden etiquetar como ninguna de las clases del problema.
- **Extractor de características**: Subsistema que extrae información relevante para la clasificación a partir de las entidades cuantificables.
- **Clasificador**: Subsistema que utiliza un vector de características de la entidad cuantificable y lo asigna a una de las *M* clases.
- **Evaluación de error de clasificación**: Error que se comete al realizar la clasificación.

- **Falso rechazo y Falsa aceptación**: También conocidos como falso negativo y falso positivo respectivamente, es el error producido cuando el sistema diagnostica una clase siendo la otra la correcta. Para problemas con dos clases, estas definiciones reflejan la importancia de una decisión contra la opuesta.

En el presente proyecto no se ha implementado una clase de rechazo ya que todas las muestras se etiquetan como algunas de las clases del problema.

El diseño de un clasificador comprende dos pasos:

1. Definir el conjunto de etiquetas
2. Definir qué salidas debe proporcionar el sistema

Si suponemos que todos los patrones a reconocer son elementos potenciales de *M* clases distintas denotadas w_j = 1,2 … , M, llamaremos

$$\Omega = \{w_1, w_2, \dots, w_M\} \tag{2.3}$$

al conjunto de clases informacionales, definiendo estas como clases conocidas y con significado. Así pues, para un problema de teledetección, un conjunto de clases informacionales podría ser:

$$\Omega = \{cultivo, agua, carretera\} \tag{2.4}$$

Aunque no es nuestro caso, resulta conveniente ampliar el conjunto Ω incorporando la clase de rechazo (w_0), para todos aquellos patrones para los que no se tiene una certeza aceptable de ser clasificados correctamente en alguna de las clases de Ω. Así pues, el conjunto extendido de clases informacionales es:

$$\Omega^* = \{w_1, w_2, \dots, w_M, w_0\} \tag{2.5}$$

Una vez establecido el conjunto de clases se procede a la construcción del clasificador. Esto conlleva las siguientes etapas:

1. Elección del modelo
2. Aprendizaje o entrenamiento del clasificador
3. Verificación de los resultados

Formalmente, un clasificador o regla de clasificación es una función $d: \boldsymbol{\rho} \rightarrow \Omega^*$ definida sobre los patrones $\boldsymbol{x}$ tal que para todo patrón $\boldsymbol{x}$, $d(\boldsymbol{x}) \in \Omega^*$. $\boldsymbol{\rho}$ es el espacio de patrones, el conjunto de todos los valores posibles que puede tomar el patrón $\boldsymbol{x}$.

En lo que respecta al aprendizaje, distinguimos dos tipos [5]:

- **Aprendizaje supervisado**. Este tipo de entrenamiento requiere disponer de un conjunto de patrones de los cuales se conoce su clase. La salida no coincidirá generalmente con la deseada, de la que se tiene conocimiento, de forma que se generará un error de salida. Este error se empleará para adaptar los parámetros del sistema.

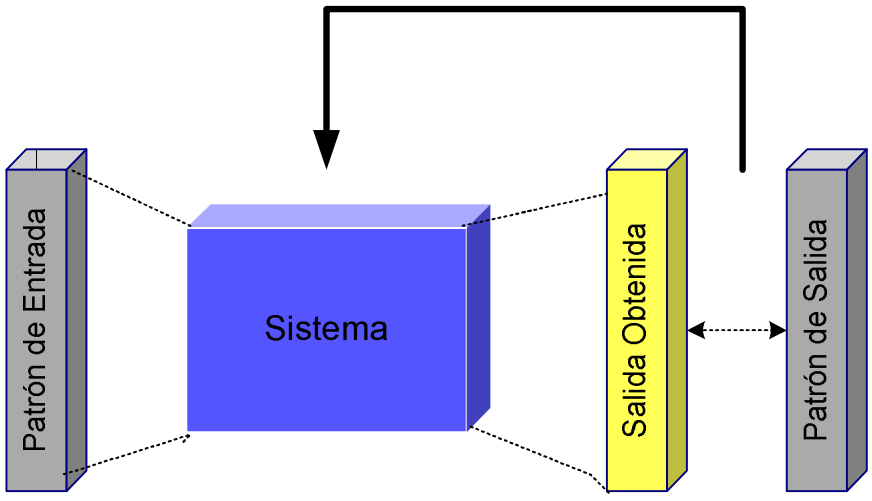

FIGURA 2.1 APRENDIZAJE SUPERVISADO

- **Aprendizaje no supervisado**. Este tipo de entrenamiento se realiza a partir de un conjunto de patrones del que no se conoce su clase. A veces ni el número de clases siquiera se conoce. La función de este aprendizaje suele ser la de verificar la validez del conjunto de clases informacionales para una clasificación supervisada, así como descubrir las regularidades presentes en los datos, extraer rasgos o agrupar patrones según su similitud. Las técnicas empleadas suelen denominarse métodos de agrupamiento o *clustering***.**

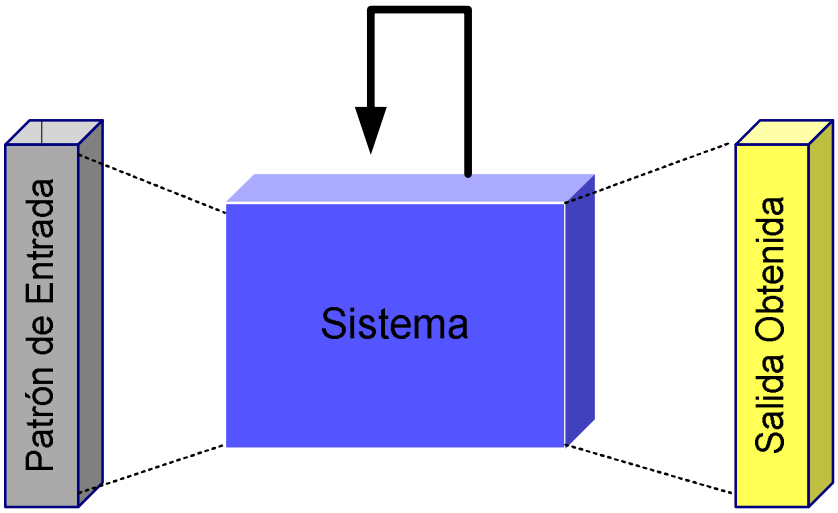

FIGURA 2.2 APRENDIZAJE NO SUPERVISADO

En el presente proyecto se ha utilizado aprendizaje supervisado. Para llevar a cabo este tipo de aprendizajes, generalmente se divide el conjunto de datos en dos grupos. El primero, y normalmente el más numeroso, es el llamado **conjunto de entrenamiento** o de *train*. Este conjunto de datos se usa para que la máquina aprenda las relaciones entrada/salida del problema a partir de los ejemplos que lo componen. Se persigue por tanto la *generalización*. El segundo grupo de datos recibe el nombre de **conjunto de *test*** y contiene los datos que no se han usado en el entrenamiento. La función de este conjunto es evaluar las prestaciones de la máquina para unas entradas no vistas en la etapa de entrenamiento. En la siguiente sección se va a estudiar un poco más en detalle este tipo de aprendizaje.

2.3 Aprendizaje supervisado

Si consideramos que en un caso ideal cada agrupamiento representa a una clase y que cada clase tiene asociado un agrupamiento bien diferenciado de los demás, un problema de clasificación supervisada puede plantearse como la búsqueda de las superficies que separan los diferentes agrupamientos. Son las denominadas **superficies de decisión**. Es importante decir que los grupos no tienen por qué ser conexos.

Las superficies de decisión determinan regiones de decisión de forma que cada clase tiene asociada una región en $\boldsymbol{\rho}$ y la decisión sobre la clase a asignar a un nuevo patrón se hará en base a la región en la que éste se encuentre en $\boldsymbol{\rho}$.

En el aprendizaje supervisado, se dice que un punto $\boldsymbol{x}$ está en la región de decisión asociada a una clase si al evaluar la función de pertenencia de esa clase, ésta lo indica. Construir un clasificador en base a regiones de decisión puede considerarse como una aproximación inversa a la construcción de la función *d*, ya que se trata de definir $\boldsymbol{R_i}$ para $i = 0,1,2,\dots,L$ como el subconjunto de $\boldsymbol{\rho}$ para el cual $d(\boldsymbol{x}) = w_i$, o sea,

$$\boldsymbol{R}=\{\boldsymbol{x}; d(\boldsymbol{x}) = w_i\} \tag{2.6}$$

donde $\boldsymbol{R_0}, \boldsymbol{R_1}, \boldsymbol{R_2}, \dots, \boldsymbol{R_j}$ son distintos y verifican que

$$\boldsymbol{\rho} = \cup_i \cdot \boldsymbol{R_i} \tag{2.7}$$

Los subconjuntos $\boldsymbol{R_i}$ forman una partición de $\boldsymbol{\rho}$. Con este planteamiento, un clasificador es una partición de $\boldsymbol{\rho}$ en L+1 subconjuntos disjuntos

$$\boldsymbol{R_0}, \boldsymbol{R_1}, \boldsymbol{R_2} \dots, \boldsymbol{R_L}(\boldsymbol{\rho} = \cup_i \cdot \boldsymbol{R_i}) \tag{2.8}$$

Cada una de las regiones de $\boldsymbol{R_i}$ se conocen como **regiones de decisión** y las fronteras entre ellas como **fronteras de decisión**.

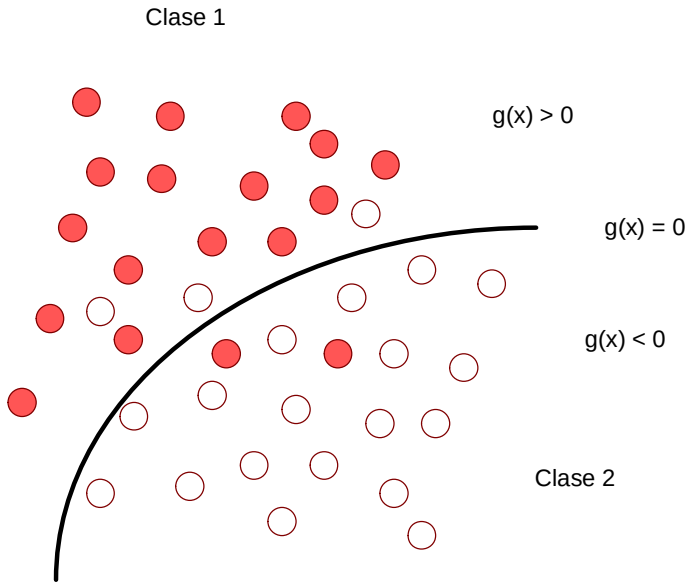

FIGURA 2.3 EJEMPLO DE CLASIFICACIÓN BINARIA

En la figura 2.3 se muestra un ejemplo de clasificación con dos clases. Dado un patrón $\boldsymbol{x}$, la decisión sobre la clase a la que pertenece dependerá de la función discriminante $g(\boldsymbol{x})$. La frontera de decisión entre las dos clases es el lugar de los puntos que verifica la ecuación $g(\boldsymbol{x}) = 0$. De este modo, si $g(\boldsymbol{x}) > 0$, la muestra se etiquetará como de clase 1, mientras que si $g(\boldsymbol{x}) < 0$, lo hará como de clase 2. Así pues, $\boldsymbol{R_i}$ es la región de decisión asociada a la clase i y comprende todos los puntos a los que se les ha asociado dicha etiqueta:

$$\boldsymbol{R_1} = \{\boldsymbol{x}; g(\boldsymbol{x}) > 0\} \tag{2.9}$$

$$\boldsymbol{R_2} = \{\boldsymbol{x}; g(\boldsymbol{x}) < 0\}$$

2.4 Clasificación multiclase

Hasta ahora hemos estudiado el caso de clasificación binaria, es decir, clasificar muestras en dos tipos posibles de clases. Cuando se tratan problemas de clasificación extraídos de la vida real, nos solemos encontrar con problemas de **multiclasificación** [6] como el de la figura 2.4 mostrada a continuación.

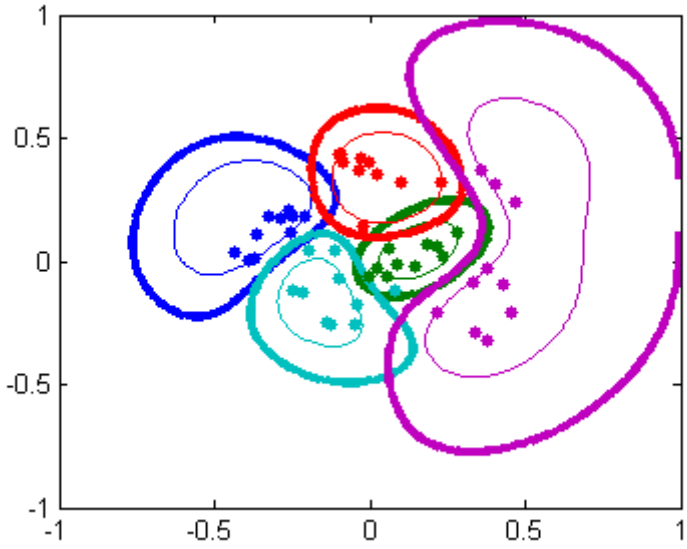

FIGURA 2.4 EJEMPLO DE UN CLASIFICADOR CON CINCO CLASES

Para abordar estos problemas de clasificación en *M* clases existen dos aproximaciones básicas empleando clasificadores binarios:

- Método ***Uno contra todos***. Consiste en combinar clasificadores binarios entrenados independientemente para resolver distintas partes del problema. Este método construye *M* modelos de SVM, donde *M* es el número de clases. Cada SVM *m-ésimo* entrenará con todos los ejemplos, etiquetando los de la clase *m-ésima* con etiqueta positiva y al resto de elementos de las demás clase, con etiqueta negativa.

- Método ***Todos contra todos***. En este método se construyen *M*(M-1)/2* clasificadores, uno para cada par de clases posibles, enfrentando así a todas las clases una a una. En cada entrenamiento se emplea únicamente los datos

de las dos clases involucradas. Así pues, un problema con 4 clases generaría los siguientes clasificadores: 1 vs 2, 1 vs 3, 1 vs 4, 2 vs 3, 2 vs 4 y 3 vs 4. Una vez realizado esto, se someten los datos de *test* a todos estos clasificadores, donde se añade un voto a la clase ganadora para cada caso. Finalmente, aquella que más votos obtenga será la clase propuesta por el sistema.

Es importante notar que como el método todos-contra-todos trabaja con menos muestras, tiene mayor libertad para encontrar una frontera que separe ambas clases [7]. Respecto al coste de entrenamiento, es preferible el uso de uno-contra-todos puesto que sólo ha de entrenar *M* SVMs. La complejidad de prueba de ambas estrategias es similar: uno-contra-todos necesita *M* evaluaciones y todos-contra-todos *M* – 1.

2.5 Máquina de Vectores Soporte

La Máquina de Vectores Soporte o SVM (*Support Vector Machine*) fue desarrollada en 1963 por Vapnik y Lerneren en los laboratorios AT&T, pero no fue hasta los años noventa, con la publicación del primer *paper* por Boser en 1992 y por Cortes y Vapnik en 1995, cuando fue desarrollada y generalizada. Fue ideada originalmente para la resolución de problemas de clasificación binarios en los que las clases eran linealmente separables [8]. Es por esto por lo que también se le conocía como **Hiperplano[3] Óptimo de Separación** *o OSH (Optimal Separating Hyperplane)*, ya que la solución obtenida era aquella en la que se clasificaban de manera correcta todas las muestras, colocando el hiperplano de separación lo más lejos posible de todas ellas [9]. Para lograr esto, se realiza un entrenamiento con los datos dispuestos para ese fin, a partir del cual se define el hiperplano óptimo, es

[3] Un hiperplano es una generalización del concepto de plano. En un espacio de una única dimensión (como una recta), un hiperplano es un punto; divide una línea en dos líneas. En un espacio bidimensional (como el plano *xy*), un hiperplano es una recta; divide el plano en dos mitades. En un espacio tridimensional, un hiperplano es un plano corriente; divide el espacio en dos mitades. Este concepto también puede ser aplicado a espacios de cuatro dimensiones y más, donde estos objetos divisores se llaman simplemente hiperplanos, ya que la finalidad de esta nomenclatura es la de relacionar la geometría con el plano.

decir, el que maximiza el margen entre las muestras de cada clase respecto a la frontera de separación.

El objetivo es que tras este entrenamiento, la máquina generalice bien para datos nuevos que no han participado en el entrenamiento, es decir, clasifique correctamente a que clase pertenecen cada una de las muestras. Para ello, intenta buscar una función *f* que minimice el coste empírico sobre el conjunto de entrenamiento, manteniendo a su vez la correcta generalización de la máquina.

Para obtener una buena generalización, hay que tratar de evitar dos de los efectos típicos del entrenamiento:

- **Sobreajuste**: los parámetros obtenidos en el entrenamiento se ajustan demasiado a las muestras por lo que se pierde generalización y al introducir una nueva muestra no la clasifica bien. En la figura siguiente podemos ver un ejemplo de sobreajuste producido por línea continua.

- **Subajuste**: el hiperplano trazado es demasiado sencillo y no logra una buena generalización debido a que se han utilizado muy pocas muestras de entrenamiento, por lo que la máquina no generara buenos resultados con muestras nuevas. La línea discontinua de la siguiente figura ilustra un ejemplo de subajuste.

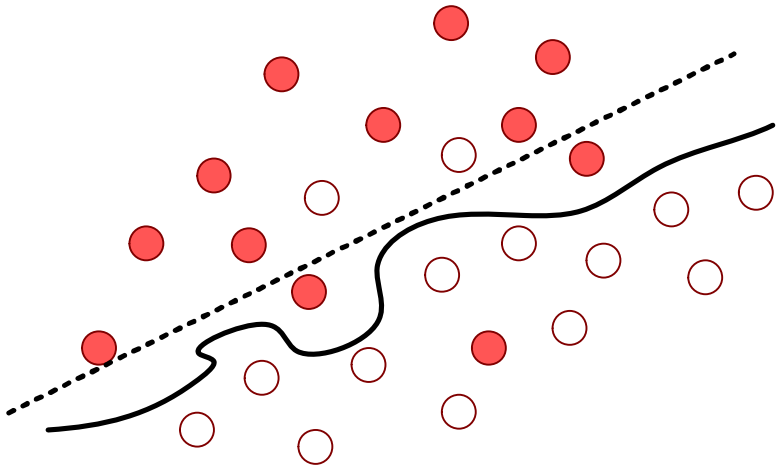

FIGURA 2.5 SOBREAJUSTE Y SUBAJUSTE

Si comparamos la SVM con otros paradigmas de aprendizaje y procesamiento automáticos como las Redes Neuronales, comprobamos que la SVM obtiene una mejor generalización, que hace que funcione bien cuando hay pocos datos y el espacio de entrada es de dimensión alta.

2.5.1 Caso separable linealmente

Se dice que un problema es linealmente separable cuando para cualquier conjunto de muestras existe un único hiperplano que clasifica con error cero. De entre todo el conjunto de muestras, se extraen una serie de vectores, los Vectores Soporte, que son los únicos que se necesitan de entre todos los datos para definir la frontera de decisión.

Usando multiplicadores de Lagrange[4] es posible representar el hiperplano deseado como combinación lineal de los propios datos. Se demuestra que la gran mayoría de los coeficientes de Lagrange serán nulos y que únicamente serán distintos de cero aquellos puntos situados exactamente a la distancia marcada por el margen, estos son los Vectores Soporte, que actuarán como resumen de todo el conjunto de datos en el sentido de que la solución únicamente depende de ellos [10]. De este modo, si se entrena una máquina sólo teniendo en cuenta los Vectores Soporte, se obtendrán los mismos resultados que entrenando con todo el conjunto de datos.

En la figura mostrada a continuación se puede observar un caso binario linealmente separable. Como se ha dicho, el hiperplano óptimo será aquel que maximice el margen, y por tanto en este ejemplo, será aquel que tiene *margen 1*. Como se observa, los Vectores Soporte son aquellos que se encuentran justo a la distancia del margen.

[4] Los **Multiplicadores de Lagrange** son un método para trabajar con funciones de varias variables que nos interesa maximizar o minimizar, y está sujeta a ciertas restricciones. Este método reduce el problema restringido en n variables en uno sin restricciones de $n + 1$ variables cuyas ecuaciones pueden ser resueltas. Este método introduce una nueva variable escalar desconocida, el multiplicador de Lagrange, para cada restricción y forma una combinación lineal involucrando los multiplicadores como coeficientes. Su demostración involucra derivadas parciales.

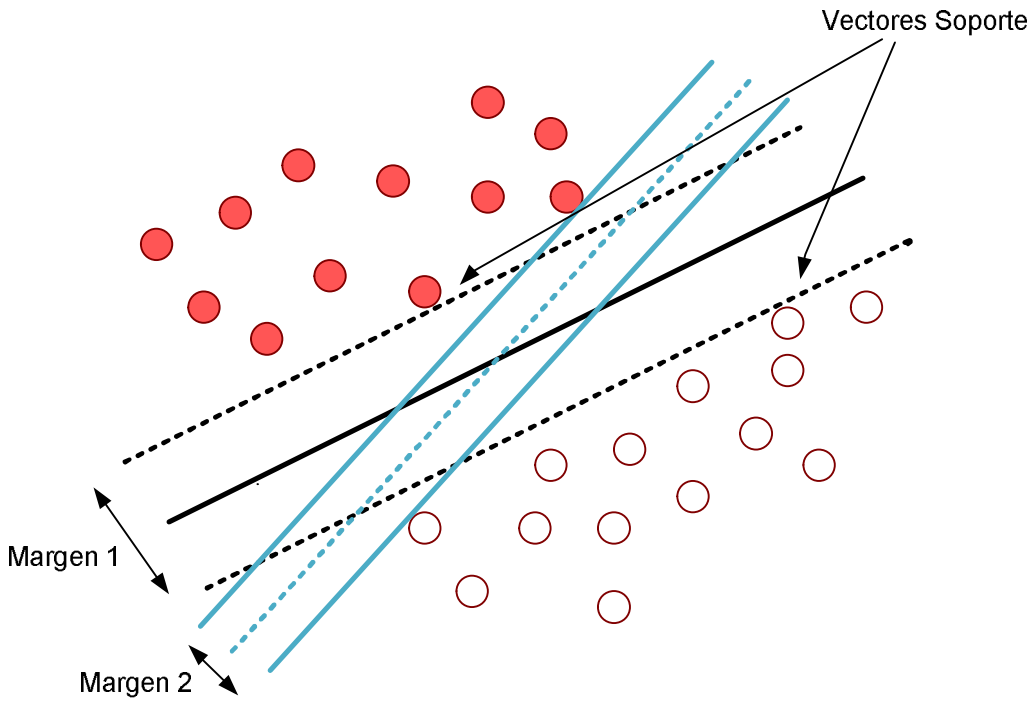

FIGURA 2.6 MAXIMIZACIÓN DEL MARGEN

Matemáticamente, dadas *N* muestras, $\boldsymbol{x_i}$, con sus correspondientes etiquetas o *labels* asociados, y_i , que definirán a qué clase pertenece cada muestra, tenemos:

$$(\boldsymbol{x_1} \cdot y_1), (\boldsymbol{x_2} \cdot y_2), \dots, (\boldsymbol{x_N} \cdot y_N); \quad \boldsymbol{x_i} \in R^J, \; y_i \in \{+1, -1\} \tag{2.10}$$

siendo J el número de dimensiones o componentes de los vectores que contienen los datos.

Un clasificador es lineal si su función de decisión puede expresarse mediante una función lineal en **x**. Así pues, la ecuación del hiperplano de separación será el lugar de los puntos **x** en los que se cumple:

$$H\colon \boldsymbol{x} \cdot \boldsymbol{w} + b = 0 \tag{2.11}$$

siendo *b* una constante que indica la posición del plano respecto al origen de coordenadas. Esta constante recibe el nombre de *sesgo*. $\boldsymbol{w}$ es el vector normal al hiperplano y tiene la forma:

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x_i} \tag{2.12}$$

donde los α_i son los **multiplicadores de Lagrange**, los cuales como se ha mencionado anteriormente, serán nulos en su mayoría, salvándose únicamente los que sean Vectores Soporte.

La clasificación se realiza determinando en qué zona del hiperplano está el punto a clasificar. Así pues, el clasificador puede representarse mediante las expresiones:

$$\begin{aligned} \boldsymbol{x_i} \cdot \boldsymbol{w} + b > +1 \quad para \quad y_i = +1 \\ \boldsymbol{x_i} \cdot \boldsymbol{w} + b < -1 \quad para \quad y_i = -1 \end{aligned} \tag{2.13}$$

Si el resultado de la operación es positivo la muestra pertenecerá a una clase, y si es negativo la otra. Las expresiones anteriores pueden combinarse en una única:

$$y_i\{\boldsymbol{x_i} \cdot \boldsymbol{w} + b\} \geq 1 \tag{2.14}$$

En un problema linealmente separable habrá infinitos hiperplanos que cumplan esta condición. El que buscamos es aquel que tenga un mayor margen. Es decir, queremos maximizar la distancia entre los datos y la frontera de decisión. Los puntos que caen sobre cada uno de los hiperplanos son los que cumplen:

$$\begin{aligned} H_1 &: \boldsymbol{x_i} \cdot \boldsymbol{w} + b = +1 \\ H_2 &: \boldsymbol{x_i} \cdot \boldsymbol{w} + b = -1 \end{aligned} \tag{2.15}$$

Por tanto estos dos hiperplanos son paralelos entre si y paralelos al hiperplano *H* (2.11). El margen será la distancia entre H_1 y H_2. Lo que se busca aquí es aumentar la generalización, es decir, que una vez entrenada la máquina, clasifique correctamente las muestras nuevas. Para ello, cuanto mayor sea la distancia entre los datos y la frontera de clasificación, mejor.

Si los datos más cercanos a la frontera se desea que tengan una salida ±1, la distancia de los datos al plano será $d = \frac{1}{\|\boldsymbol{w}\|}$, que será la distancia de H_1 y H_2 a *H*. Así pues, el margen del hiperplano *H* (distancia a los vectores más cercanos

pertenecientes a diferentes clases) a maximizar es $\frac{2}{\|\boldsymbol{w}\|}$. Maximizar este margen es equivalente a minimizar la norma de w [11]. De este modo, el problema de encontrar el hiperplano óptimo puede formularse como:

$$L(\boldsymbol{w}) = min\left\{\frac{1}{2}\|\boldsymbol{w}\|^2\right\} \tag{2.16}$$

con la restricción (2.14) que garantiza que el hiperplano separará las muestras de distintas clases. Si usamos los multiplicadores de Lagrange e incorporamos la restricción (2.13), la expresión a minimizar queda de la forma:

$$L_d = \frac{1}{2}\|\boldsymbol{w}^2\| - \sum_{i=1}^{N} \alpha_i (y_i(\boldsymbol{x_i} \cdot \boldsymbol{w} + b) - 1) \tag{2.17}$$

Así pues, para minimizar la expresión anterior debemos derivar con respecto a ***w*** y *b* e igualar a 0. Esto nos deja dos ecuaciones:

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x_i} \tag{2.18}$$

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{2.19}$$

Teniendo en cuenta las ecuaciones anteriores, la máquina de vectores soporte, actuando como clasificador y para el caso en el que el problema sea linealmente separable, puede escribirse como:

$$f(\boldsymbol{x}) = sign(\sum_{i=1}^{N} \alpha_i y_i (\boldsymbol{x_i x}) + b) \tag{2.20}$$

2.5.2 Caso no separable linealmente

En el apartado anterior hemos estudiado el caso en el que las muestras eran linealmente separables, es decir, era posible trazar una frontera lineal para delimitar ambas clases de datos. En el caso en el que esto no sea posible, hay dos soluciones: buscar una frontera no lineal, como veremos en el siguiente apartado, o tratar de encontrar el hiperplano que cometa un menor número de errores.

Para esta última opción, vamos a introducir unas variables positivas, ξ_i, que controlan el error permitido y penalizan las muestras mal clasificadas [12]. Así pues, la ecuación (2.14) es modificada de la forma:

$$y_i\{\boldsymbol{x_i} \cdot \boldsymbol{w} + b\} \geq 1 - \xi_i \tag{2.21}$$

$$\xi_i \geq 0, \forall\, i \tag{2.22}$$

De este modo, en las muestras bien clasificadas se cumplirá que $0 < \xi_i < 1$, dependiendo de cómo de cerca esté de la frontera la muestra, mientras que en las mal clasificadas $\xi_i > 1$.

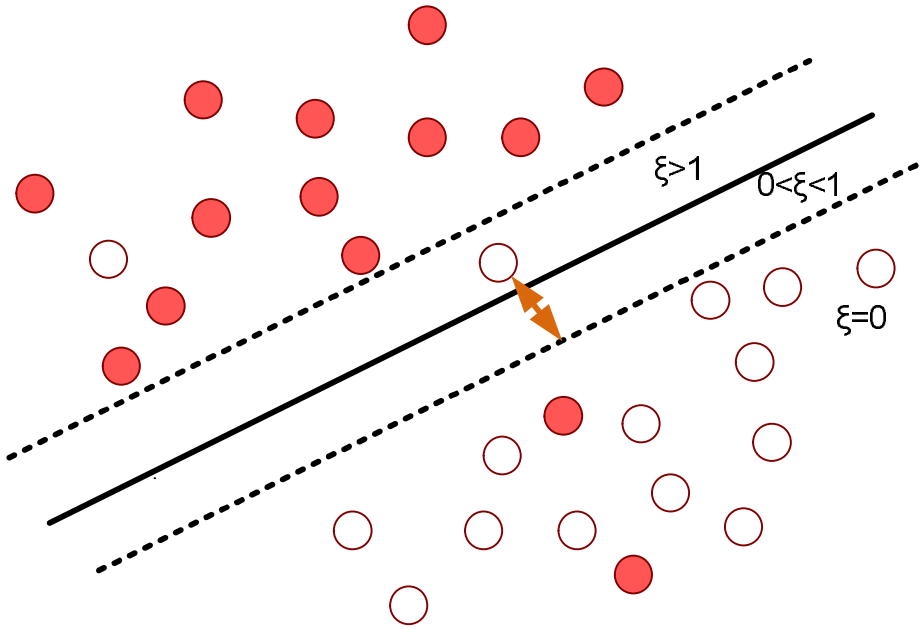

FIGURA 2.7 CASO NO SEPARABLE LINEALMENTE

Así pues, en esta ocasión la ecuación a minimizar (2.16), es modificada de la forma:

$$L_p = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N}\xi_i \tag{2.23}$$

sujeta a las restricciones (2.21) y (2.22). C es una constante que penaliza las muestras mal clasificadas y que determina la importancia de la maximización de la distancia frente a la minimización de los errores. En el presente proyecto, el valor

de esta constante se ha calculado mediante **validación cruzada**, tal y como se explicará en el siguiente capítulo.

Usando multiplicadores de Lagrange, la ecuación (2.23) queda de la forma:

$$L_{pd} = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\mu_i\xi_i - \sum_{i=1}^{N}\alpha_i(y_i(\boldsymbol{x_i}\cdot\boldsymbol{w} + b) - 1 + \xi_i) \quad (2.24)$$

donde μ_i es el multiplicador de Lagrange introducido para forzar la positividad de ξ_i. Si ahora derivamos con respecto a cada variable e igualamos a cero para obtener la solución óptima, como hicimos en el caso separable, obtenemos las ecuaciones (2.18), (2.19) y :

$$C - \mu_i - \alpha_i = 0 \quad (2.25)$$

El problema cuadrático queda de la forma:

$$L_d = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j y_i\, y_j \boldsymbol{x_i}, \boldsymbol{x_j} \quad (2.26)$$

2.5.3 SVM no lineal

La SVM no lineal se puede interpretar como una generalización del hiperplano óptimo de decisión, ya que permite la resolución de problemas no separables y trazar fronteras de clasificación no lineales [13]. Así pues, para el caso en el que los datos no son linealmente separables, puede aplicarse una transformación $\Phi(\boldsymbol{x})$ sobre el espacio de trabajo con el objetivo de obtener un **espacio de características**, generalmente de dimensión superior, donde sí sean separables las muestras y donde se debe trazar el hiperplano óptimo de separación. La formulación es básicamente la misma, únicamente reemplazando $\boldsymbol{x}$ por $\Phi(\boldsymbol{x})$.

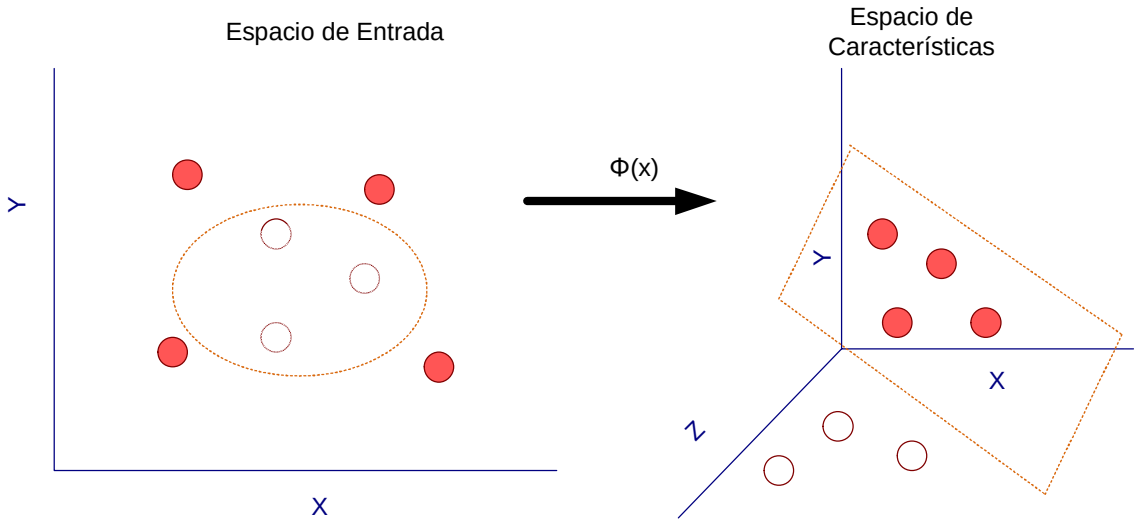

FIGURA 2.8 TRANSFORMACIÓN ESPACIAL DEL ESPACIO DE ENTRADA

El *Teorema de Cover* [14] proporciona la justificación de por qué una transformación no lineal a un espacio de mayor dimensión aumenta las posibilidades de disponer de un conjunto de datos separables, si estos no lo eran en el espacio de entrada. Para construir una SVM en el espacio resultante, este debe ser un *Espacio de Hilbert*[5], y por tanto cumplir:

$$k(\boldsymbol{x}, \boldsymbol{x_i}) = < \Phi(\boldsymbol{x}), \Phi(\boldsymbol{x_i}) > \qquad (2.27)$$

donde la función *k* es llamada *Función núcleo o* ***Función kernel***. Teniendo esta función, es posible aplicar el algoritmo de entrenamiento de las SVM sin conocer Φ . Existen distintas funciones kernel que permiten adaptar la SVM a cada conjunto de muestras, con el fin de obtener mejores resultados. Las más usadas son:

[5] El concepto de **espacio de Hilbert** es una generalización del concepto de *espacio euclídeo*. Esta generalización permite que nociones y técnicas algebraicas y geométricas aplicables a espacios de dimensión dos y tres se extiendan a espacios de dimensión arbitraria, incluyendo a espacios de dimensión infinita. Es por tanto un espacio Euclídeo de dimensión infinita.

- **Lineal**:

$$k(\boldsymbol{x}, \boldsymbol{x_i}) = \boldsymbol{x} \cdot \boldsymbol{x_i} \tag{2.28}$$

- **Polinómica**:

$$k(\boldsymbol{x}, \boldsymbol{x_i}) = (\gamma \boldsymbol{x} \cdot \boldsymbol{x_i} + c)^{\alpha} \tag{2.29}$$

- **Gaussiana**:

$$k(\boldsymbol{x}, \boldsymbol{x_i}) = exp(-\gamma |\boldsymbol{x} - \boldsymbol{x_i}|^2) \tag{2.30}$$

siendo γ una constante de proporcionalidad, c un coeficiente y α el rango del polinomio. En el presente trabajo se ha optado por un kernel gaussiano.

El incremento en el número de dimensiones debido a la transformación podría repercutir enormemente en el coste computacional, incrementándolo considerablemente [15]. Este incremento se verá suavizado por el hecho de que no es necesario trabajar en ese espacio, basta con conocer su producto escalar (2.14).

La función de clasificación de la SVM (2.20) es redefinida de la siguiente manera:

$$f(\boldsymbol{x}) = sign(\sum_{i=1}^{N} \alpha_i y_i (k(\boldsymbol{x}, \boldsymbol{x_i}) + b) \tag{2.31}$$

El *Teorema de Mercer* [16] expone que para garantizar que la función $k(\boldsymbol{x}, \boldsymbol{x_i})$ sea un producto interno en un espacio de alta dimensionalidad, se debe cumplir la condición:

$$\iint k(\boldsymbol{u}, \boldsymbol{v}) g(\boldsymbol{u}) g(\boldsymbol{v}) d\boldsymbol{u} d\boldsymbol{v} > 0 \tag{2.32}$$

es decir, la función kernel debe ser una función definida positiva. Por tanto, el problema cuadrático de entrenar la SVM queda de la forma:

$$L_d = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i \, y_j k(\boldsymbol{x_i}, \boldsymbol{x_j}) \tag{2.33}$$

sujeto a las restricciones:

$$0 \leq \alpha_i \leq C \tag{2.34}$$

$$\sum_i \alpha_i \, y_i = 0 \tag{2.35}$$

El vector del hiperplano resultante es:

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \Phi(\boldsymbol{x_i}) \tag{2.36}$$

Se puede dar el caso de que el problema tampoco sea separable en el nuevo espacio. En estos casos, se actúa de igual manera que en el problema lineal: introduciendo un término de coste C y las pérdidas ξ_i. Así pues, reescribiendo las ecuaciones (2.21) y (2.22) para el caso no lineal, obtenemos:

$$y_i\{\Phi(\boldsymbol{x_i}) \cdot \boldsymbol{w} + b\} \geq 1 - \xi_i \tag{2.37}$$

$$\xi_i \geq 0, \forall \, i \tag{2.38}$$

De este modo, la nueva función objetivo será la misma que en el caso lineal para problemas no separables:

$$min_{\boldsymbol{w},b,\varepsilon_i} \frac{1}{2}\|\boldsymbol{w}^2\| - C \sum_{i=1}^{N} \xi_i \tag{2.39}$$

Si introducimos las ecuaciones (2.37) y (2.38) en (2.39), el langrangiano del problema queda de la forma:

$$L_{pd} = \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \mu_i \xi_i - \sum_{i=1}^{N} \alpha_i (y_i(\Phi(\boldsymbol{x_i}) \cdot \boldsymbol{w} + b) - 1 + \xi_i) \tag{2.40}$$

ecuación similar a (2.24) en la que únicamente se ha sustituido $\boldsymbol{x_i}$ por $\Phi(\boldsymbol{x_i})$.

Las condiciones que debe cumplir un vector **w** para ser solución de (2.40) vienen determinadas por el ***Teorema de Karush-Kuhn-Tucker*** (KKT) [17] que es una generalización del Teorema de los Multiplicadores de Lagrange. Se puede decir que resolver el problema de la SVM es equivalente a encontrar una solución para las condiciones KKT. En el siguiente capítulo entraremos más en detalle sobre estas condiciones

2.6 Sequential Minimal Optimization

Aunque las SVMs están empezando a cobrar importancia en el aprendizaje máquina, aún no se ha generalizado su uso en el ámbito de la ingeniería. Hay dos posibles razones: primero, el entrenamiento de una SVM es lento, especialmente para problemas grandes, y segundo, los algoritmos de entrenamiento son complejos y en ocasiones difíciles de implementar [18].

En el presente proyecto, se ha utilizado un algoritmo que es conceptualmente sencillo, fácil de implementar y por lo general más rápido que los tradicionales métodos de entrenamiento de SVMs. Es el denominado algoritmo de Optimización Mínima Secuencial (*Sequential Minimal Optimization*) o SMO.

Como veíamos en el apartado anterior, entrenar una SVM requiere solucionar un problema de optimización cuadrática muy grande:

$$max_{\propto}\boldsymbol{w}(\propto) = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j y_i\, y_j k(\boldsymbol{x_i}, \boldsymbol{x_j})$$

$$0 \leq \alpha_i \leq C \qquad (2.41)$$

$$\sum_i \alpha_i\, y_i = 0$$

El algoritmo SMO soluciona este problema dividiéndolo en pequeños problemas de **programación cuadrática** (QP de sus siglas en inglés) los cuales soluciona de manera analítica, evitando así usar un optimizador de programación cuadrática iterativo como parte del algoritmo, lo cual consumiría mucho tiempo. La

cantidad de memoria que requiere la SMO es lineal al conjunto de datos de entrenamiento, lo que permite a este algoritmo poder trabajar con conjuntos de muestras muy grandes.

Así pues, dado el problema (2.41), se dice que un punto es óptimo si y sólo si cumple las condiciones de KKT. Estas condiciones son sencillas; el problema cuadrático se soluciona cuando para todo *i*:

$$\begin{aligned} \propto_i = 0 &\rightarrow y_i f(x) \geq 1 \\ 0 < \propto_i < C = 0 &\rightarrow y_i f(x) = 1 \\ \propto_i = C &\rightarrow y_i f(x) \leq 1 \end{aligned} \tag{2.42}$$

Debido a su gran tamaño, este problema de QP no puede resolverse fácilmente utilizando las técnicas tradicionales de resolución de QP debido a que esto implica la utilización de una matriz con un número de elementos igual al cuadrado del número de muestras de entrenamiento, lo que desbordaría la memoria.

Existen varios métodos para que el entrenamiento de la SVM tenga un menor coste computacional. Uno bastante popular es el algoritmo de **Chunking** [19] que consiste en tomar un conjunto arbitrario inicial de vectores a partir de los cuales se entrena la SVM. A continuación se toma un nuevo conjunto, incluyendo los vectores fuera del primer conjunto que más violen las condiciones KKT, y se itera hasta tener un conjunto óptimo. En cada iteración se resuelve un problema de QP que tiende a ser más grande en cada iteración. En última instancia, todos los multiplicadores de Lagrange distintos de cero han sido identificados y el problema es solucionado.

El algoritmo de descomposición propuesto por **Osuna** [20] es otro de los métodos para solventar el problema de QP que plantea el entrenamiento de una SVM. Este algoritmo descompone el problema grande de QP en una serie de pequeños problemas de QP. Cada uno de estos sub-problemas contiene al menos un ejemplo que viola las condiciones de KKT. Este método usa una matriz de tamaño constante para cada sub-problema lo que supone añadir y extraer el mismo número de ejemplos en cada paso. De este modo, una secuencia de sub-

problemas en los que siempre se añade al menos un violador de KKT convergerá asintóticamente.

A diferencia de los métodos de entrenamiento anteriormente descritos, la SMO resuelve en cada iteración el problema de optimización más pequeño posible, esto es, optimizar **dos multiplicadores de Lagrange simultáneamente**. Así, en cada iteración, encuentra los valores óptimos para esos multiplicadores y actualiza la SVM. La ventaja de este método es que la optimización de estos dos multiplicadores se realiza de manera analítica. La rápida resolución de estos sub-problemas hace que el problema principal de QP se resuelva igualmente rápido. Otro punto a su favor es que no requiere almacenar matrices adicionales, por lo que es posible trabajar con problemas con muchas muestras.

En la siguiente figura se realiza una comparación de los tres métodos. Para cada método se muestran tres iteraciones, que vienen representadas por las tres líneas finas. Cada línea representa el conjunto de entrenamiento, mientras que las cajas muestran los multiplicadores de Lagrange que son optimizados en ese paso. Vemos como la SMO optimiza únicamente dos multiplicadores en cada iteración a diferencia de los otros.

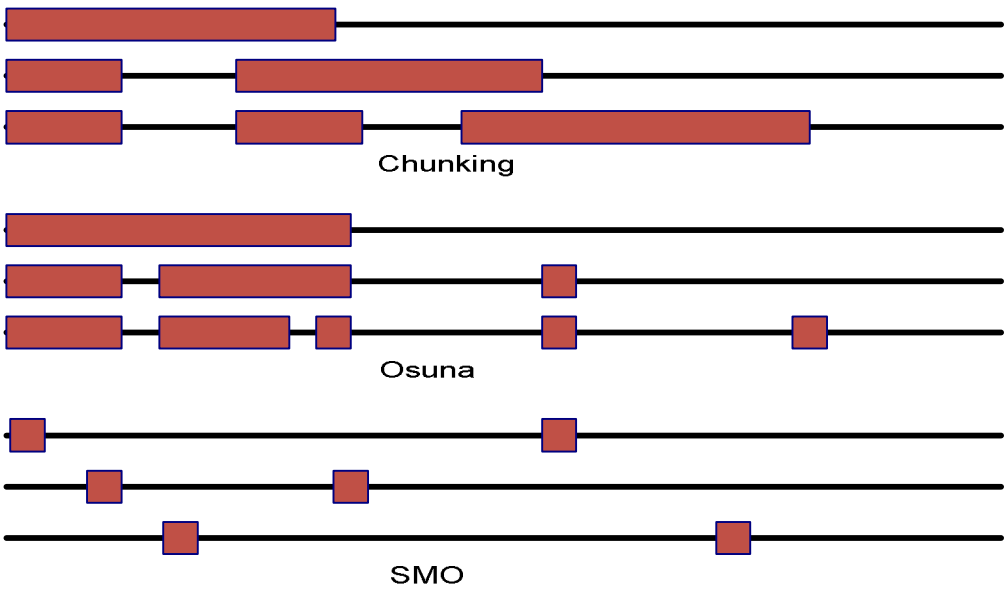

FIGURA 2.9 TRES MÉTODOS DE ENTRENAMIENTO DE SVM: CHUNKING, OSUNA Y SMO

El **método analítico** que emplea la SMO para solucionar el problema de optimizar los dos multiplicadores de Lagrange es el siguiente: lo primero que hace la SMO es calcular los valores máximos que pueden tomar ambos multiplicadores. Las restricciones vistas en (2.41) hacen que el valor que pueden tomar estos se sitúe en las diagonales de las cajas de la figura 2.10.

El algoritmo primero calcula $\propto_2$ así como el final de la diagonal en términos de $\propto_2$. Si el signo de $y_1$no es igual que el de y_2, es decir, pertenecen a clases distintas, las siguientes limitaciones inferiores y superiores se aplican sobre $\propto_2$:

$$L = \max(0, \propto_2^{old} - \propto_1^{old}) \qquad H = \min(C, C + \propto_2^{old} - \propto_1^{old}) \tag{2.43}$$

Si por el contrario, y_1 e y_2 son de la misma clase:

$$L = \max(0, \propto_1^{old} - \propto_2^{old} - C) \qquad H = \min(C, C + \propto_1^{old} + \propto_2^{old}) \tag{2.44}$$

Si la función objetivo es la misma en ambos extremos y el kernel obedece las condiciones de Mercer, entonces la optimización conjunto no puede hacer más progresos.

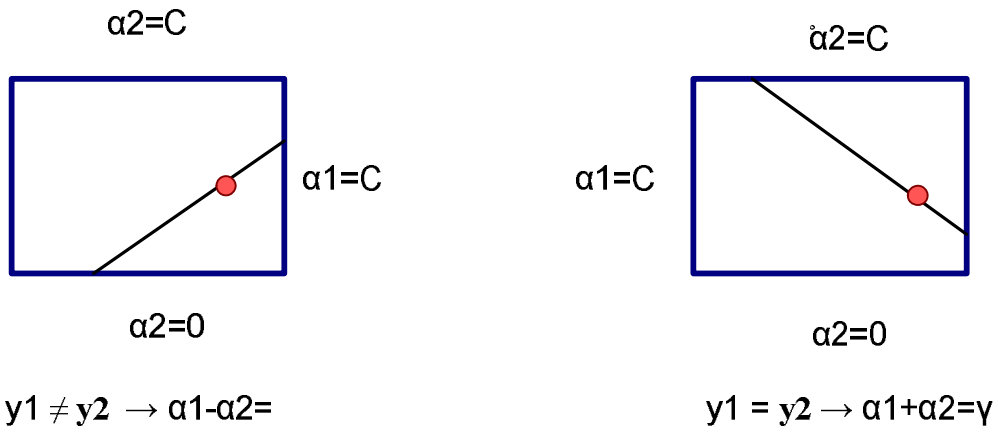

FIGURA 2.10 CÁLCULO DE ∝1 Y ∝2 USANDO SMO

El valor de $\propto_2$ es actualizado en cada iteración de la siguiente manera:

$$\propto_2^{new,clipped} = \begin{cases} H, & si \quad \propto_2^{new} \geq H; \\ \propto_2^{new}, & si \quad L < \propto_2^{new} < H \\ L, & si \quad \propto_2^{new} \leq L \end{cases} \tag{2.45}$$

donde,

$$\propto_2^{new} = \propto_2^{old} - \frac{y_2(E_1 - E_2)}{\eta} \tag{2.46}$$

[18] siendo E_i el error del i-ésimo ejemplo: $f^{old}(\boldsymbol{x_i}) - y_i$ y η la segunda derivada de la función objetivo sobre la diagonal:

$$\eta = 2k(\boldsymbol{x_1}, \boldsymbol{x_2}) - k(\boldsymbol{x_1}, \boldsymbol{x_1}) - k(\boldsymbol{x_2}, \boldsymbol{x_2}) \tag{2.47}$$

Los errores E_i de cada ejemplo se van almacenando en una caché. Por su parte, $\propto_1$ es actualizado de la siguiente manera:

$$\propto_1^{new} = \propto_1^{old} + y_1 y_2 \, (\propto_2^{old} - \propto_2^{new,clipped}) \tag{2.48}$$

La SMO siempre optimizará dos multiplicadores de Lagrange en cada iteración, con uno de los multiplicadores habiendo violado las condiciones de KKT previamente. De este modo, la función objetivo aumentará en cada iteración y el algoritmo convergerá asintóticamente.

Con el objetivo de acelerar esta convergencia, la SMO hace uso de la **heurística**[6] para seleccionar a los dos multiplicadores a optimizar. Concretamente se utilizan dos heurísticos: uno para la elección del primer multiplicador y otro para la elección del segundo.

Para elegir el **primer multiplicador** se usa un bucle, el cual recorre todas las muestras de entrenamiento en busca de un ejemplo que viole las condiciones de KKT. Una vez que lo encuentra, se selecciona mediante el segundo heurístico el segundo multiplicador y son conjuntamente optimizados. La SVM es actualizada

[6] Se denomina **heurística** a la capacidad de un sistema para realizar de forma inmediata innovaciones positivas para sus fines.

con los nuevos valores de los multiplicadores y el bucle prosigue la búsqueda de muestras que violen las condiciones de KKT.

Para aumentar la velocidad del entrenamiento, el bucle no siempre recorre todo el conjunto de muestras. Después de un primer recorrido por todos de datos, el bucle pasa a iterar únicamente por aquellos datos cuyos multiplicadores no son ni 0 ni C, los ejemplos no acotados. Aquí se repite el mismo proceso: de este grupo de muestras se cogen los que violen las condiciones de KKT y se optimizan. El bucle itera sucesivamente por los ejemplos no acotados hasta que todas las muestras cumplen las condiciones. Una vez que esto se produce, el bucle vuelve a iterar sobre todo el conjunto de entrenamiento. Esta alternancia se mantiene hasta que todos los datos cumplen las condiciones de KKT. Llegado ese punto, el algoritmo finaliza.

Una vez que ya se tiene el primer multiplicador, la SMO busca un **segundo multiplicador** que maximice la optimización conjunta. Para ello se fija en (2.46) para ver con qué multiplicador da un mayor paso. Como evaluar la función de kernel consume mucho tiempo, la SMO aproxima el paso dado calculando únicamente $|E_1 - E_2|$. De este modo, se selecciona un error que maximice ($|E_1 - E_2|$). Si E_1 es positivo, la SMO selecciona una muestra que tenga un error E_2 mínimo mientras que si E_1 es negativo, se elige un dato con un error E_2 máximo.

En algunas ocasiones la heurística para elegir a este segundo multiplicador puede no provocar progresos positivos, como por ejemplo en el caso de que las dos muestras cogidas sean idénticas. Para evitar estas situaciones, la SMO utiliza una jerarquía para encontrar un segundo multiplicador con el que se hagan progresos positivos al optimizarlo conjuntamente con el primer multiplicador. Esta jerarquía es como sigue: (**A**) si el heurístico descrito anteriormente no hace progresos positivos, la SMO empieza a probar con los ejemplos no acotados. (**B**) Si no encuentra ninguna muestra con la que se hagan progresos positivos, pasa a probar con todo el conjunto de datos hasta que con algún dato se hagan. Tanto en (A) como en (B), la búsqueda comienza desde posiciones **aleatorias**. Si tampoco se produce un progreso positivo, lo cual es sumamente extraño, la primera muestra es sustituida por otra.

Cuando un multiplicador de Lagrange está siendo optimizado, su caché de error es puesta a 0. Siempre que se produce una optimización, los errores almacenados de todos los multiplicadores no acotados que no están implicados en la optimización son actualizados de la siguiente manera:

$$E_k^{new} = E_k^{old} + y_1\left(\propto_1^{new} - \propto_1^{old}\right)k(\boldsymbol{x_1}, \boldsymbol{x_k}) + y_2\left(\propto_2^{new,clipped} - \propto_2^{old}\right)k(\boldsymbol{x_2}, \boldsymbol{x_k}) + \\ + b^{old} - b^{new} \tag{2.49}$$

De esta manera, cuando la SMO necesita conocer el valor de un error E_i, busca en la caché de errores si el correspondiente multiplicador de Lagrange no está en los límites. De lo contrario evaluará la actual función de decisión de la SVM basándose en el vector de $\propto$ actual.

Es importante decir que en los sistemas tradicionales de reconocimiento no es necesario que se cumplan totalmente las condiciones de KKT para obtener una alta precisión. En el caso de la SMO, el algoritmo verifica las condiciones de KKT con un margen de error ε, el cual suele tomar unos valores comprendidos entre 0.01 y 0.001. Si queremos obtener una salida más precisa, el algoritmo se ralentizará.

2.7 Prior-SVM

La prior-SVM es una variación de la Máquina de Vectores Soporte realizada en 2004 por Parrado-Hernández, Ambroladze y Shawe-Taylor [21]. Se trata de modificar el problema de manera que se introduzca conocimiento *a priori* en la SVM y adaptar la SMO para que lo resuelva. La motivación es obtener mejores resultados de clasificación que la SVM basándose en las conclusiones extraídas de [22], donde se señala que es más sensato tener un clasificador *a priori* bueno.

Así pues, la prior-SVM está diseñado para resolver el siguiente problema, que como observamos es una modificación de (2.39):

$$min_{\boldsymbol{v},\eta,\varepsilon_i}\left[\frac{1}{2}\|\boldsymbol{v}\|^2 + C\sum_{i=1}^{m-r}\varepsilon_i\right] \tag{2.50}$$

sujeto a las restricciones:

$$y_i(\boldsymbol{v} + \eta \boldsymbol{w_r})^T \Phi(\boldsymbol{x_i}) \geq 1 - \xi_i \qquad i = 1, \dots, m - r \tag{2.51}$$

$$\xi_i \geq 0, \forall\, i \quad \text{i=1,...,m-r}$$

El clasificador final será $\boldsymbol{w} = \boldsymbol{v} + \eta \boldsymbol{w_r}$. Vemos como el conocimiento *a priori* se plasma con la introducción de un nuevo clasificador, $\boldsymbol{w_r}$, cuya relevancia en la clasificación final dependerá de η. De este modo, si el clasificador *a priori* no obtiene los resultados esperados, η tendrá un valor más bajo y la clasificación dependerá principalmente de $\boldsymbol{v}$.

Si procedemos como en casos anteriores y usamos los multiplicadores de Lagrange $\propto_i$ para las principales restricciones y v_i para ξ_i, obtenemos después de derivar con respecto a **v**, ξ_i y η:

$$\boldsymbol{v} - \sum_{j=1}^{m-r} \propto_j y_j \boldsymbol{\Phi}(\boldsymbol{x_i}) = 0 \tag{2.52}$$

$$0 \leq \propto_j \leq C \qquad j = 1, \dots, m - r \tag{2.53}$$

$$\sum_{j=1}^{m-r} \propto_j y_j \boldsymbol{\Phi}(\boldsymbol{x_j}) \cdot \boldsymbol{w_r} = 0 \tag{2.54}$$

Sustituyendo obtenemos:

$$max_{\propto_i} \sum_{i=1}^{m-r} \propto_i - \frac{1}{2} \sum_{i,j=1}^{m-r} \propto_i \propto_j y_i y_j \boldsymbol{\Phi}(\boldsymbol{x_i})^{\boldsymbol{T}} \boldsymbol{\Phi}(\boldsymbol{x_i}) \tag{2.55}$$

sujeto a:

$$\sum_{i=1}^{m-r} y_i \propto_i g_i = 0 \qquad i = 1, \dots, m - r \tag{2.56}$$

$$0 \leq \propto_i \leq C \qquad i = 1, \dots, m - r \tag{2.57}$$

donde $g_i = \sum_{k=m-r+1}^{m} \propto_k y_k k(\boldsymbol{x_i}, \boldsymbol{x_k})$ y $\propto_k$ son las variables normalizadas anteriormente aprendidas para los últimos *r* ejemplos. Una vez que hemos calculado los $\propto_k$, calculamos η usando la ecuación:

$$y_j \left(\sum_{i=1}^{m-r} \propto_i y_i k(\boldsymbol{x_i}, \boldsymbol{x_j}) + \eta g_j \right) = 1 \tag{2.58}$$

El clasificador final queda de la forma:

$$f(\boldsymbol{x}) = sign(\sum_{i=1}^{m-r} \propto_i y_i k(\boldsymbol{x_i}, \boldsymbol{x}) + \eta \sum_{k=m-r+1}^{m} \propto_k y_k k(\boldsymbol{x_k}, \boldsymbol{x})) \quad (2.59)$$

Capítulo 3

PROPUESTA: DISEÑO DEL SOFTWARE

Una vez sentadas las bases teóricas en el capítulo anterior, en este se va a mostrar en detalle el trabajo realizado para solucionar el problema propuesto. Comenzaremos realizando un planteamiento inicial del problema y analizando las opciones elegidas para su resolución. Tras esto se explicará en qué consisten las funciones MEX, que nos serán de gran utilidad, y pasaremos a explicar cada uno de las funciones programadas, tanto en lenguaje C como en el entorno MATLAB.

3.1 Planteamiento

El punto de partida del proyecto es un código programado en **MATLAB** en el que se entrena la máquina Prior-SVM mediante la técnica de SMO. Como se ha explicado anteriormente, el objetivo del trabajo es, por un lado, el de reprogramar, completar y mejorar este código en un lenguaje de medio-bajo nivel con el fin de aumentar la velocidad de ejecución y poder así trabajar con muestras más grandes, y, por otro lado, integrarlo en una completa *toolbox* en MATLAB para entrenar y clasificar muestras de manera sencilla.

MATLAB [23] es el nombre abreviado de "*MATrix LABoratory*". Se trata de un software matemático de alto nivel para realizar cálculos numéricos con vectores y matrices. Para ciertas operaciones es muy rápido, cuando puede ejecutar sus funciones en código nativo con los tamaños más adecuados para aprovechar sus capacidades de vectorización. En otras aplicaciones resulta bastante más lento que el código equivalente desarrollado en otros lenguajes como C/C++ o Fortran. Sin embargo, siempre es una magnífica herramienta de alto nivel para desarrollar aplicaciones técnicas, fácil de utilizar (a diferencia de los lenguajes de bajo nivel) y que aumenta significativamente la productividad de los programadores respecto a otros entornos de desarrollo. De este modo, MATLAB es el lenguaje idóneo para programar un entrenamiento de pocas muestras de una SVM: sencillo, intuitivo, capaz de llevar a cabo numerosas operaciones matemáticas, rápido de programar y que trabaja muy bien con vectores y matrices.

El problema viene cuando el programa contiene bucles, ya que estos consumen mucho tiempo. Esto se acentúa si se emplean muchos datos. En esos casos la ejecución es extremadamente lenta, por lo que se suele optar por programar el entrenamiento en un lenguaje de más bajo nivel. Esto es una tarea más compleja ya que estos lenguajes no están diseñados específicamente para trabajar con matrices y operaciones matemáticas, por lo que el código será mucho más largo y no tan sencillo e intuitivo como MATLAB.

El lenguaje elegido para reprogramar el código ha sido **C**. Se trata de un lenguaje estructurado de nivel medio, ni de bajo nivel como ensamblador, ni de alto nivel como Ada o MATLAB. Esto permite una mayor flexibilidad y potencia, a cambio de menor abstracción.

Ya que el objetivo del proyecto era hacer un programa que consumiera el menor tiempo posible, se ha programado de la forma más eficiente posible. Una de las cosas en las que se ha hecho más hincapié, ha sido el acceso a memoria. Aunque en MATLAB las matrices se introducen por filas, se almacenan por columnas, mientras que en C se almacena por filas. Este detalle es de suma importancia en nuestro código, ya que vamos a tener que recorrer numerosas matrices y lo más rápido es acceder a la siguiente posición de memoria.

Como se ha comentado anteriormente, se va a crear una *toolbox* en MATLAB para entrenar y clasificar datos mediante Prior-SVM. Esta *toolbox* debe contener, entre otras funciones, nuestro código reprogramado en C. Para conseguir esto se recurre a los **ficheros MEX**. Los ficheros MEX [24] permiten llamar desde MATLAB a funciones escritas en C (también en C++ o Fortran) como si fueran funciones propias de MATLAB.

Para la depuración del código, se ha empleado el comando `debug` de MATLAB para los ficheros *.m*, mientras que para las funciones MEX escritas en C se han utilizado dos tipos de depuraciones. Por un lado, desde MATLAB es posible depurar este tipo de funciones mediante los comandos `mex -g filename.c` y `msdev filename.dll` escritos en el *prompt* de MATLAB y en la ventana de comando de DOS respectivamente. Una vez hecho esto, el código se podrá ser depurado en cualquier editor. Otra forma ha sido transformando las funciones MEX en funciones normales de C, sustituyendo el método *mexFunction* por un *main*.

Una herramienta de gran utilidad empleada para mejorar la velocidad del código ha sido el *profiler* de MATLAB. Se trata de una utilidad que permite conocer cuánto tiempo se ha invertido en cada función y en cada línea de código de un fichero **.m*. El *profiler* mejora la calidad de los programas, pues permite detectar los "cuellos de botella". Por ejemplo, sabiendo el número de veces que se llama a una función y el tiempo que cuesta cada llamada, se puede decidir si es mejor emplear más memoria en guardar resultados intermedios para no tener que calcular varias veces lo mismo. Esta herramienta genera un documento con el resumen de los resultados, el *Profile Summary*. A continuación se muestran un par de capturas de pantalla del mismo.

Profile Summary

Generated 14-Feb-2010 11:44:45 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
EjecutarnSVMMatlab	1	17.077 s	0.588 s	
...s.mlwidgets.workspace.WhosInformation (Java-method)	1	0.001 s	0.001 s	
decisor (MEX-function)	1	0.709 s	0.709 s	
etapSVMSMOmatlab	1	14.339 s	1.108 s	
etapSVMSMOmatlab>examineExample	29377	13.232 s	9.215 s	
etapSVMSMOmatlab>takeStep	10649	3.948 s	3.786 s	
etapSVMSMOmatlab>updateOBJ	3182	0.022 s	0.022 s	
gaussiankernelmatlab	2	1.167 s	1.022 s	
mean	1212	0.140 s	0.140 s	
randperm	916	0.074 s	0.074 s	
repmat	4	0.145 s	0.145 s	
svmpredict (MEX-function)	3	0.209 s	0.209 s	
svmtrain (MEX-function)	1	0.059 s	0.059 s	
workspacefunc	4	0.096 s	0.002 s	
workspacefunc>createComplexScalar	19	0.006 s	0.006 s	
workspacefunc>getShortValueObjectJ	9	0.024 s	0.012 s	
workspacefunc>getShortValueObjectsJ	1	0.027 s	0.003 s	
workspacefunc>getStatObjectJ	18	0.051 s	0.001 s	

FIGURA 3.1 EJEMPLO DEL USO DEL *PROFILER* DE MATLAB 1

```
    time    calls  line
                    8 function porcentajenSVMMatlab= EjecutarnSVM
                    9 %****************************************
                   10
< 0.01          1   11 [Nr,dr]=size(xtrain);
< 0.01          1   12 Nt=length(xtest);
                   13 %****************************************
                   14
  0.02          1   15 a=randperm(Nr);
< 0.01          1   16 b=randperm(Nt) ;
                   17
< 0.01          1   18 xtrain=xtrain(a,:);
< 0.01          1   19 ytrain=ytrain(a,:);
< 0.01          1   20 xtest=xtest(b,:);
< 0.01          1   21 ytest=ytest(b,:);
                   22 %****************************************
                   23
< 0.01          1   24 sigma=sqrt(dr);
< 0.01          1   25 eps=0.001;
< 0.01          1   26 sizeCache=100;
                   27 %****************************************
                   28
< 0.01          1   29 xr1=xtrain(1:round(Nr/2),:);
< 0.01          1   30 xr2=xtrain(round(Nr/2)+1:Nr,:);
< 0.01          1   31 yr1=ytrain(1:round(Nr/2),:);
< 0.01          1   32 yr2=ytrain(round(Nr/2)+1:Nr,:);
< 0.01          1   33 Nr1=length(xr1)
< 0.01          1   34 Nr2=length(xr2)
                   35 %****************************************
                   36
                   37 %gamma = 1/(2*sigma^2);
                   38
< 0.01          1   39 switch (C)
< 0.01          1   40     case 1
                   41         model = svmtrain(yr1, xr1, '-s 0 -t
< 0.01          1   42     case 10
                   43         model = svmtrain(yr1, xr1, '-s 0 -t
< 0.01          1   44     case 100
  0.07          1   45         model = svmtrain(yr1, xr1, '-s 0 -t
```

FIGURA 3.2 EJEMPLO DEL USO DEL *PROFILER* DE MATLAB 2

3.2 Funciones MEX

Como se ha comentado anteriormente, los ficheros MEX permiten llamar desde MATLAB a funciones escritas en C como si fueran funciones propias de MATLAB. Al compilar estas funciones se generan librerías compartidas, las denominadas funciones MEX. Estas funciones son ejecutables de extensión **.dll* ó **.mexw32* [7] que pueden ser cargadas y ejecutadas por MATLAB de forma automática.

Las funciones MEX tienen varias aplicaciones:

- Evitan tener que reescribir en MATLAB funciones que ya han sido escritas en C.
- Por motivos de eficiencia puede ser interesante reescribir en C las funciones críticas que consumen más CPU del programa.

Para poder trabajar con los ficheros MEX, primero hay que seleccionar el compilador que se va a utilizar. Esto se hace mediante la siguiente instrucción en el *prompt* de MATLAB:

```
>> mex –setup
```

El compilador utilizado en el presente proyecto ha sido Microsoft Visual C++ 6.0. La sintaxis del comando para compilar y crear un fichero MEX a partir de lo que se va a llamar un fichero C-MEX (un fichero C que cumple las condiciones necesarias para poder crear con él un fichero MEX) es la siguiente:

```
>> mex filenamea.c –output filenameb.dll
```

donde `filenamea.c` es el nombre del fichero en C y `filenameb.dll` es el nombre del fichero MEX que se va a generar.

El código fuente de un fichero MEX programado en C tiene dos partes. La primera parte contiene el código de la función C que se quiere implementar como fichero MEX. La segunda parte es la función *mexFunction* que hace de interfaz entre

[7] Las funciones MEX tienen una extensión diferente en función de los sistemas operativos en que hayan sido generadas. Tanto *.dll como *.mexw32 son para Windows. La extensión *.dll es para versiones de Matlab anteriores a la 7.1. En Linux se usa la extensión *.mexglx.

C y MATLAB y que sustituye al *main* de C. Los ficheros MEX deben incluir la librería "*mex.h*" donde está declarada la función *mexFunction*, que tiene la siguiente cabecera:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

Los cuatro argumentos tienen los siguientes significados:

1. ***prhs*** es un vector de punteros a los valores de los argumentos de entrada (*right hand side arguments*) que se van a pasar a la función C.
2. ***nhrs*** es el número de argumentos de entrada de la función.
3. ***plhs*** y ***nlhs*** son análogos a los anteriores pero referidos a los argumentos de salida (*left hand side arguments*).

La función MEX trabaja con matrices, independientemente del tipo de dato a almacenar. Son las denominadas *mxArrays*. En el diagrama mostrado a continuación se observa cómo se realiza la comunicación entre C y MATLAB, cómo llegan los datos al fichero MEX, qué es lo que se hace con estos datos en la *mexFunction* y cómo se devuelven finalmente los resultados a MATLAB. Para ello se sirve de un ejemplo.

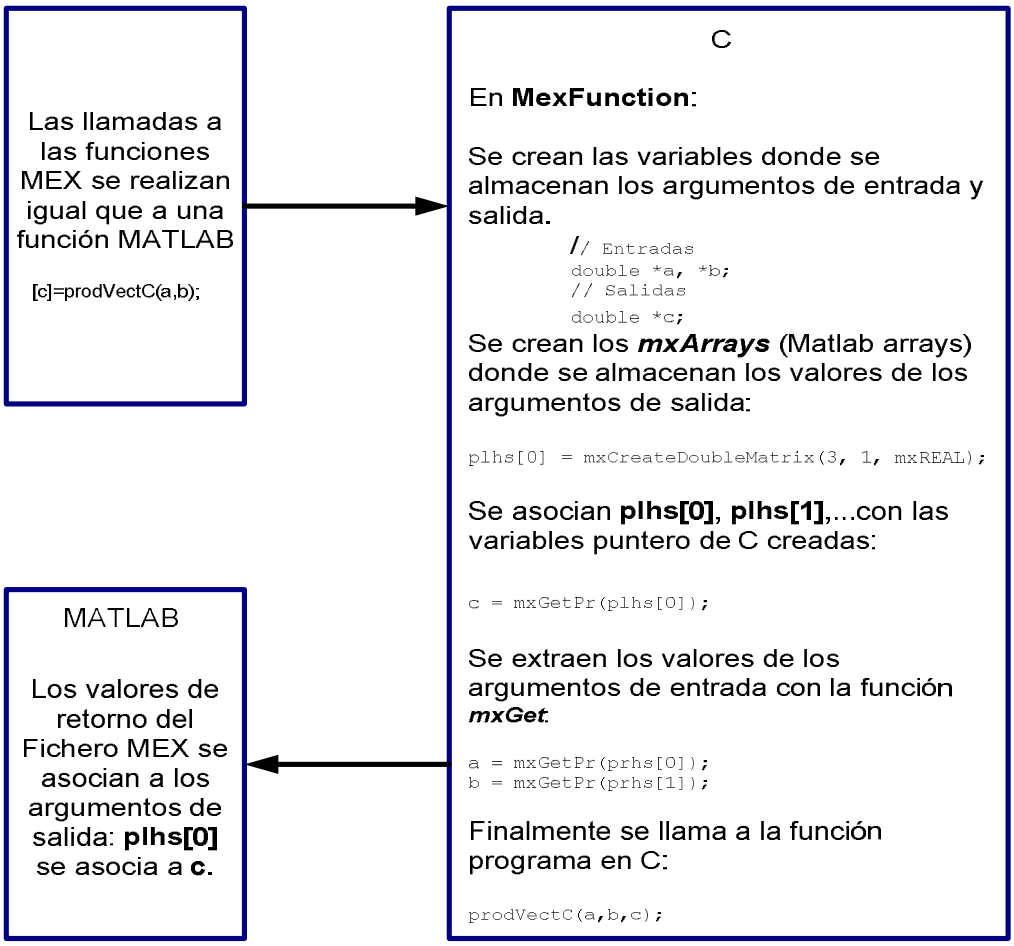

FIGURA 3.3 DIAGRAMA DE BLOQUES DEL PROGRAMA

3.3 Diseño de software

En este apartado se van a explicar cada una de las funciones y scripts programados para la *toolbox* de Prior-SVM que se pretende desarrollar. Como se ha comentado con anterioridad, esta *toolbox* contendrá las funciones en C (funciones MEX) que entrenan la máquina así como otras escritas en MATLAB que complementan la *toolbox* y que por su sencillez se ha optado por programarlas directamente en ese lenguaje.

El código original estaba compuesto únicamente por dos funciones: una que calculaba la matriz de kernel a partir de unos datos de entrada y otra que realizaba el entrenamiento de la SVM y devolvía el vector **w** *y* η a partir de la matriz de kernel y los datos de entrenamiento con sus etiquetas. En el presente proyecto, además de crear sus equivalentes en C, se ha creado de forma modular una completa *toolbox* para no sólo entrenar muestras sino también para clasificar ejemplos mediante la Prior-SVM. De este modo, se han programado diferentes funciones para cargar los datos y compilar las funciones, calcular el valor óptimo del parámetro C mediante validación cruzada, calcular la matriz de kernel y entrenar los datos de entrenamiento, clasificar las muestras de *test*, así como un script que es el eje central del programa y que va llamando a todas estas funciones. Para la clasificación *a priori* se ha empleado la librería LIBSVM.

A continuación se muestra el diagrama de bloques que esquematiza el trabajo realizado. Las cajas de color azul representan las funciones programadas en MATLAB, las verdes las escritas en C y la roja funciones externas, es decir, no programadas en este proyecto.

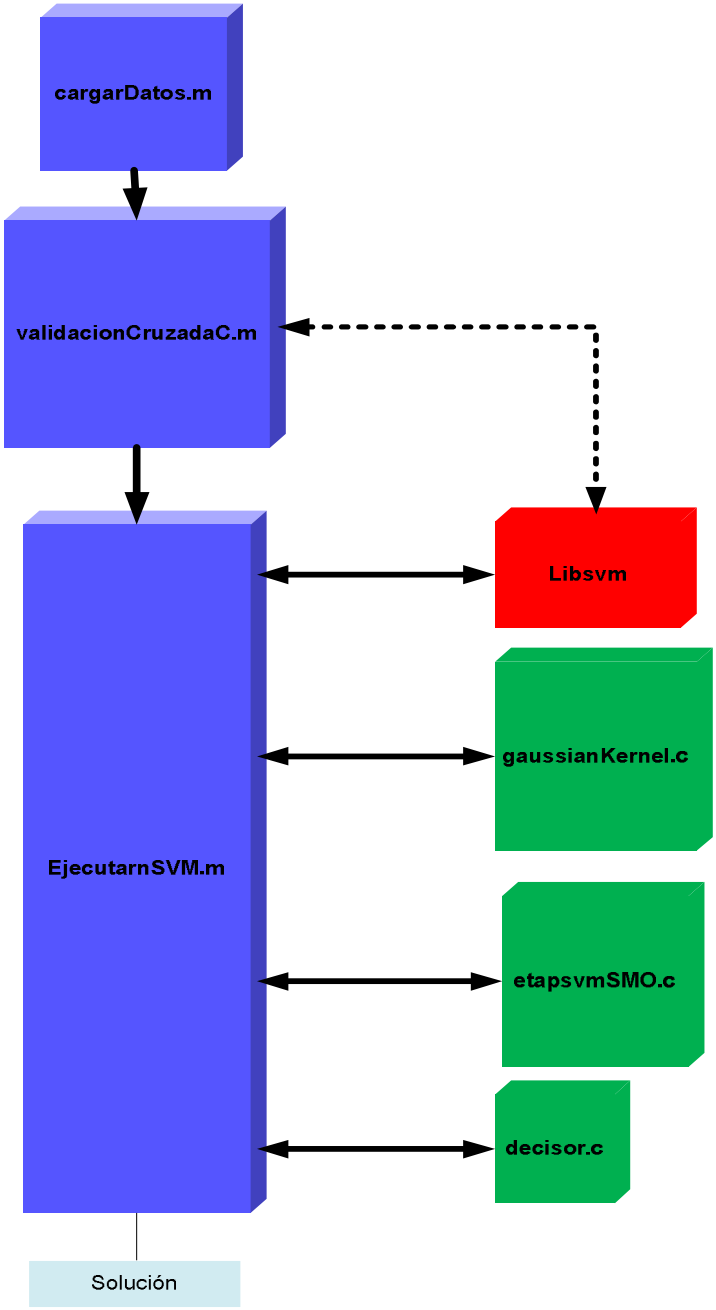

FIGURA 3.4 DIAGRAMA DE BLOQUES DEL SISTEMA

Como vemos, la *toolbox* está compuesta por varias funciones interrelacionadas entre sí las cuales pueden ser usadas de manera independiente y que se llaman desde el entorno MATLAB. El orden lógico de trabajo con esta *toolbox* de MATLAB es el siguiente:

1. Llamada a la función *cargardatos.m*, la cual cargará los datos de entrenamiento y *test* en el entorno MATLAB.
2. Llamada a *validacionCruzada.m* para obtener la C óptima con la que entrenar dichos datos. Para calcular esta constante, la función llama varias veces a la librería LIBSVM (línea de puntos) con los datos de entrenamiento y se queda con la C con la que se obtienen mejores resultados. La razón por la que se utiliza aquí LIBSVM en lugar de nuestro programa es porque es más rápido y debido a que tenemos que entrenar varios conjuntos de datos, se ahorra bastante tiempo, especialmente si la muestra a entrenar es grande.
3. Se llama a *EjecutarnSVM.m*, la cual realizará el entrenamiento de la Prior-SVM mediante SMO y clasificará las muestras. Para ello:
   3.1. Se llama a LIBSVM para obtener la clasificación *a priori*.
   3.2. Mediante *gaussianKernel.c* se obtiene la matriz de kernel.
   3.3. Se llama a *etapsvmSMO.c* para obtener el clasificador *a posteriori* y el valor de η. A esta función se le pasa entre otras variables, el resultado de la clasificación *a priori*.
   3.4. Con todo lo anterior, se llama a *decisor.c* para obtener la salida del clasificador final

Es importante resaltar que cada función descrita anteriormente se puede llamar por separado pasándole los parámetros adecuados, sin tener que seguir el orden anteriormente citado. Las funciones *gaussianKernel.c* y *etapsvmSMO.c* son las funciones reprogramadas en C. Constituyen la parte más importante, pues se encargan del entrenamiento de la máquina.

En las siguientes secciones vamos a explicar más en detalle cada una de las funciones de la *toolbox*.

3.3.1 cargarDatos.m

Como su nombre indica, este script carga los datos de entrenamiento y de *test* en el entorno de MATLAB mediante el comando `load`. El conjunto de datos debe tener una extensión *.mat* y estar compuesto por:

- Dos variables, denominadas ***xr*** y ***xt***, que constituyen los datos de entrenamiento y *test* respectivamente. Con los primeros se entrenará la máquina y con los segundos evaluaremos el clasificador resultante. Cada una de estas variables es un array de tipo *double* cuyos componentes pueden tomar cualquier valor real positivo o negativo. Cada fila es un vector que representa un dato/muestra/ejemplo, mientras que cada columna es una característica de ese dato.

- Dos variables, ***yr*** e ***yt***, que son las etiquetas de los datos de entrenamiento y *test* respectivamente. Los primeros nos servirán para entrenar la máquina, mientras los segundos los usaremos para conocer el porcentaje de aciertos de la máquina. Se trata de vectores columna que pueden tomar únicamente el valor -1 ó +1, dependiendo de la clase a la que pertenezca la muestra asociada. Debido a que el clasificador que vamos a implementar es binario, únicamente habrá dos posibles clases.

3.3.2 validacionCruzadaC.m

Se trata de un script que, una vez cargados los datos de entrenamiento en el entorno MATLAB, calcula la C óptima con la que entrenar la SVM. Recordemos que como se mencionó en la sección 2.3.2, C es una constante que penaliza las muestras mal clasificadas. De este modo, cuanto mayor es C, más importancia se da a los errores de clasificación.

Este cálculo se realiza mediante validación cruzada[8] y comprende los siguientes pasos:

1. Se definen los C candidatos. Esto es, el conjunto de valores de C entre los cuales se va a decidir el más idóneo para usar en el entrenamiento. Aunque es configurable, se han propuesto los siguientes candidatos por defecto: 1, 5, 30, 60, 200.
2. Se dividen los datos de entrenamiento, ***xr***, y las etiquetas, ***yr***, en dos mitades: ***xr1***, ***xr2***, ***yr1***, ***yr2***.
3. Se llama a *svmtrain.m*, de la librería LIBSVM, para entrenar la SVM pasándole ***xr1*** e ***yr1*** como datos de entrenamiento así como el primero de los candidatos de C.
4. Se llama a *svmpredict.m*, también de LIBSVM, pasándole ***xr2*** e ***yr2*** como datos de *test*. La salida de la función es el porcentaje de aciertos obtenidos al entrenar la SVM con ***xr***1 y probar la máquina con ***xr2***.
5. Se repiten los pasos 3 y 4 pero esta vez se pasan ***xr2*** e ***yr2*** como datos de entrenamiento y ***xr1*** e ***yr****1* como datos de *test*.
6. Se hace la media de los porcentajes obtenidos en ambos casos y se almacena.
7. Se inicializa la variable Copt a este primer candidato.
8. Se repiten los pasos 3, 4, 5 y 6 pasándole el siguiente valor candidato de C. Si el porcentaje obtenido es mejor, se actualiza la variable Copt. Esto se repite con todos los candidatos.
9. La C resultante será la Copt al final del programa.

En el caso en el que se quiera realizar el entrenamiento directamente con una determinada C, que no tiene por qué ser la óptima, en vez de llamar a *validaciónCruzada.m*, bastará con que en el *prompt* de MATLAB se inicialice el valor de C o se le pase por parámetro a *etapsvmSMO.c*.

[8] La **validación cruzada**, es la práctica estadística de partir una muestra de datos en subconjuntos de tal modo que el análisis es inicialmente realizado en uno de ellos, mientras los otros subconjuntos son retenidos para su uso posterior en la confirmación y validación del análisis inicial.

3.3.3 EjecutarnSVM.m

Una vez que ya tenemos las funciones MEX compiladas, los datos cargados y la constante C seleccionada, podemos comenzar el entrenamiento de la SVM y la clasificación de muestras. Esto es realizado por una serie de funciones que son llamadas desde *EjecutarnSVM.m*, cuya cabecera es:

```
function porcentajenSVM = EjecutarnSVM(C,xtrain,ytrain,xtest,ytest,sigma);
```

Como vemos, esta función escrita en MATLAB recibe los datos de entrenamiento y los de *test*, junto con sus etiquetas correspondientes, la constante C y sigma. Devuelve el porcentaje de muestras bien clasificadas.

Es importante decir que si el parámetro de entrada `sigma`, es cero. El valor que se utilizará en el programa será la raíz cuadrada del número de columnas de los datos de entrenamiento.

Una de las funciones a las que se llama desde aquí es *gaussianKernel.c*, la cual, como veremos en la sección 3.3.5, calcula la matriz de kernel necesaria para entrenar la SVM. Una matriz de kernel tiene una dimensión de *NxN*, donde *N* es el número de los datos. El problema aquí es que operar con esta matriz es muy poco eficiente, especialmente si el número de muestras es elevado, ya que en realidad sólo se van a utilizar las filas de la matriz de kernel de los Vectores Soporte. De este modo, si por ejemplo tenemos, en una situación ideal, un problema de 5000 datos y sólo 100 Vectores Soporte y además sabemos que estos 100 Vectores Soporte son los 100 primeros datos, entonces sólo habría que calcular una matriz de 100x5000. En esta situación, calcular una matriz de 5000x5000 sería extremadamente ineficiente.

Para tratar de solventar este problema se ha creado una **caché de kernel** de tamaño configurable. Así, en vez de alojar espacio en la memoria para *NxN*, se fija un tamaño máximo a reservar (por defecto se ha puesto la mitad del tamaño de los datos). Así, si por ejemplo se tiene capacidad para almacenar 500 filas (vectores) y la SMO pide una fila de la matriz de kernel, primero busca en la caché si la tiene ya

calculada y si no la tiene, elimina la fila de la caché que se haya accedido menos veces y en su lugar pone la fila que necesita calcular, calculándola en ese momento.

Los pasos que sigue la función son:

1. Se barajan los datos de entrenamiento. Se intercambian las filas para que la distribución estadística que pudieran tener las muestras no tenga consecuencias en el entrenamiento.
2. Se asigna el valor de ε, margen de error, a un valor por defecto de ε=0.001. Si la sigma introducida por parámetro es nula, se utiliza el valor $\gamma = \sqrt{dr}$, siendo *dr* el número de columnas de los datos de entrenamiento o número de características de cada muestra.
3. Se fija el tamaño de la caché de kernel. Aunque se puede modificar, por defecto tendrá capacidad para N/2 muestras, donde N es el número total de datos a entrenar.
4. Se divide el conjunto de entrenamiento en dos mitades iguales: ***xr1***, ***yr1*** y ***xr2***, ***yr2***.
5. Se llama a *svmtrain.c*, perteneciente a LIBSVM, para obtener el clasificador *a priori* ***w1***. La máquina es entrenada con ***xr1*** e ***yr1***.
6. Normalizamos ***w1*** dividiéndolo por su norma.
7. Se calcula la salida blanda del clasificador ***w1*** normalizado para ***xr2*** llamando a la función *svmpredict.c* de LIBSVM. A esta clasificación *a priori* le llamaremos ***g***.
8. Se calcula la caché de kernel llamando a *gaussiankernel.c* y pasándole por parámetro sigma y las filas de ***xr1*** y ***xr2*** que quepan en la caché.
9. Llegado a este punto, se llama a *etapsvmSMO.c*, el algoritmo programado para entrenar la SVM. A esta función le pasamos, entre otras cosas, ***xr2*** e ***yr2*** como conjunto de entrenamiento y g como resultado del clasificador a priori. Devuelve η y el clasificador *a posteriori* ***w2***. Esta es la función principal del programa. La estudiaremos más en detalle en 3.3.6.
10. El clasificador final será ***w=eta*w1 + w2***, por lo que para predecir el conjunto de *test*:

a. Se calcula la salida blanda del clasificador ***w1*** normalizado para el conjunto de *test*, ***xt***. Esta salida se le llamará ***ot***.
b. Se calcula la salida blanda del clasificador ***w2*** para el conjunto de *test*, ***xt***. Esta salida es ***ft***.
c. La salida del clasificador conjunto es ***eta*ot+ft***.

11. Hallamos su salida dura, ***yy***.
12. Se compara *yy* con las etiquetas de los datos de *test*, ***yt***, y se calcula el porcentaje de aciertos.

3.3.4 LIBSVM

LIBSVM es una librería para SVMs creada en 2001 por Chih-Chung Chang y Chih-Jen Lin [25]. Se trata de un programa que soporta estimación de distribución, clasificación y regresión de Vectores Soporte. En el presente proyecto se han utilizado dos de las funciones de esta librería:

- ***svmtrain.c***. Es una función que realiza el entrenamiento de unos datos en función de unos parámetros. Desde *EjecutarnSVM.m* se llama a esta función de la siguiente manera:

```
model = svmtrain(yr1, xr1, '-s 0 -t 2 -g 0.0250 -c 10');
```

Como vemos, además de los datos de entrenamiento y sus respectivas etiquetas, hay una serie de parámetros configurables. En nuestro caso se han utilizado:

- *-s 0* indica que es un problema de SVM de tipo C-SVC.
- *-t 2* indica la utilización de un kernel de tipo gaussiano.
- *-g 0.0250* hace referencia al grado del kernel gaussiano, γ. El valor a introducir aquí es $g = \frac{1}{2*\gamma^2}$.
- *–c 10* valor del parámetro C, en este caso 10.

La función devuelve *model*, que es un conjunto de datos con los resultados del entrenamiento, entre los cuales está **w**.

- ***svmpredict.c***. Esta otra función de LIBSVM se encarga, una vez realizado el entrenamiento, de clasificar un conjunto de muestras. Desde *EjecutarnSVM.m* se le llama de la siguiente manera:

```
[predicted_label, accuracy, g] = svmpredict(yr2, xr2, model);
```

La función recibe por parámetro los datos a clasificar y el modelo obtenido en el entrenamiento. Devuelve la predicción de la clase para cada muestra, la precisión y la salida blanda.

El uso de estas dos funciones *externas* se debe a la necesidad de contar con una clasificación *a priori*, g, la cual será utilizada en el clasificador *a posteriori*.

En el sitio web de LIBSVM además de poder adquirir el software para diferentes entornos, se proporcionan manuales, un *applet*[9] para probar el programa y numerosos conjuntos de datos multiclase utilizables para SVMs.

3.3.5 gaussianKernel.c

Esta función MEX tiene como objetivo calcular la matriz de kernel. Recordemos que mediante esta matriz pasamos de un espacio de entrada con muestras no separables a un espacio caracterizado donde los datos son separables.

La cabecera de esta función es como sigue:

```
double* gaussianKernel(double* xa,double* xb,double ss);
```

Como podemos ver, recibe dos conjuntos de datos y sigma (`ss`). Devuelve la caché de kernel.

[9] Un ***applet*** es un componente de una *aplicación* que se ejecuta en el contexto de otro programa, por ejemplo un navegador web.

3.3.6 etapsvmSMO.c

Esta es la función más importante de todo el programa, pues es la que se encarga del entrenamiento, mediante SMO, de la máquina. Desde *EjecutarnSVM.m* se le llama de la siguiente manera:

```
[w2,eta,pos]=etapsvmSMO(g,k,yr2,eps,C,indicesCache,usoCache,xr2,xr1,sigma);
```

Como vemos recibe numerosos parámetros:

- ***g*** es el resultado de la clasificación *a priori*. Recordemos que Prior-SVM es una modificación de la máquina SVM y que utiliza conocimiento *a priori*.
- ***k*** es la matriz de kernel.
- ***yr2*** son las etiquetas de los datos a entrenar ***xr2***.
- ***eps*** es el ε que se utiliza en el entrenamiento.
- ***C*** es la constante que penaliza las muestras mal clasificadas.
- ***indicesCache*** es el vector que contiene los índices de las muestras almacenadas actualmente en la caché.
- ***usoCache*** es el vector que contiene el número de veces que se ha accedido a cada componente de la caché.
- ***xr2***, ***xr1*** y ***sigma*** son datos necesarios para calcular el kernel de una muestra en el caso de que no estuviese almacenada en la caché.

La función comprende tres rutinas jerarquizadas: *etapsvmSMO*, *examineExample* y *takeStep*, que se relacionan entre sí de la siguiente manera:

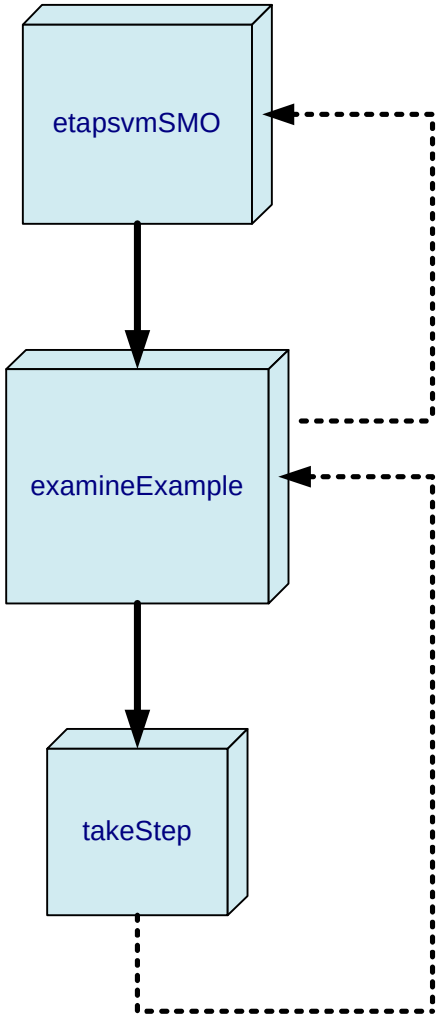

FIGURA 3.5 DIAGRAMA DE BLOQUES DE LA FUNCIÓN ETAPSVMSMO.C

Como vemos, la rutina principal recibe el mismo nombre que la función: etapsvmSMO. En ella se selecciona el primer multiplicador de Lagrange y se llama a *examineExample* para encontrar el segundo. Una vez que se tienen ambos, desde *examineExample* se llama a *takeStep* para intentar una optimización conjunta y actualizar los valores de los multiplicadores. Si hay avances positivos, *examineExample* devolverá un 1 a la rutina principal y si no un 0.

A continuación se explica más en detalle el funcionamiento de cada rutina, incluyendo el pseudocódigo en el que se ha basado el diseño:

- ***etapsvmSMO***

Es la rutina principal. Aquí se inicializan todas las variables globales. Su función principal es ir eligiendo en cada iteración al primer multiplicador, para lo cual se usa un bucle que recorre todas las muestras de entrenamiento buscando un ejemplo que viole las condiciones de KKT. Tras hacer un primer recorrido por todos los datos, el bucle pasa a iterar únicamente por las muestras no acotadas, aquellas cuyos multiplicadores no son ni 0 ni C. En el momento en el que no se produzcan avances positivos, se vuelve a iterar por todas las muestras. Esta alternancia se mantiene hasta que todos los datos cumplen las condiciones de KKT, es decir, cuando se ha examinado todo (`examinarTodo=0`) y no hay avances (`numCambiado=0`). Llegado ese punto, el algoritmo finaliza.

```
etapsvmSMO:
      Inicializa el vector de alpha a cero
      Inicializa el umbral a cero
      numCambiado=0;
      examinarTodo=1;
      while(numCambiado>0 | examinarTodo){
            numCambiado=0;
            if(examinarTodo)
                  itera I por todos los datos de entrenamiento
                  numCambiado += examineExample(I)
            else
            itera I por todo los datos con 0<alpha<C
            numCambiado +=examinarEjemplo(I)
            if (examinarTodo==1)
                  examinarTodo=0
            else if (numCambiado==0)
```

```
            examinarTodo=1;
      }
```

- ***examineExample***

Una vez que ya se tiene el primer multiplicador, en esta rutina se selecciona el segundo y se llama a *takeStep* para la optimización conjunta. Si se produce un avance devolverá 1, si no 0. La cabecera es la siguiente:

```
int examineExample(int i2);
```

Como se explicó en el apartado 2.6, la elección de este segundo optimizador se hace de modo que se maximice $|E_1 - E_2|$. Si esta heurística falla y no produce avances positivos (*takeStep* devuelve 0), vemos en el pseudocódigo como recurre primero a hacer un "barrido" por los ejemplos no acotados y si tampoco hay avances recurre a iterar por todas las muestras. Una vez seleccionado qué multiplicador va a ser optimizado, se llama a *takeStep* con ambos. El pseudocódigo de esta rutina es la siguiente:

```
examineExample(i2)
      y2=signo[i2]
      alph2=multiplicador de Lagrange de i2
      E2= salida SVM de la muestra i2-y2 (chequear en errorCache)
      r2=E2*y2
      if((r2<-tol && alph2<C) || (r2>tol && alph2>0)) {
            if(número de alpha ≠ 0 y C > 1){
                  i1=resultado de la elección tras el segundo
                  heurístico
                  if takeStep(i1,i2)
                        return 1
            }
            Itera por todos los alpha ≠ de 0 y C, empezando en un
            punto aleatorio
```

```
        {
                i1=identidad del alpha actual
                if takeStep(i1,i2)
                      return 1
        }
        Itera por todos los i1 posibles, empezando en un punto
        aleatorio
        {
                i1=bucle variable
                if takeStep(i1,i2)
                return 1
        }
    }
    return 0
    Fin rutina
```

- **takeStep**

En esta rutina se lleva a cabo la optimización siguiendo los conceptos teóricos vistos en el capítulo 2.6. Si observamos el pseudocódigo mostrado a continuación, cuando no se producen avances positivos, la rutina devuelve 0 a *examineExample*. Esto se produce, por ejemplo, si ambas muestras son iguales.

En el caso de que sí se pueda optimizar correctamente ambos multiplicadores, se calculan *L* y *H*, los valores máximo y mínimo que pueden tomar los multiplicadores según las restricciones (2.41). Tras esto se procede a calcular $\propto_2$, para lo cual anteriormente se ha hecho lo propio con η (2.47), a partir de los kernel de las dos muestras. Por último, se actualiza el valor del primer multiplicador, el vector de la caché de errores y el clasificador **w**.

El pseudocódigo es el siguiente:

```
takeStep(i1,i2)
      if(i1==i2) return 0
      alph1= multiplicador de Lagrange de i1
      y1=signo[i1]
      E1=salida de SVM del punto[i1]
      S=y1*y2
      Cálculo de L,H
      if(L==H)
            return 0
      k11=kernel(i1,i1)
      k12=kernel(i1,i2)
      k22=kernel(i2,i2)
      eta=2*k12-k11-k22
      if(eta<0){
            a2=alph2-y2*(E1-E2)/eta
            if(a2<L) a2=L
            else if (a2>H)a2=H
      }
      else{
            Lobj= función objetivo en a2=L
            Hobj= función objetivo en a2=H
            if(Lobj>Hobj+eps)
                  a2=L
            else if(Lobj<Hobj-eps
                  a2=H
            else
                  a2=alph2
      }
if(a2<1e-8)
      a2=0
```

```
else if (a2>C-1e-8)

      a2=C

if(|a2-alph2|<eps*(a2+alph2+eps))

      return 0

a1=alph1+s*(alph2-a2)

Actualizo vector de pesos para reflejar los cambios en a1 y a2

Actualizo la caché de errores usando nuevos multiplicadores de
Lagrange

Almaceno a1 en el vector de alpha

Almaceno a2 en el vector de alpha

return 1

Final rutina
```

3.3.7 decisor.c

Se trata de una función MEX que se encarga, una vez que se ha entrenado la máquina, de clasificar las muestras de *test*. Es equivalente a *svmpredict.c* de LIBSVM con la que obteníamos la salida blanda *a priori ot* con el clasificador ***w1***. Esta función es llamada desde *EjecutarnSVM.m* para obtener la salida *blanda (ft*) de los datos de *test* (***xt***) en el clasificador *a posteriori* (***w2***). Tiene la forma:

```
ft=decisor(k2,w2,pos,indicesCache,usoCache,xtest,xr2,sigma);
```

La función recibe los datos de *test* y usando la matriz de kernel (***k2***) y el clasificador previamente calculados, devuelve la decisión blanda. Una vez tenemos la decisión blanda, bastará con quedarnos con el signo de cada clasificación, para obtener la decisión dura (2.31).

Como vemos, además de los parámetros anteriormente citados, a la función se le pasan más variables (*pos, indicesCache, usoCache, xr2, sigma*), que están relacionados con el uso de la caché de kernel. Ya que no estamos calculando la

matriz entera de kernel, necesitamos las herramientas necesarias para calcular, si fuese necesario, algún punto de la matriz que no esté en la caché.

3.3.8 EjecutarnSVMMatlab.m

Esta función es similar a *EjecutarnSVM.m*, que como mencionamos anteriormente, era la encargada de ir llamando a todas las funciones involucradas en el entrenamiento y clasificación de la máquina. La diferencia radica en que el entrenamiento y el cálculo de la matriz de kernel no se realiza en funciones escritas en C, sino en MATLAB. Así, desde *EjecutarnSVMMatlab.m* se llama a *gaussiankernelmatlab.m* y *etapSVMSMOmatlab.m*, en vez de a *gaussianKernel.c* y *etapsvmSMO.c*.

El objetivo de esta función es poder comparar la velocidad de ejecución de ambas implementaciones de Prior-SVM de una forma justa, de ahí que se haya creado el mismo entorno. La cabecera de esta función es:

```
function porcentajenSVMMatlab=EjecutarnSVMMatlab(C,xtrain,ytrain,xtest,ytest);
```

que como observamos recibe los datos de entrenamiento y *test* y el parámetro C, exactamente lo mismo que *EjecutarnSVM.m*.

3.3.9 EjecutarLIBSVM.m

Esta función escrita en MATLAB realiza tanto el entrenamiento de una máquina SVM mediante SMO, como la clasificación de datos, ambas cosas mediante la librería LIBSVM. Se encarga por tanto de llamar a *svmpredict.c* y *svmtrain.c*, funciones explicadas en el apartado 3.3.4. La cabecera es como sigue:

```
function porcentajenLIBSVM=EjecutarnLIBSVM(C,xtrain,ytrain,xtest,ytest);
```

La finalidad de esta función es la de poder comparar, tanto en velocidad como en prestaciones, una máquina SVM, que no tiene conocimiento *a priori*, con una Prior-SVM que sí hace uso de este conocimiento.

Así pues, se ha creado el mismo entorno de ejecución para los tres escenarios (C, MATLAB y LIBSVM), con el fin de realizar comparaciones justas.

3.3.10 Otras funciones

- `function [w,eta_out,pos]=etapsvmSMO(gr,Ka,Ya,epsa,C)`

  Esta función es llamada desde *EjecutarnSVMMatlab.m* para el entrenamiento de Prior-SVM. Se trata de la función original en la que se ha basado la función MEX *etapsvmSMO.c*. Como vemos, recibe por parámetro los resultados de la clasificación *a priori* (***gr***), la matriz de kernel (***Ka***), las etiquetas de los datos a entrenar (***Ya***), épsilon (*epsa*) y el valor del parámetro *C*. Devuelve el clasificador (**w**), η (*eta_out*) y la posición de los Vectores Soporte (***pos***).

- `function K=gaussiankernelmatlab(xr,xc,ss)`

  Es la otra función de la que se ha partido en el presente proyecto. Escrita en MATLAB, recibe dos conjuntos de datos (*xr* y *xc*) y γ. Devuelve la matriz de kernel.

- `compilaC.m`

  Script que compila las funciones MEX, incluidas las pertenecientes a LIBSVM.

Capítulo 4

VALIDACIÓN EXPERIMENTAL

En este capítulo se va a llevar a cabo una validación experimental del trabajo realizado. Se empezará por describir bajo qué condiciones se van a realizar las pruebas y se procederá a comparar los tiempos de entrenamiento de la máquina Prior-SVM en función del tamaño de la caché empleada. Tras esto, pasaremos a comparar la velocidad de ejecución de la versión en C y en MATLAB. Por último compararemos una SVM con una Prior-SVM y analizaremos la importancia o no de introducir conocimiento *a priori*.

4.1 Descripción de los experimentos

Los experimentos llevados a cabo en este proyecto han sido realizados en el sistema operativo Windows XP, en un ordenador *Pentium Centrino* con un procesador de 1,73 GHz y 1GB de RAM. Las características del ordenador influirán directamente en el tiempo empleado por cada programa.

Como se ha comentado, para llevar a cabo una comparación lo más justa posible entre las implementaciones en C y MATLAB, se han programado dos funciones similares (*EjecutarnSVMMatlab.m* y *EjecutarnSVM.m*), en donde la única

diferencia radica en que el cálculo de la matriz de kernel y el entrenamiento de la máquina por el mecanismo de la SMO es realizado, en una en MATLAB y en la otra en C. Es decir, ambos programas seguirán exactamente los mismos pasos y únicamente serán distintas sus implementaciones de *etapsvmSMO* y *gaussianKernel*. También se ha creado una tercera función, *EjecutarnLIBSVM.m*, que llama a las funciones necesarias de la librería LIBSVM para entrenar y clasificar la máquina SVM. Esta función será utilizada en el apartado 4.4 para la comparación entre SVM y Prior-SVM.

La función que entrena la Prior-SVM, tanto en MATLAB como en C, tiene partes aleatorias al seleccionar los multiplicadores a optimizar, tal y como se describe en los heurísticos de la SMO en el capítulo anterior. Esto hace que el tiempo de ejecución para un mismo conjunto de datos, varíe de una ejecución a otra, ya que en cada una, los bucles realizan un número diferente de iteraciones. Esta variación se hace más notoria cuanto mayor sea el número de muestras. Es por esto por lo que para algunos experimentos se suprimirá esta aleatoriedad, de forma que obtengamos comparaciones más justas.

Para calcular el tiempo de ejecución de cada programa, se ha utilizado el comando `cputime`, mediante el cual se contabiliza el tiempo desde que se entra en la función en cuestión hasta que se muestran los resultados en el *prompt* de MATLAB.

La siguiente figura es una captura de pantalla en la que se muestra un ejemplo de uso de la *toolbox* con el entrenamiento programado en C. En ella observamos cómo se cargan los datos, se selecciona la C óptima y se entrena y clasifican las muestras, mostrando por pantalla los resultados obtenidos y el tiempo transcurrido.

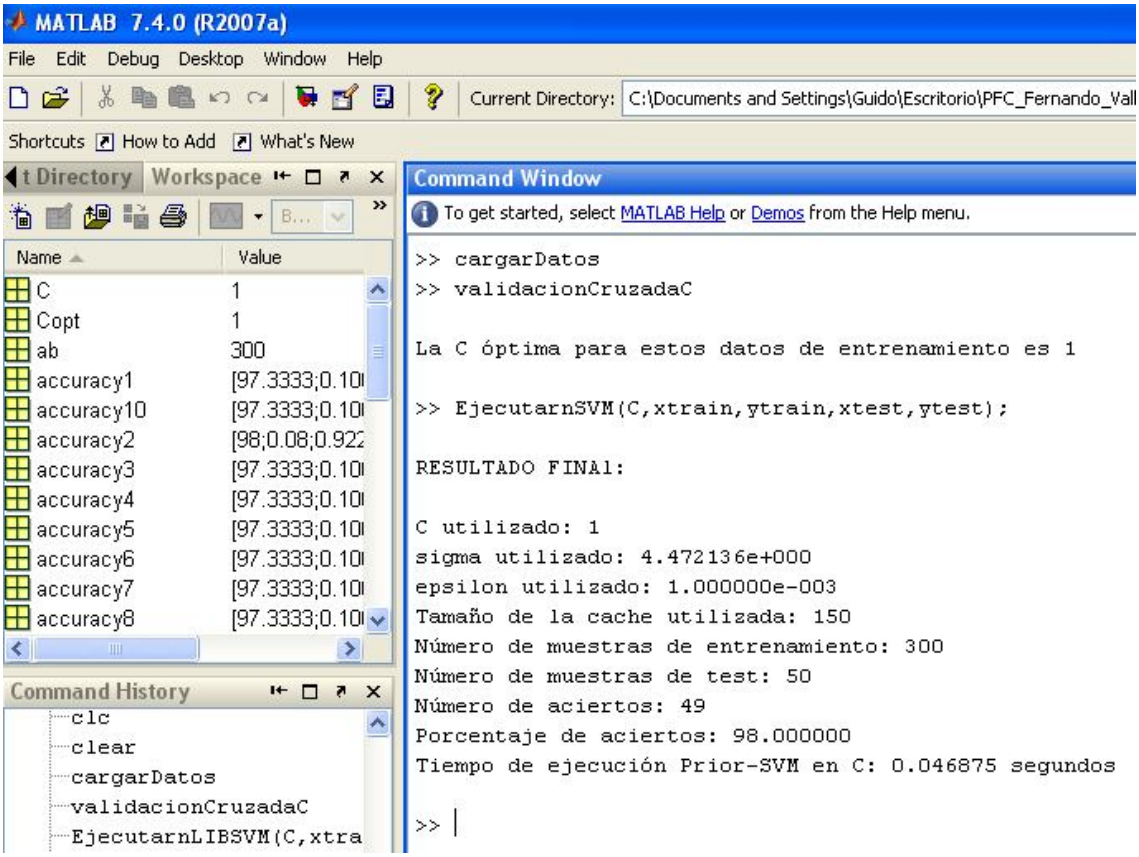

FIGURA 4.1 CAPTURA DE PANTALLA DE LA *TOOLBOX* CON ENTRENAMIENTO EN C

A continuación se muestra el uso de la *toolbox* con el entrenamiento de Prior-SVM llevado a cabo en MATLAB. El formato salida de los datos es similar al anterior.

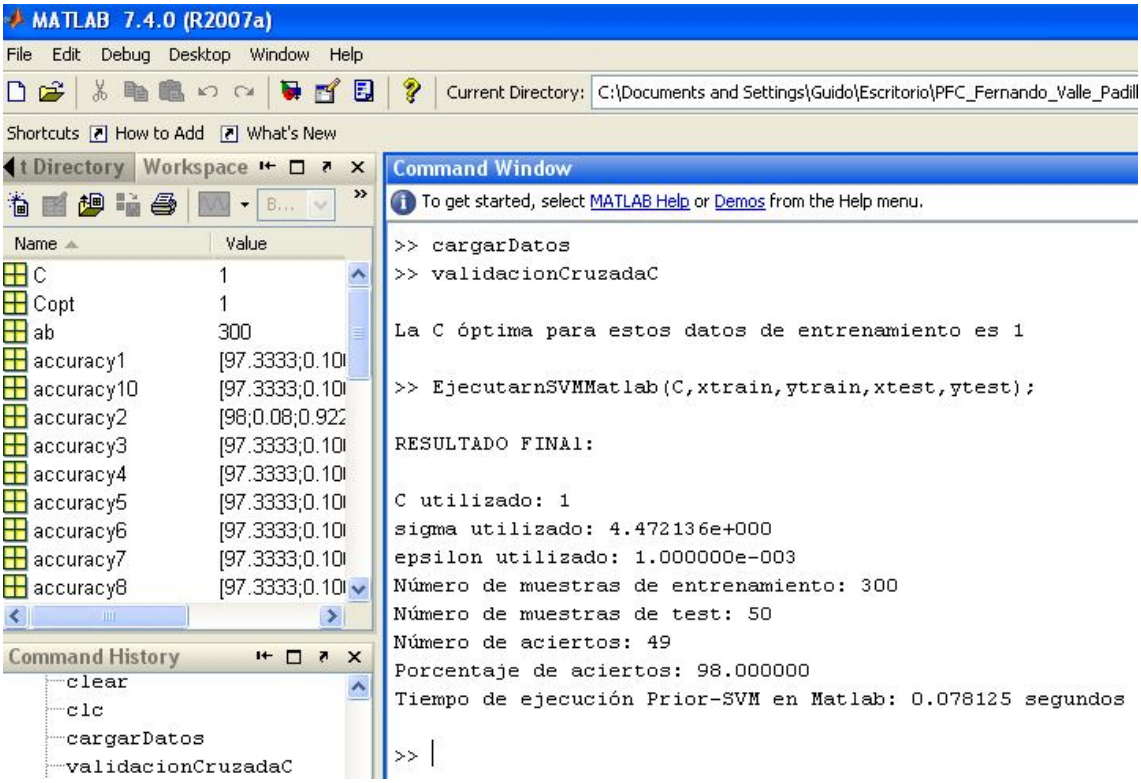

FIGURA 4.2 CAPTURA DE PANTALLA DE LA *TOOLBOX* CON ENTRENAMIENTO EN MATLAB

Por último, en la figura 4.3 observamos una captura de pantalla de un ejemplo de uso de la librería LIBSVM.

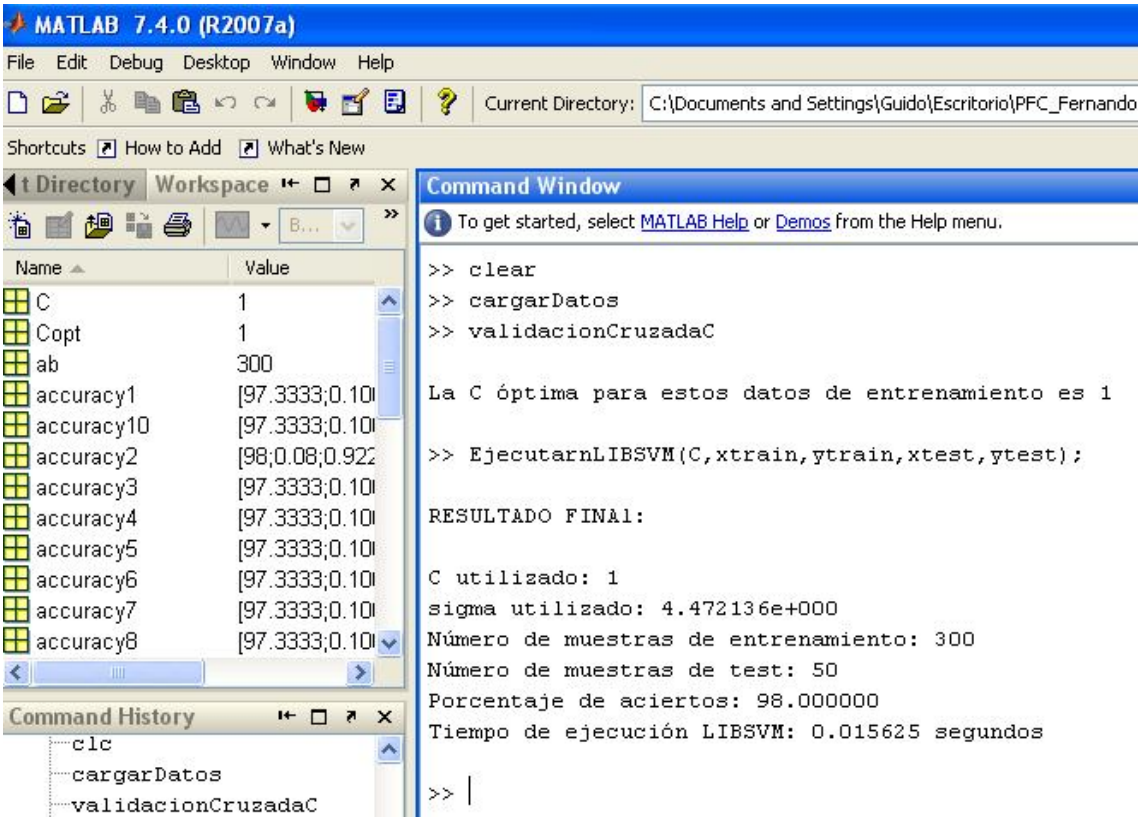

FIGURA 4.3 CAPTURA DE PANTALLA DE LA *TOOLBOX* DE LIBSVM

Como vemos, las tres *toolboxes* se han diseñado de manera similar para facilitar la comparación de resultados. En el ejemplo mostrado, similar en los tres casos, se han entrenado 300 datos, el parámetro C seleccionado ha sido *C=1* y se ha probado el clasificador con 50 muestras. En los tres casos ha habido un 98% de muestras bien clasificadas. Observamos también como la implementación en C es prácticamente el doble de rápida que su equivalente en MATLAB y como el tiempo de LIBSVM es el más bajo de las tres implementaciones.

Los datos para los experimentos se han obtenido de [26], donde se pueden encontrar numerosos datos para experimentos de clasificación. Como nuestra *toolbox* trata problemas binarios, se han modificado algunos de estos conjuntos de datos que eran multiclase para adaptarlos a nuestro problema. Los conjuntos de muestras tendrán distintas dimensiones, *NxJ*, donde N hace referencia al número de ejemplos/vectores y J al número de columnas o atributos de cada vector.

Los conjuntos de datos utilizados en los experimentos se pueden ver en la siguiente tabla, junto con una breve descripción de los mismos:

Nombre	Descripción
Vino	Datos de 3 cosechas distintas. Se clasifica, en función de 14 propiedades químicas, entre las muestras pertenecientes a la primera cosecha y las pertenecientes a las otras dos.
Escritura	Cada muestra representa un carácter escrito por una persona. Cada carácter presenta 16 atributos. Se clasificará, en función de estos atributos, entre aquellos caracteres que han sido escritos por la persona 1 y cuales por las otras 9.
Satélite	El conjunto de datos es una imagen de satélite. Para cada pixel, se clasificará, en función de 34 atributos, entre los pixeles que son carretera y los que no lo son.
Reconocimiento Carácter	Utilizando 20 fuentes distintas, se han escrito las letras del abecedario. A partir de 16 propiedades, se va a clasificar ente aquellas muestras que representen la letra A y las que no.
Telescopio	Los datos representan una imagen de un telescopio. A partir de 11 atributos, se clasificará cada pixel entre los que muestren una partícula γ y los que no.

TABLA 4.1 DESCRIPCIÓN DE LOS DATOS UTILIZADOS

4.2 Comparativa Caché de Kernel

En este apartado se va a realizar un experimento para estudiar la relación existente entre el tiempo de ejecución del programa desarrollado en C y el tamaño de la caché utilizado. Para ello, se emplearán unos datos de entrenamiento que tienen una dimensión de 5920x20 y se irá aumentando progresivamente el tamaño de la caché. Los datos de *test* serán de 1480x20. Se van a promediar cinco ejecuciones para cada tamaño de caché para conseguir resultados más justos.

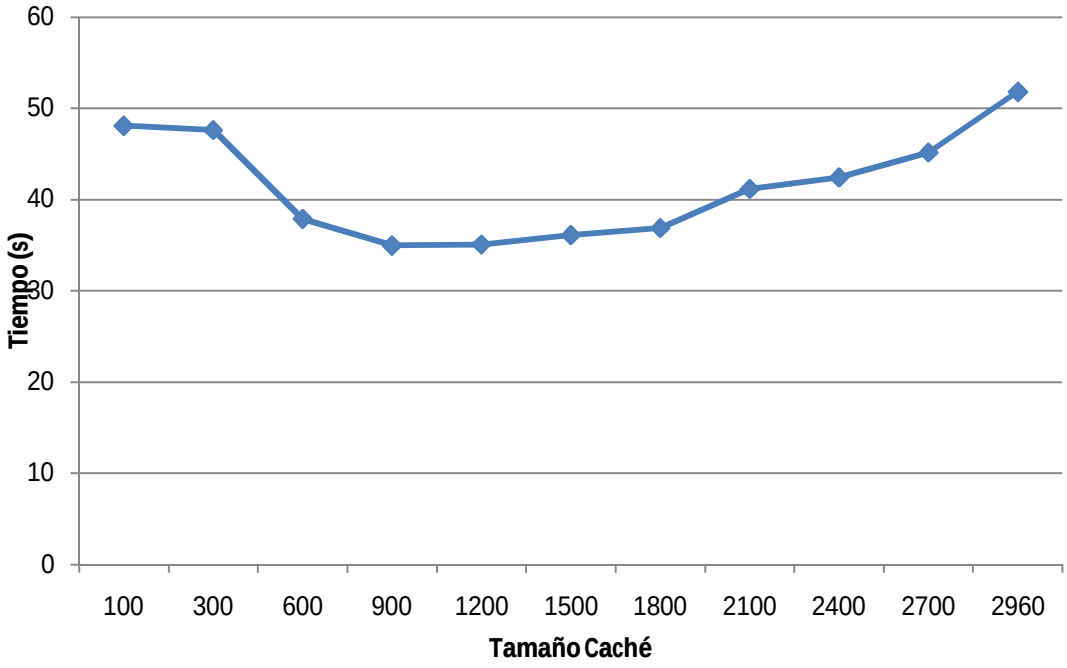

FIGURA 4.4 PRIOR-SVM Y TAMAÑO CACHÉ DE KERNEL

En la gráfica se observa como el tamaño de la caché para el que se obtiene un mejor tiempo se sitúa en torno a las 1000 muestras. Para tamaños inferiores, el tiempo de entrenamiento de la máquina es más elevado, debido a que la caché es demasiado pequeña y se requiere hacer muchos cálculos de valores que no se encuentran en ella.

Vemos como en el rango de 900 a 1800 no se produce un aumento sustancial del tiempo empleado. Es a partir de las 2000 muestras cuando el tiempo empieza a aumentar más notablemente. Esto se debe a que el tamaño de la caché es

demasiado grande para el número de vectores soporte que hay en los datos y por tanto, se pierde demasiado tiempo en calcular una matriz de kernel más grande de lo necesario.

Dependiendo de los datos que se utilicen, se obtendrá un tamaño óptimo de la caché diferente. El estudio de este tamaño óptimo, que se sale de los objetivos del presente proyecto, puede ser una buena línea de continuación del trabajo presentado.

4.3 Comparativa MATLAB y C

En este apartado se va a proceder a comparar el **tiempo de ejecución** del entrenamiento de la máquina Prior-SVM y la clasificación de muestras, entre una implementación en C y otra en MATLAB. Recordemos que conseguir que la implementación en C fuera más rápida que su equivalente en Matlab, es el principal objetivo del proyecto.

Debido a la aleatoriedad anteriormente citada, vamos a comparar los códigos entre MATLAB y C de dos formas distintas:

- **Comparación sin aleatoriedad**. En esta primera comparativa vamos a suprimir las partes aleatorias de ambos códigos. El entrenamiento y clasificación darán los mismos resultados pero el tiempo empleado será mucho mayor en ambos casos. El hecho de que ambos códigos hagan el mismo número de iteraciones nos dará una buena idea de si se ha conseguido el objetivo del proyecto.

- **Comparación promediada**. Aquí se van a ejecutar ambos códigos de forma normal, es decir, con la aleatoriedad propia del SMO. Para hacer la comparación más justa y realista se van a promediar cinco ejecuciones, de forma que el tiempo mostrado es la media de cinco ejecuciones realizadas en condiciones similares.

4.3.1 Comparación sin aleatoriedad

En la tabla mostrada a continuación se muestran, para cada conjunto de datos, las clases a clasificar, el número de datos usados en el entrenamiento y en la clasificación, el valor de C empleado y el tiempo de ejecución de todo el proceso tanto de la versión en MATLAB como de la versión en C. La última columna indica el porcentaje de tiempo que se ha mejorado de una implementación a otra.

Datos	Clases	Nº Atrib.	C	Nº Datos Train	Nº Datos *Test*	Tiempo Matlab (s)	Tiempo C (s)	%T. Diferencia
Vino	Cosecha 1 Cosechas 2,3	13	1	130	48	0,18	0,13	**-27,7**
Escritura	Escritor 1 Escritores 2-9	16	200	2500	1000	32,08	14,58	**-54,5**
Satélite	Asfalto No Asfalto	34	200	4435	2000	159,90	67,01	**-58,09**
Reconocimiento Carácter	Letra A Resto letras	16	60	5000	2300	251,63	101,71	**-59,57**
Telescopio	Partículas γ Otras	11	5	6000	2980	390,76	141,18	**-63,87**

TABLA 4.2 COMPARACIÓN SIN ALEATORIEDAD MATLAB-C PRIOR-SVM

Como se observa en la tabla, la implementación en C mejora sensiblemente los tiempos de su equivalente en MATLAB para todos los conjuntos de datos. Cuanto más grande es el conjunto de muestras con los que se ha entrenado, mayor es la diferencia. Así por ejemplo, para el conjunto de datos de *telescopio*, que contiene 6000 muestras de entrenamiento, la versión en C lo hace un 63% más rápido.

De los experimentos realizados, se ha observado como el tamaño de los datos de *test* no afecta prácticamente en el consumo de tiempo de todo el proceso. Lo mismo ocurre con la C empleada.

En el siguiente apartado veremos la misma tabla pero ya introduciendo la aleatoriedad propia del método SMO, que hará que ambas versiones se ejecuten mucho más rápido.

Los resultados mostrados en la anterior tabla han sido representados en la figura 4.5, donde se observa cómo cada vez es mayor la diferencia de tiempo de una implementación a otra.

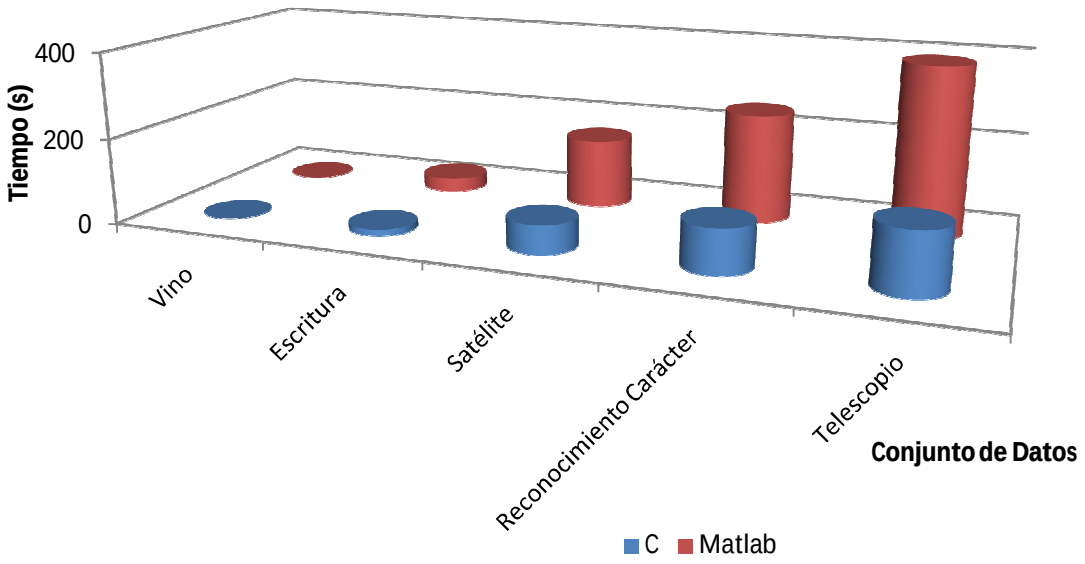

FIGURA 4.5 COMPARACIÓN SIN ALEATORIEDAD MATLAB-C PRIOR-SVM 1

En la siguiente gráfica se muestra el tiempo empleado por ambas implementaciones en resolver el problema para distintos tamaños de datos de entrenamiento pertenecientes a un mismo conjunto Los datos utilizados son los mismos que los empleados en el apartado 4.2. El número de datos de *test* empleados permanecerá fijo (1480x20) mientras los de entrenamiento irán aumentando progresivamente. De este modo podremos ver la diferencia o no de

tiempo de ejecución del entrenamiento y clasificación de ambas implementaciones en función del número de muestras a entrenar. Como se ha quitado la aleatoriedad, ambos programas harán exactamente las mismas iteraciones y operaciones, por lo que los resultados serán más justos.

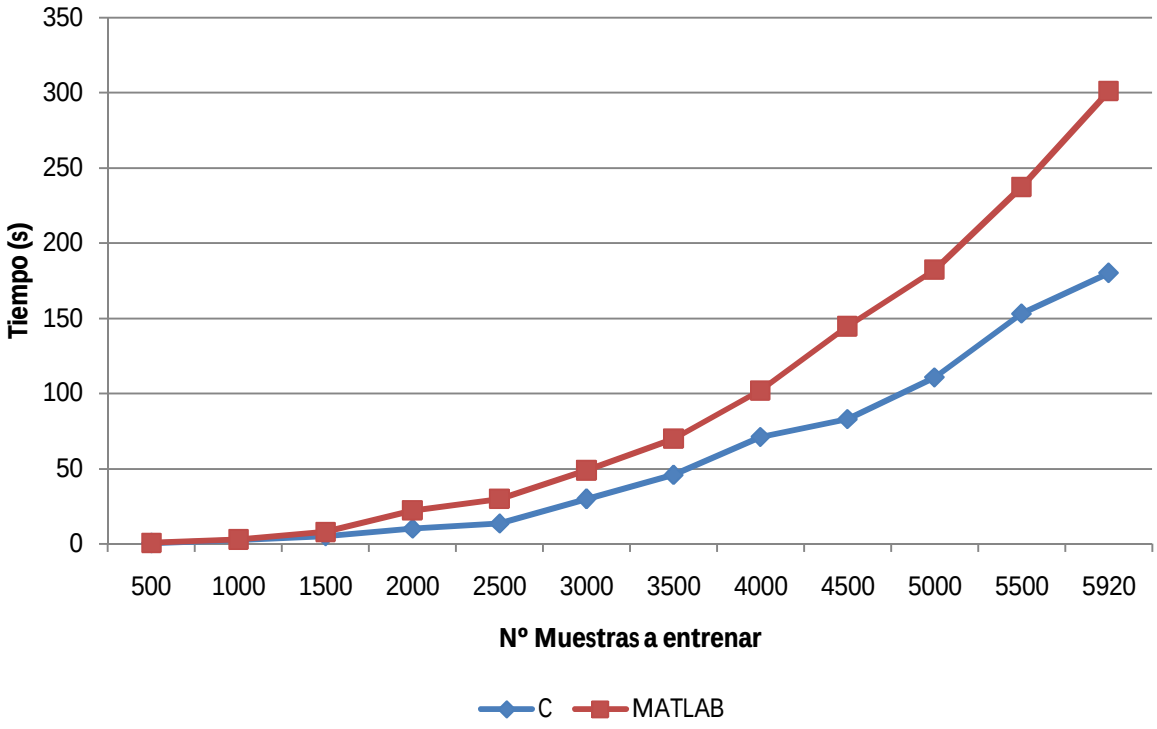

FIGURA 4.6 COMPARACIÓN SIN ALEATORIEDAD MATLAB-C PRIOR-SVM 2

Observamos cómo, una vez más, el tiempo empleado por el programa escrito en C mejora los tiempos de su versión en Matlab. Para pocas muestras, se obtienen resultados prácticamente similares, pero a medida que entrenamos la Prior-SVM con más datos, la diferencias de tiempos es cada vez mayor.

4.3.2 Comparación promediada

En este apartado se va a llevar a cabo el mismo experimento que en el anterior, pero con aleatoriedad y haciendo la media del tiempo invertida en cinco ejecuciones, para así poder obtener un resultado más justo.

Datos	Clases	Nº Atrib	C	Nº Datos Train	Nº Datos *Test*	Tiempo Matlab (s)	Tiempo C (s)	%T.Diferencia
Vino	Cosecha 1 Cosecha 2,3	13	1	130	48	0,20	0,14	**-30**
Escritura	Escritor 1 Escritores 2-9	16	200	2500	1000	13,01	7,58	**-41,7**
Satélite	Asfalto No Asfalto	34	200	4435	2000	28,19	12,20	**-56,7**
Reconocimiento carácter	Letra A Resto letras	16	60	5000	2300	54,76	21,78	**-60,2**
Telescopio	Partículas γ Otras	11	5	6000	2980	78,09	29,78	**-61,8**

TABLA 4.3 COMPARACIÓN PROMEDIADA MATLAB-C PRIOR-SVM

Como se observa en la tabla, una vez insertada la aleatoriedad propia de la SMO y tras haber promediado varias ejecuciones, la implementación en C sigue siendo más rápida que la de MATLAB. Además, vemos como los tiempos para ambos casos son más bajos que los obtenidos en la tabla 4.2. Los porcentajes de mejora, son parecidos a los obtenidos en el apartado anterior, llegando a una reducción del 61% para el conjunto *telescopio*.

Como ya hiciéramos en el apartado anterior, en la siguiente figura se representan los tiempos de cada grupo y se observa como a medida que los datos de entrenamiento son más numerosos, las diferencias son mayores.

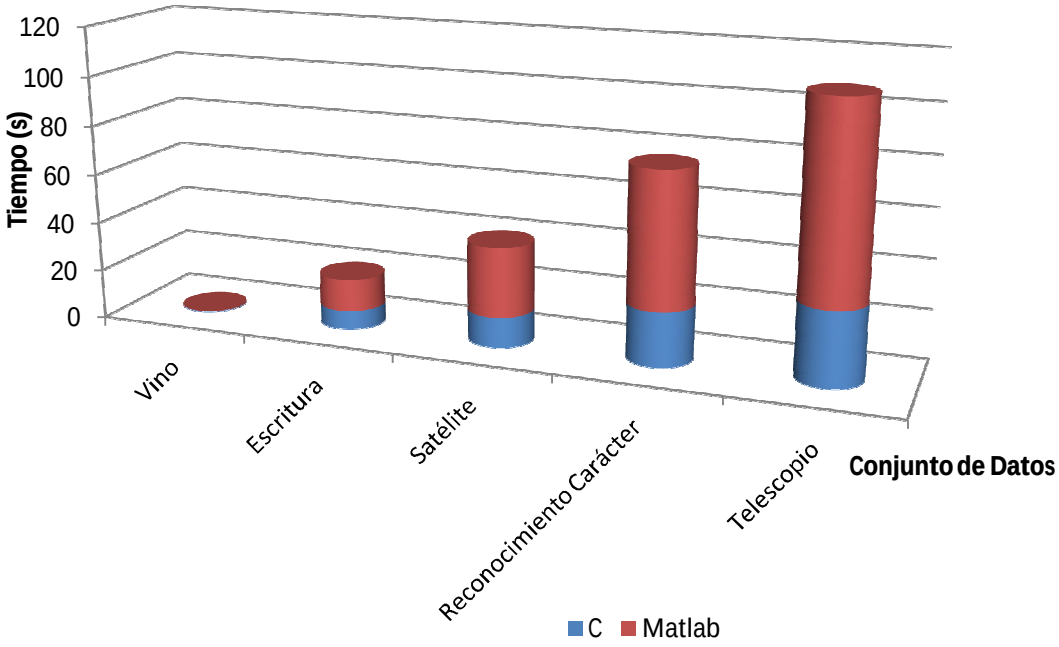

FIGURA 4.7 COMPARACIÓN PROMEDIADA MATLAB-C PRIOR-SVM 1

Si procedemos de igual manera que en la figura 4.6 pero insertando la aleatoriedad y promediamos cinco ejecuciones, obtenemos:

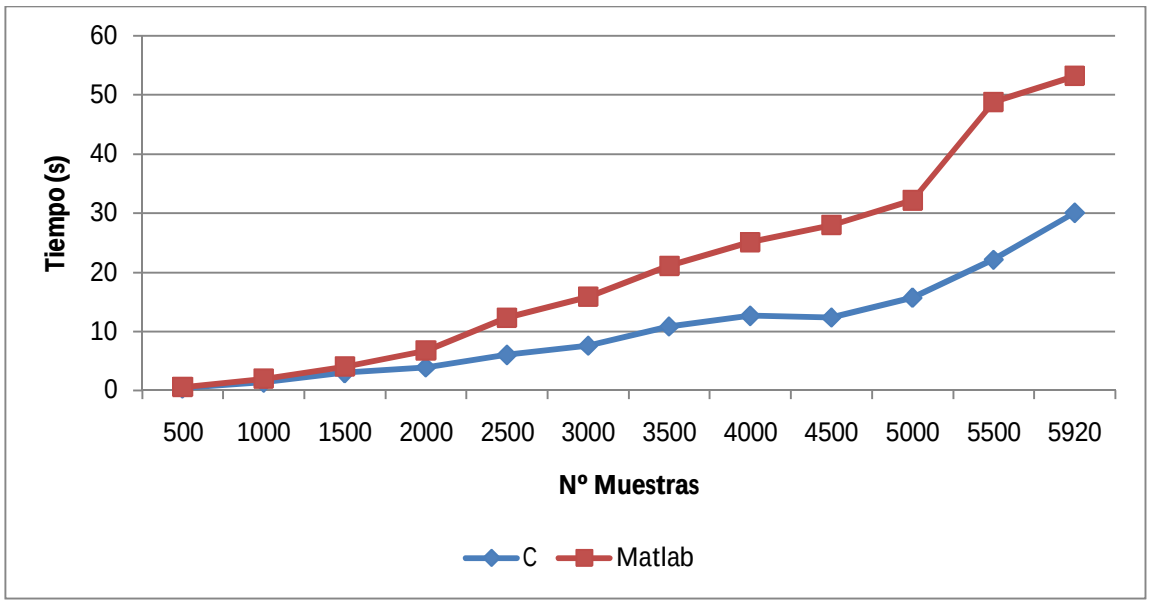

FIGURA 4.8 COMPARACIÓN PROMEDIADA MATLAB-C PRIOR-SVM 2

Observamos de nuevo como para pocas muestras no hay mucha diferencia en el tiempo de ejecución y como, a medida que entrenamos con más datos, la diferencia de tiempos es cada vez mayor.

4.4 Comparativa SVM y Prior-SVM

El objetivo de este apartado es comparar la SVM y la Prior-SVM en términos de muestras bien clasificadas. Aunque esto se sale de los objetivos principales del proyecto, se ha considerado interesante comparar dos máquinas de entrenamiento y clasificación de muestras entrenadas por el mismo método, SMO, pero donde una utiliza conocimiento *a priori* y otra no. Por último, se realizará una comparación del tiempo de ejecución de ambas muestras.

Para llevar a cabo las comparaciones se va a usar el código de la librería LIBSVM para entrenar la SVM y la *toolbox* desarrollada en el presente trabajo para entrenar la Prior-SVM.

En la tabla mostrada a continuación se muestra el porcentaje de datos bien clasificados de cada máquina para los diferentes conjuntos de datos. Además, se han incluido el número de Vectores Soporte obtenidos (en el caso del Prior-SVM se indican tanto los de la etapa *a priori* como *a posteriori*).

Datos	Clases	Nº Atributos	C	Nº Datos Train	Nº Datos *Test*	Nº VS SVM	Nº VS Prior-SVM	%Aciertos SVM	%Aciertos Prior-SVM
Vino	Cosecha 1 Cosecha 2 ó 3	13	1	130	48	76	41/16	**87,3**	**85,4**
Escritura	Escritor 1 Escritores 2-9	16	200	2500	998	171	113/39	**91,08**	**92,43**
Satélite	Asfalto No asfalto	34	200	4435	2000	302	168/75	**89,70**	**90,21**
Reconocimiento carácter	Letra A Resto letras	16	60	5000	2300	269	160/64	**98,53**	**97,98**
Telescopio	Partículas γ Otras	11	5	6000	2980	383	219/72	**97,65**	**98,03**

TABLA 4.4 COMPARACIÓN SVM Y PRIOR-SVM

Observamos como los resultados obtenidos son muy similares para ambas máquinas. Según vemos, los tres primeros conjuntos de datos son problemas más difíciles de clasificar, de ahí que ambas máquinas obtengan peores porcentajes de aciertos (en torno al 90%) que los dos últimos experimentos (98%). Así pues, no se puede extraer ninguna conclusión clara acerca de qué máquina ofrece mejores resultados, ya que dependerá del tipo y número de datos.

En la siguiente gráfica se va a comparar los resultados obtenidos por ambas máquinas dependiendo del número de muestras con las que se entrene. Se utilizarán los mismos datos que hemos empleado en la gráfica 4.8.

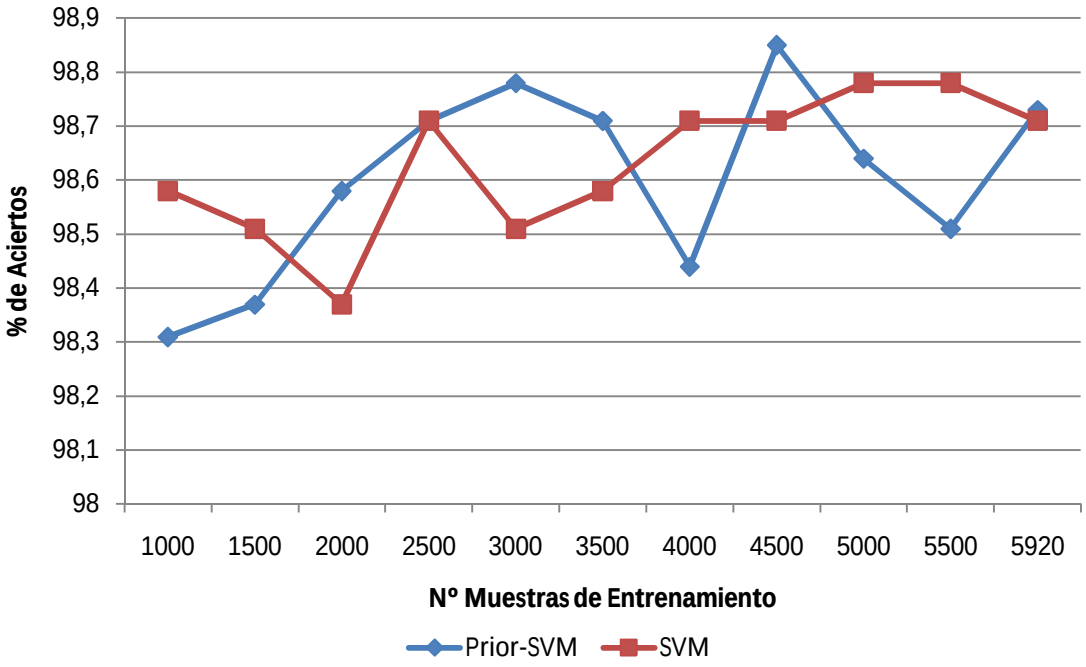

FIGURA 4.9 COMPARACIÓN SVM Y PRIOR-SVM 1

Observamos que, como se ha visto en la tabla anterior, ambas máquinas ofrecen resultados muy similares. Además, vemos cómo cuanto más numeroso es el conjunto de entrenamiento, mejor será el clasificador y mayor serán los porcentajes de muestras bien clasificadas. Los resultados han oscilado en torno al 98%, lo cual muestra el potencial del entrenamiento SMO.

Por último, en la gráfica 4.10 se comparan de nuevos ambas máquinas, pero esta vez en tiempo de ejecución. Se usarán los mismos datos que hemos empleado en la gráfica anterior.

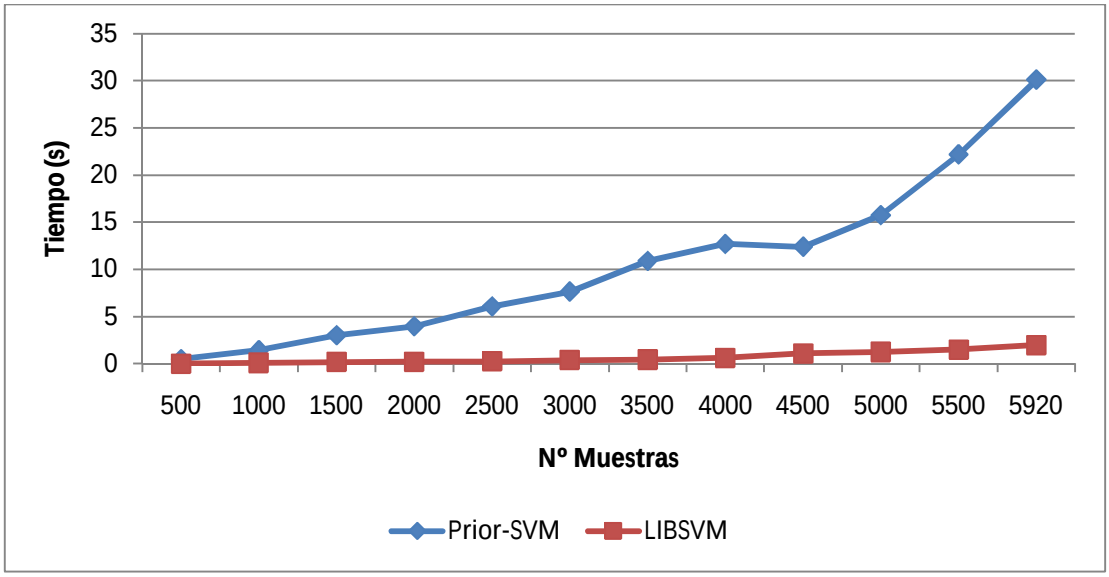

FIGURA 4.10 COMPARACIÓN SVM Y PRIOR-SVM 2

Observamos aquí la gran diferencia existente entre ambas máquinas hoy en día en términos de velocidad de ejecución. La LIBSVM es mucho más rápida, cosa que se hace más notable a medida que aumentan el número de muestras usadas en el entrenamiento.

Capítulo 5

CONCLUSIONES Y LÍNEAS FUTURAS

En este último capítulo se repasará el trabajo realizado, haciendo hincapié en los resultados obtenidos. Se extraerán las conclusiones del proyecto y se indicarán posibles líneas futuras para continuar el trabajo.

5.1 Conclusiones

El objetivo del presente proyecto era el de reprogramar un código en MATLAB de la forma más óptima posible, de modo que se produjese un sustancial aumento de la velocidad de ejecución sin alterar los resultados finales.

Este código entrenaba mediante el método de la SMO una variación de la SVM denominada Prior-SVM. La diferencia entre ambas máquinas radica, principalmente, en que la primera no usa conocimiento *a priori* y la segunda sí. El lenguaje elegido fue C, que por su condición de lenguaje de medio nivel, debía mejorar las prestaciones.

Además de reprogramar el código, se ha completado y mejorado, introduciendo elementos como una caché de kernel para hacerlo más eficiente. También se ha creado una completa *toolbox* en MATLAB con varias funciones que realizan la

compilación de funciones, carga de datos, la selección de parámetros, el entrenamiento, la clasificación y la muestra de resultados. La mayoría de las funciones han sido escritas en MATLAB, excepto las relacionadas con el entrenamiento de la máquina. Para integrar estas funciones escritas en C en la *toolbox*, se ha hecho uso de las funciones MEX, que permiten llamar desde el entorno MATLAB a funciones desarrolladas en otros lenguajes.

Además de esta *toolbox*, se han creado también otras dos más: una que integra el código del entrenamiento realizado en MATLAB y otra para la librería LIBSVM. De este modo, se consigue una comparación más eficiente.

Una vez programadas las distintas funciones se han realizado varios experimentos para comprobar el rendimiento de la *toolbox*. En el primero de ellos, se ha estudiado el tamaño ideal que debe tener la caché de kernel, llegando a la conclusión de que dicho tamaño dependerá de cada caso, esto es, datos de entrada, valor de C etc.

Tras esto, se ha pasado a comparar la velocidad de ejecución del entrenamiento programado en C con respecto al programado en MATLAB. Para ello se ha dividido el experimento en dos partes. En la primera se ha quitado la parte aleatoria, que introduce el método SMO, para que ambos implementaciones hiciesen exactamente las mismas iteraciones. De esta forma se obtendrían unos resultados más justos. En la segunda parte, se ha restablecido la aleatoriedad y se han promediado cinco ejecuciones. En ambos casos los resultados han sido similares: **el código en C mejora sustancialmente la velocidad de ejecución** con respecto a la versión en Matlab.

Para pocas muestras de entrenamiento se ha visto como el tiempo invertido por ambos programas era prácticamente el mismo, aunque ligeramente favorable a la versión en C. A medida que se entrena la máquina con más muestras, la diferencia de tiempos era más notable, llegándose a reducir el tiempo más de un 60% para un conjunto de 6000 muestras.

Por último, y saliéndonos ligeramente de los objetivos del proyecto, se ha comparado la SVM y la Prior-SVM. Es decir, una máquina que no hace uso de conocimiento *a priori*, y otra que sí lo usa a la hora de clasificar.

Los experimentos reflejan que ambas ofrecen un porcentaje de aciertos muy elevado incluso con pocas muestras de entrenamiento. Se han comparado ambas máquinas para distintos conjuntos, con números de muestras y atributos variables, sin llegar a una conclusión clara sobre cual clasificador es mejor. Así pues, dependiendo de los datos (rango de valores, número de atributos, C usada, número de datos de entrenamiento, etc), una u otra máquina ofrecerá mejores resultados. Habría que realizar más experimentos y analizarlos cuidadosamente para identificar en qué casos es más recomendable el uso de una sobre la otra, lo cual se sale de los objetivos de este trabajo. También se comparó el tiempo de ejecución de ambas máquinas, y, como se esperaba, se observó que LIBSVM presentaba tiempo mucho mejores.

Así pues, podemos afirmar que el programa realizado mejora el tiempo de ejecución de su versión en MATLAB, manteniendo los elevados porcentajes de muestras bien clasificadas, que era el objetivo del presente proyecto.

5.2 Líneas Futuras

Utilizando el presente proyecto como punto de partida, hay varias líneas por las que se puede continuar el trabajo realizado. Es posible introducir variaciones en el diseño y en la implementación para tratar de sacarle más rendimiento al programa y/o reducir el tiempo de ejecución. Algunas opciones serían:

- Adaptar el código para que soporte problemas multiclase. Ya sea por el método de *uno contra todos* o *todos contra todos*.
- Reprogramar el código en otro lenguaje de medio/bajo nivel para tratar de mejorar la velocidad del entrenamiento.

- Modificar el código para que se utilice el tamaño de la caché de kernel que proporcione una ejecución más eficiente para cada conjunto de datos.
- Modificar el código para que se pueda trabajar con otros kernel, como el lineal o el polinómico. Para esto y para el tamaño de la caché se podría utilizar validación cruzada para hallar el caso óptimo.
- Identificar los casos en los que es más recomendable usar la máquina SVM frente a la Prior-SVM y viceversa.
- Utilizar más de un clasificador *a priori* y estudiar si se consiguen mejores resultados.

BIBLIOGRAFÍA

[1] Tom M. Mitchell. *Machine Learning*. McGraw-Hill International Edition. 1997.

[2] Miguel González Mendoza. *Aprendizaje Estadístico, Redes Neuronales y Support Vector Machines: Un enfoque global.* ITESM CEM: Intelligent Transportation Systems Research Group. Universidad Veracruzana, Xalapa, México. 2005.

[3] Ginny Mak. *The implementation of support vector machines using the sequential minimal optimization algorithm*. Master's Project. School of Computer Science McGill University, Montreal, Canada. 2000.

[4] Juan Sebastián Romero Fernández. *Clasificación de Terrenos en Imágenes Multiespectrales Mediantes Máquinas de Vectores Soporte*. *Los problemas de la clasificación, 29.* Proyecto Fin de Carrera. Dpto. Teoría de la Señal. Universidad Carlos III de Madrid. 2004.

[5] Jesús Cáceres Tello. *Reconocimiento de patrones y el aprendizaje no supervisado*. Dpto. Ciencias de la Computación. Universidad de Alcalá. 2007.

[6] Cecilio Angulo Bahón. *Aprendizaje con máquinas núcleo en entornos de multiclasificación*. Tésis. Dpto. Ciència dels ordinadors. 2001.

[7] David Tomás, José. L. Vicedo, Armando Suárez (UA), Empar Bisbal y Lidia Moreno (UPV). *Una aproximación multilingüe a la clasificación de preguntas basada en aprendizaje automático*. 2005.

[8] V. Vapnik y A. Lerner. *Pattern recognition using generalized portraid method. Automation and remote control*, 24. 1963.

[9] C. Burges. *A tutorial on support vector machines for pattern recognition*. Bell Laboratories, Lucent Technologies, USA. Data Mining and Knowledge Discovery, 2, 121–167. 1998.

[10] Raúl Sánchez Santofimia. *Reconocimiento en Imágenes de Baja Resolución Mediante Clasificadores SVM (Máquinas de Vectores Soporte). La Máquina de Vectores Soporte, 17.* Proyecto Fin de Carrera. Dpto. Teoría de la Señal. Universidad Carlos III de Madrid. 2005.

[11] Kristin P.Bennett, Erin J. Bredensteiner. *Duality and Geometry in SVM Classifiers*. 17th International Conf. on Machine Learning. 2000.

[12] N. Cristianini y J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Capítulo 6 SVM, 93. 2002.

[13] Gustavo Camps i Valls. *Redes neuronales y Máquinas de Vectores Soporte para la predicción y modelización de la concentración valle de Ciclosporina (CyA) en pacientes con trasplante renal*. Phd Thesis, Dto. Ingeniería Electrónica. Universidad de Valencia. 2002.

[14] T.M. Cover. *Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. IEEE Transaction on Electronics Computer*. Vol. 14, 326-334. 1965

[15] Noelia López Martín. *Reconocimiento de Imágenes de Baja Resolución Mediante Clasificadores SVM. Clasificación no lineal*, 13. 2008.

[16] José Luis Alba Castro. Curso de doctorado 2003-2005: *Decisión, estimación y clasificación. Departamento de Teoría de la Señal y Comunicaciones*. Universidad de Vigo.

[17] R. Fletcher. *Practical Methods of Optimization*. 2000.

[18] John C. Platt. *Fast Training of Support Vector Machines using Sequential Minimal Optimization*. Capítulo 12, 41. 1999.

[19] http://svmlight.joachims.org/

[20] Osuna, Freund, R., Girosi, F. *Improved Training Algorithm for Support Vector Machines*. IEEE NNSP´97, Amelia Island. 1997.

[21] *Data Dependent Priors in PAC-Bayes Bounds*. John Shawe-Taylor, Emilio Parrado-Hernandez and Amiran Ambroladze. 19th International Conference on Computational Statistics Paris - France, August 22-27, 2010.

[22] Schölkopf, B., Platt, J. y Hoffman, T. *Tighter PAC- Bayes bounds. Advances in Neural Information Processing Systems 19*.

[23] http://www.mathworks.com/

[24] Javier García de Jalón, José Ignacio Rodríguez, Jesús Vidal. *Aprenda MATLAB 7.0 como si estuviera en Primero*. 2005.

[25] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM : a library for support vector machines*. 2001. Software en http://www.csie.ntu.edu.tw/~cjlin/libsvm

[26] http://archive.ics.uci.edu/ml/index.html/