Práctica 11

SVM

Máquinas de Vectores Soporte

Dedicaremos esta práctica a estudiar el funcionamiento de las, tan de moda, máquinas de vectores soporte (SVM).

1 Las máquinas de vectores soporte

Las SVM han sido implementadas en diversos lenguajes de programación y con diferentes algoritmos de optimización. Lo primero es una simple anécdota mientras que lo segundo hace que ante algunos conjuntos de datos los resultados ofrecidos por diferentes implementaciones de SVM no ofrezcan los mismos resultados.
Nosotros veremos dos implementaciones: una más didáctica y otra más orientada a la experimentación. Ambas se utilizarán desde MATLAB.

2 Implementación más visual

Podemos descargar esta implementación de:

http://www.aic.uniovi.es/SSII/P11/**svm**.zip

Una vez descargada la descomprimimos y situamos el *Current Directory* de MATLAB en dicho directorio.

2.1 Clasificación

Ahora podemos escribir `uiclass` en la ventana de comandos con lo que se visualiza el entorno gráfico de la aplicación:

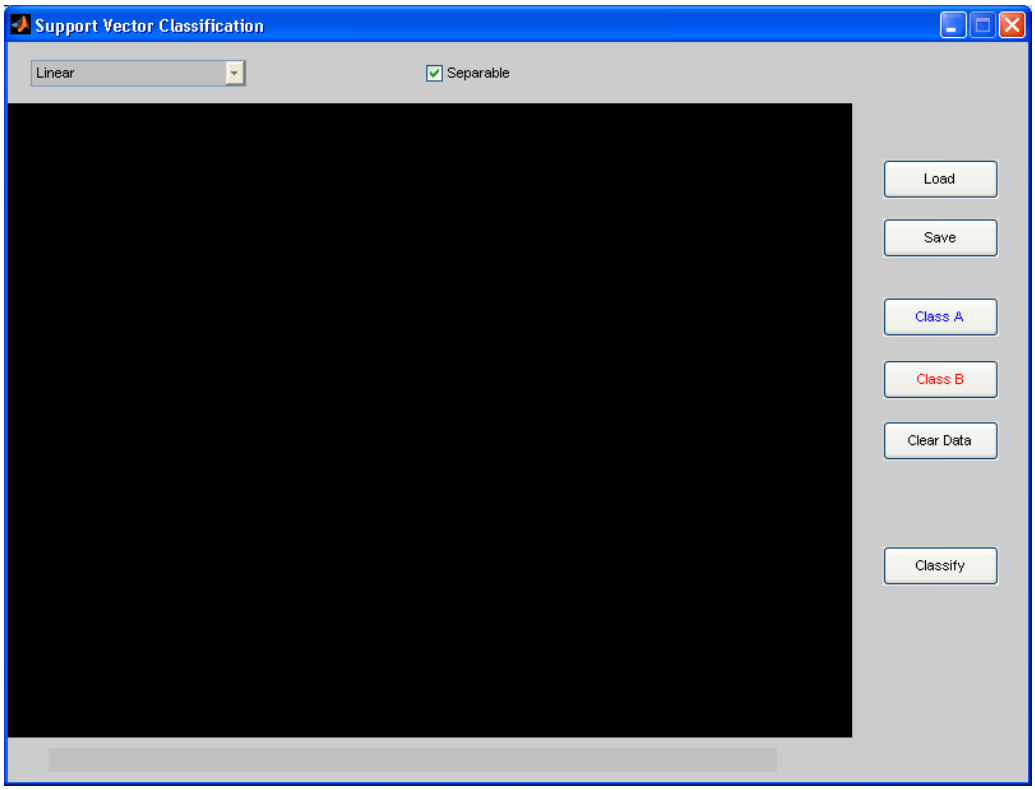

Figura 1.- Entorno gráfico (clasificación)

2.1.1 Separable en el espacio de entrada

Carguemos el problema `Examples\Classification\linsep.mat` que contiene los siguientes ejemplos:

X_1	X_2	Y
1	1	-1
3	3	1
1	3	1
3	1	-1
2	2.5	1
3	2.5	-1
4	3	-1

y pulsemos sobre *Classify*. Acabamos de entrenar el sistema y aparece representado el hiperplano que separa la clase positiva de la negativa:

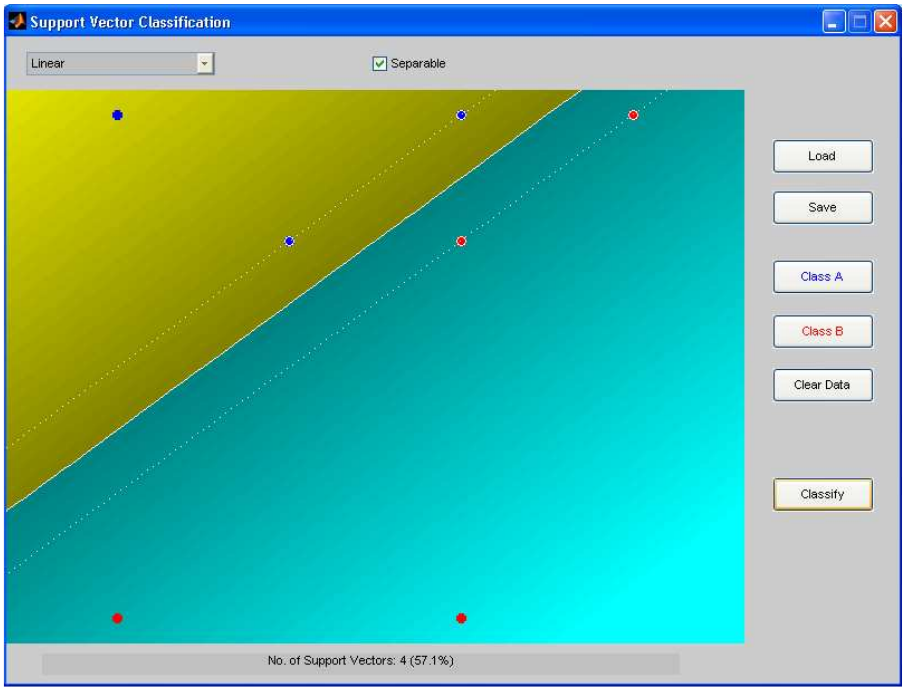

Figura 2.- Separable linealmente

Vemos que separa correctamente los ejemplos positivos de los negativos. Podemos ir al *Workspace* de MATLAB y ver los valores almacenados en *alpha*. Los ejemplos correctamente clasificados y fuera del margen tienen una alpha 0 (o casi cero).

2.1.2 No separable en el espacio de entrada

Si cargamos ahora el conjunto `nlinsep.mat`:

X_1	X_2	Y
1	1	-1
3	3	1
1	3	1
3	1	-1

2	2.5	1
3	2.5	-1
4	3	-1
1.5	1.5	1
1	2	-1

y entrenamos, vemos que el hiperplano no separa bien los ejemplos positivos de los negativos. Si desactivamos la casilla *Separable* y modificamos el valor de la C (en *Bound*) vemos que cuanto mayor sea el valor de la C mejor es la solución (prueba primero con el valor 1 y después con el valor 10). Pero en ningún caso podrá clasificar todos los ejemplos correctamente, ya que este conjunto no es separable en el espacio de entrada.

¿Pero será linealmente separable en otro espacio? Esto es lo que intentamos con las funciones *kernel*. En el *combobox*, en lugar de seleccionar *Linear*, selecciona *Polynomial* con grado 2 y entrena:

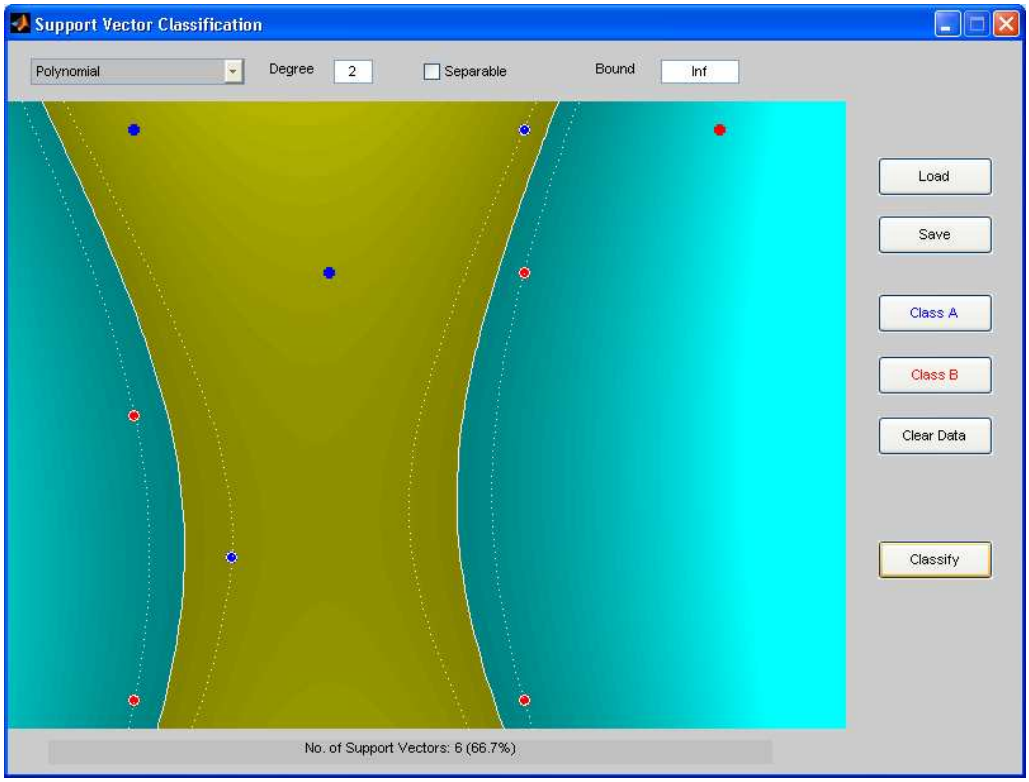

Figura 3.- No es separable linealmente en el espacio de entrada pero si en el espacio generado con un polinomio de grado 2

Se ha separado correctamente. Observa las alphas de cada ejemplo.

2.1.3 Un conjunto más complicado

Vemos lo que sucede con un conjunto de datos un poco más grande. Para ello tomaremos los lirios (ya conocidos de prácticas anteriores). Si representamos los lirios fijándonos únicamente en sus pétalos vemos lo siguiente:

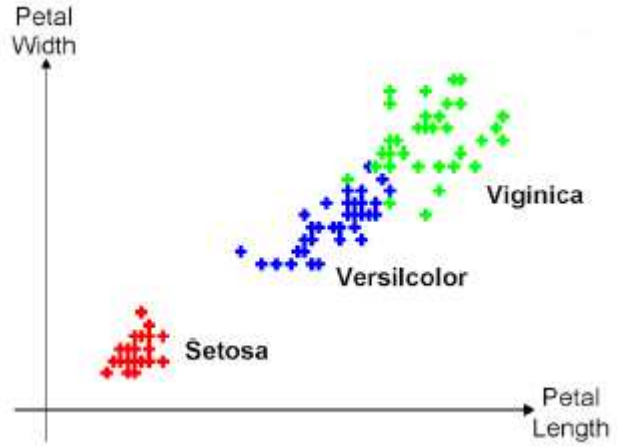

Figura 4.- Representación del conjunto de datos de los lirios

Las SVM se utilizan en problemas en los que hay dos clases (positiva y negativa). Por tanto necesitamos un conjunto de datos para aprender cada clase. En el directorio de los ejemplos hay tres ficheros de los lirios:

1. `iris1v23.mat`
   - Clase positiva: Versicolor
   - Clase negativa: Virgínica y Setosa
2. `iris2v13.mat`
   - Clase positiva: Setosa
   - Clase negativa: Virgínica y Versicolor
3. `iris3v12.mat`
   - Clase positiva: Virgínica
   - Clase negativa: Setosa y Versicolor

[Trabajo] Utiliza los kernels lineal, polinomial y RBF para intentar separar las clases. Modifica los parámetros. ¿Son separables las clases?

2.2 Regresión

Para resolver problemas de regresión debemos ejecutar `uiregress`[1].

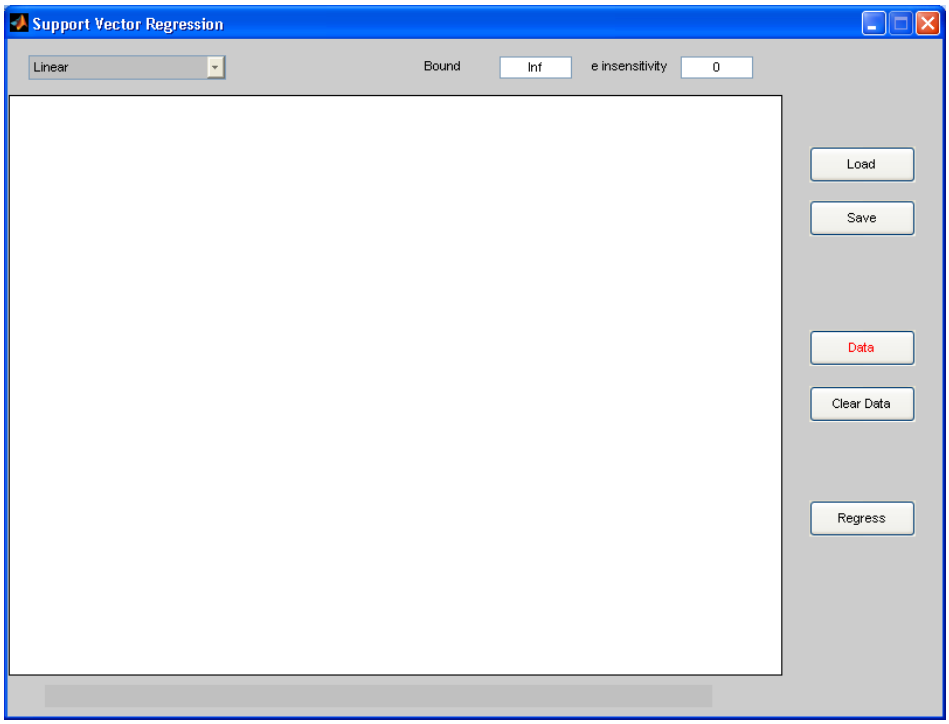

Figura 5.- Entorno gráfico (regresión)

Podemos cargar algún ejemplo del directorio de ejemplos y probarlo.

3 Implementación más experimental

Para la experimentación utilizaremos la implementación de SVM que vienen integrada en el SPIDER y otra un poco más robusta.

3.1 SVM del SPIDER

Primero veremos como funciona el SVM que viene integrado en el SPIDER. Esta implementación es menos robusta que la que veremos en el siguiente epígrafe.

Vamos primero a generar un conjunto pequeño. La clase positiva estará formada por 20 ejemplos centrados en (-1,-1) y con una desviación estándar de 1. La clase negativa tendrá las mismas características pero estará centrada en (1,1).

```
randn('seed',1)
m=20;
d=1;
s=1;
x1=[randn(m,1)*s-d randn(m,1)*s-d]; %(-d,-d) +
x2=[randn(m,1)*s+d randn(m,1)*s+d]; %(+d,+d) -
de=data([x1;x2],[ones(m,1);-ones(m,1)]);
```

Podemos resolverlo con un kernel lineal y ver su hiperplano:

[1] Si tienes el SPIDER en el path, esta implementación de SVM no te funcionará, ya que algunas funciones de ambas implementaciones tienen el mismo nombre y, por tanto, entran en conflicto.

```
a=svm;
a.C=Inf;
a.child=kernel('linear');

[res sis] = train(a,de);
loss(res)
plot(sis);
```

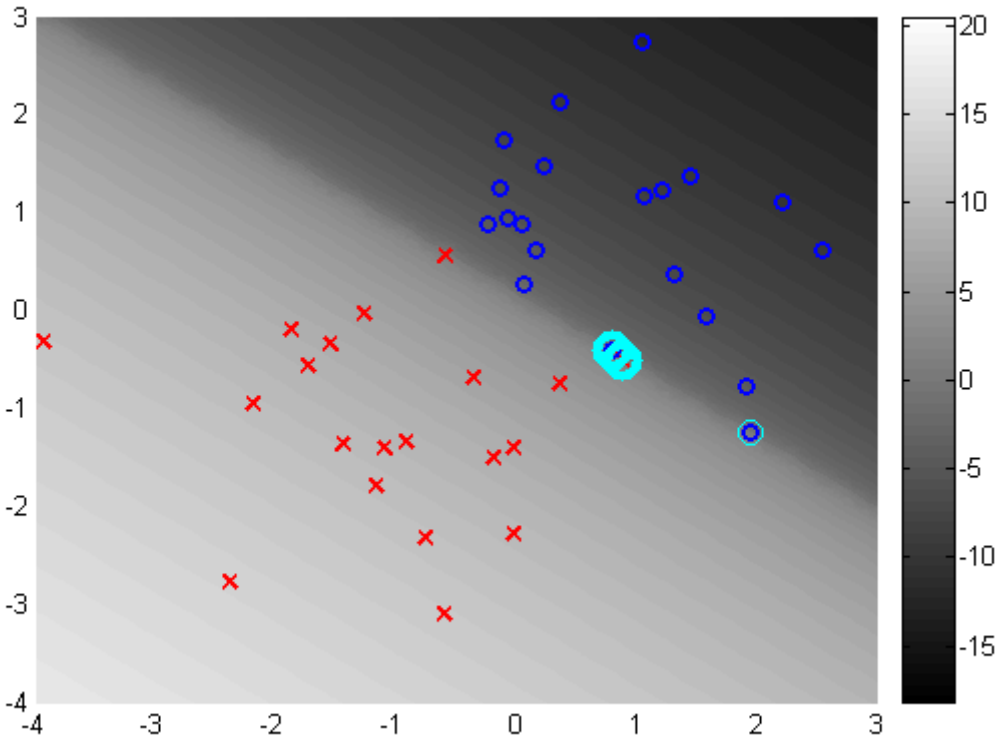

Figura 6.- Resultado con el kernel lineal

En la imagen se ven con círculos los ejemplos negativos y con equis los positivos. La tonalidad del fondo representa lo alejado que se está del hiperplano separador (en uno u otro sentido). Cuanto más claro, más positivo y cuanto más oscuro, más negativo. Los ejemplos que aparecen rodeados son los vectores de soporte.
Podemos, ahora, evaluar lo aprendido sobre otro conjunto (conjunto de test):

```
x1t=[randn(m,1)*s-d randn(m,1)*s-d]; %(-d,-d) +
x2t=[randn(m,1)*s+d randn(m,1)*s+d]; %(+d,+d) -
dt=data([x1t;x2t],[ones(m,1);-ones(m,1)]);
tst=test(sis,dt);
loss(tst)
```

¿Falla mucho en ejemplo no vistos en el entrenamiento?
Podemos utilizar otros kernels:

```
grado=3;
a.child=kernel('poly',grado);
```

para un kernel polinómico de grado 3 o

```
sigma=1;
a.child=kernel('rbf',sigma);
```

para un kernel RBF con sigma 1.

[**Trabajo**] Prueba con diferentes kernels y parámetros de kernel para ver cuál genera mejores resultados sobre el conjunto de test. Puedes también modificar la C: `a.C=1`

3.2 LibSVM

El `libSVM` es una implementación de las `SVM` muy rápida y robusta. Además, esta implementación es capaz de trabajar con más de dos clases. En ese caso las clases no serán +1 y -1, sino que serán números enteros consecutivos del 1 en adelante. Es decir, si tenemos 4 clases, las clases serán 1, 2, 3 y 4.

Hemos desarrollado un objeto `SPIDER` para que podáis ejecutarlo en este entorno. El objeto podéis obtenerlo en:

http://www.aic.uniovi.es/SSII/P11/libsvm.zip

Descomprimimos el fichero en un directorio e incluimos ese directorio en `path` del `Matlab`, de manera que los dos directorios contenidos en el `zip` queden incluidos en el `path`. Uno de los directorios contiene el código con los ficheros `mexw32` del `libsvm` y el otro contiene la definición de la clase `mylibsvm`. Si creo el directiorio Z:\spider_jul2406\spider\mylibsvm y en el mismo coloco los directorios @mylibsvm y libsvm-mat-2.82-2, tendría que añadir al path lo siguiente:

>> addpath('Z:\spider_jul2406\spider\mylibsvm\libsvm-mat-2.82-2')
>> addpath('Z:\spider_jul2406\spider\mylibsvm')
Una vez hecho esto podemos escribir:

```
>> help mylibsvm
```

Y obtendremos una ayuda detallada de los parámetros que admite el algoritmo. Aquí comentamos los más relevantes:

- `-s`. Selecciona el tipo de SVM. 0 para SVM de clasificación y 3 para SVM de regresión.
- `-t`. Para seleccionar el kernel. 0 para lineal, 1 para polinomial, 2 para RBF y 3 para sigmoidal
- `-d`. Para indicar el grado del polinomio con kernel polinomial
- `-g`. Para indicar la gamma
- `-c`. Para indicar cuánto queremos ajustarnos al conjunto de entrenamiento. Es la C del problema de optimización

Si tenemos creados los conjuntos `de` y `dt` del epígrafe anterior, podemos utilizarlos de la siguiente manera:

```
>> a=mylibsvm('c=10;t=2;g=0.15');
```

Acabamos de crear un objeto mylibsvm con kernel RBF, gamma 0.15 y c 10^2. Para entrenarlo lo hacemos de la misma manera que en el epígrafe anterior, ya que estamos utilizando un objeto que hemos definido para el SPIDER.

```
>> [res sis] = train(a,de);
training mylibsvm....
param_string->  -c 10.000000 -t 2 -g 0.150000
Accuracy = 97.5% (39/40) (classification)
Mean squared error = 0.1 (regression)
Squared correlation coefficient = 0.904762 (regression)
```

Una vez entrenado vemos cuanto falla en reescritua:

```
>> loss(res)
data -> mylibsvm -> class_loss=0.025
data dimensions:
X = 0x0  Y = 1x1
```

Podemos ver ahora cómo se comporta con el conjunto de test

```
>> tst=test(sis,dt);
Accuracy = 97.5% (39/40) (classification)
Mean squared error = 0.1 (regression)
Squared correlation coefficient = 0.904762 (regression)
>>
>> loss(tst)
data -> mylibsvm -> class_loss=0.025
data dimensions:
X = 0x0  Y = 1x1
```

[NOTA] Os recomendamos que utilicéis esta implementación del SVM para las pruebas que tengáis que hacer, ya que nuestra humilde experiencia nos indica que se trata de un algoritmo más robusto y fiable.

3.3 Trabajo

Conéctate al sitio:

http://www.ics.uci.edu/~mlearn/databases/

Aquí ya hemos estado otras veces. Podemos descargarnos los problemas que queramos para experimentar con ellos. Como vamos a experimentar con el SVM sólo nos interesan problemas que tengan todos sus atributos numéricos.

Selecciona uno de ellos y descárgalo. Modifícalo convenientemente para poder crear un conjunto de datos para el SPIDER. Utiliza diversas configuraciones del SVM para aprender. ¿Cuál de esas configuraciones produce mejores resultados en validación cruzada?

[2] Podemos pasarle los parámetros que queramos siempre que los separemos con ';'