Master Universitario en Ingeniería Biomédica

Master in Biomedical Enginering

OWL METABOLOMICS

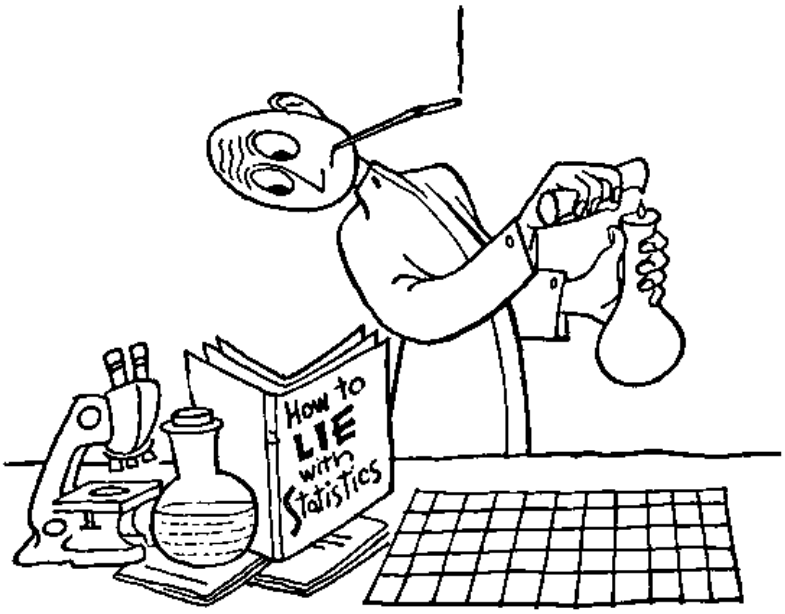
Máquinas de Vector Soporte con R

imartinez(at)owlmetabolomics.com

github: @imarranz

calonso(at)owlmetabolomics.com

10 de Enero de 2019



The secret language of statistics, so appealing in a fact-minded culture, is employed to sensationalize, inflate, confuse, and oversimplify. Statistical methods and statistical terms are necessary in reporting the mass data of social and economic trends, business conditions, opinion polls, the census. But without writers who use the words with honesty and understanding and readers who know what they mean, the result can only be semantic nonsense.

How to lie with statistics, 1954

Índice

Índice de figuras

Índice de tablas

1. Máquinas de Vector Soporte (SVM, Support Vector Machine)

En el contexto de *Machine Learning*, las Máquinas de Vector Soporte son modelos de aprendizaje supervisado asociados a los algoritmos que analizan datos para su análisis de clasificación y/o regresión.

Dado un conjunto de muestras de entrenamiento, cada una de ellas clasificada en una categoría, el algoritmo de entrenamiento de una SVM construye un modelo que asigna una clase a cada observación. Un modelo SVM es una representación de estas muestras en el espacio de tal manera que las muestras de cada categoría están separadas de forma clara.

1.1. Historia

El algoritmo original de las Máquinas de Vector Soporte fue escrito por *Vladimir N. Vapnik* y *Alexey Ya. Chervonenkis* en 1963. En 1992, *Bernhard E. Boser*, *Isabelle M. Guyon* y *Vladimir N. Vapnik* sugirieron una metodología para crear clasificadores no lineales aplicando el mismo concepto de hiperplanos de margen máximo.

1.2. Motivación

La clasificación de muestras es una tarea común en *Machine learning* en la que dados unos datos en los que cada observación pertenece a alguna clase y la finalidad es decidir a qué clase asignar una nueva observación. En el caso de las SVM, cada observación se considera un vector de *p* dimensiones (tenemos *p* variables) y tratamos de separar cada clase. Si lo hacemos con hiperplano de dimensiones *p-1* estaremos aplicando un clasificador lineal. Hay muchos hiperplanos que podrían clasificar nuestros datos. Podemos razonar y buscar aquel

hiperplano que muestra la mayor separación entre clases. Elegimos entonces aquel hiperplano cuya distancia a los puntos más cercanos de cada lado sea máxima. Si tal hiperplano existe se denomina *hiperplano de máximo margen* y el clasificador lineal asociado se define como *clasificador de máximo margen*.

1.3. Definición

Podemos definir más formalmente este hiperplano. Una Máquina de Vector Soporte construye un hiperplano o conjunto de hiperplanos *n*-dimensionales que pueden ser usado para clasificación, regresión u otras tareas. Geométricamente, una buena separación se alcanzará por aquel hiperplano que tenga la mayor distancia entre las observaciones de cada clase en las muestras de entrenamiento.

1.3.1. kernel

A veces, el problema original puede ser resuelto en un espacio de dimensión finita, pero en otras ocasiones sucede que los conjuntos a discriminar no tienen una separación lineal en ese espacio. Para solventar este inconveniente, el espacio de dimensión finita donde está planteado el problema puede ser transformado a un espacio de dimensión mayor, donde es esperable que la separación entre clases se más fácil de calcular.

El aumentar la dimensión del espacio en el que estamos trabajando implica un coste computacional mayor. Para que este aumento sea razonable las transformaciones a espacios de dimensión mayor se diseñan de tal manera que los productos escalares en estos nuevos espacios puedan ser calculados fácilmente en términos de las variables iniciales. Para ello se utilizan las funciones *kernel* $k(x, y)$ seleccionadas específicamente para resolver este problema. Los hiperplanos en una mayor dimensión son definidos como aquellos conjuntos de puntos tales que su producto escalar con un vector en ese espacio es constante. Los vectores que definen los hiperplanos pueden ser elegidos como una combinación lineal con parámetros α_i de imágenes de los vectores de características x_i. Si

elegimos el hiperplano con estas propiedades, los puntos x en el espacio de características son llevados hiperplanos que se define por la siguiente relación:

$$\sum_i \alpha_i \cdot k(x_i, x) = \text{constante} \tag{1.1}$$

Tenemos que tener en cuenta que si $k(x, y)$ se vuelve pequeño a medida que y crece más lejos de x, cada término de la suma mide el grado de cercanía de la prueba Punto x al punto de base de datos correspondiente x_i. De esta manera, la suma de los núcleos anteriores puede usarse para medir la proximidad relativa de cada punto de prueba a los puntos de datos originados en uno u otro de los conjuntos a discriminar.

1.3.1.1. Construcción de un *kernel*

Como hemos comentado, la función *kernel* nos lleva el espacio de características (donde están nuestros datos) a un nuevo espacio de dimensión mayor de tal manera que sea fácil calcular el producto escalar

Sea el *kernel* definido como

$$\Phi(x_1, x_2) = \Phi(x_1^2, x_2^2, \sqrt{2x_1x_2}) = (z_1, z_2, z_3) \tag{1.2}$$

que lleva un punto $x \in R^2$ a $z \in R^3$. En la figura 1.1 observamos que este kernel separa el interior de la circunferencia con el exterior con un hiperplano en R^3, siendo mucho más fácil separar las clases en las que están divididas las observaciones.

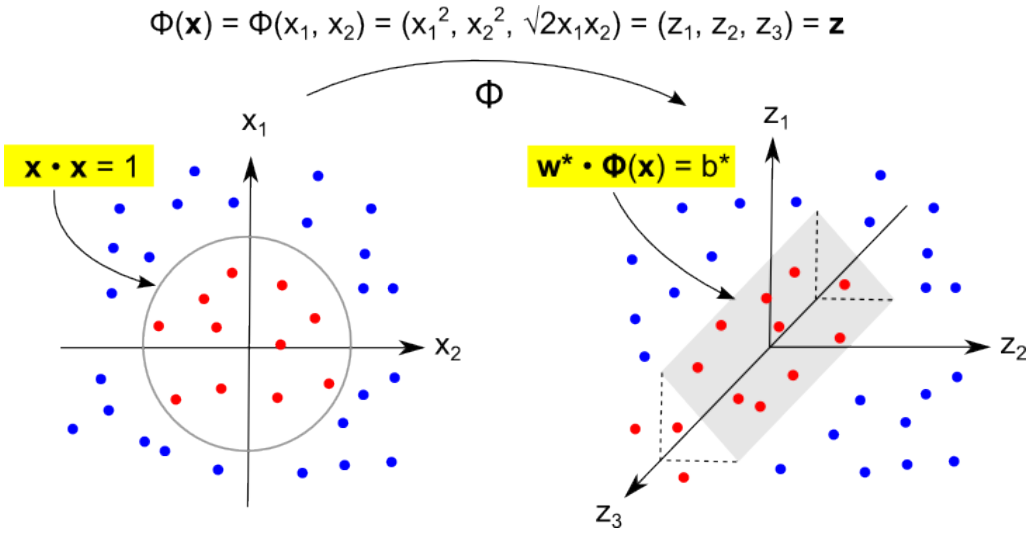


Figura 1.1: *El kernel así definido separa el interior de la circunferencia con el exterior con un hiperplano en R^3, siendo mucho más fácil separar las clases en las que están divididas las observaciones.*

Las Máquinas de Vector Soporte funcionan agrupando los puntos de las características según sus clases. En la figura 1.2 se generan dos vectores de características bidimensionales $x = \{x_1, x_2\}$ de tal manera que la clase $y = -1$ puntos (triángulos) están bien separados de la clase $y = 1$ (círculos).

El algoritmo encuentra el mayor margen lineal posible que separa estas dos regiones. Los separadores se apoyan sobre los puntos avanzados que están justo en la línea frente a sus respectivas regiones. Estos puntos, marcados como dos triángulos en negrita y un círculo en negrita en la figura 1.2, se llaman los *vectores de apoyo* o *vectores soporte*, ya que están apoyando las líneas de separación. De hecho, la tarea de aprendizaje del algoritmo de Máquinas de Vector Soporte consiste en determinar estos puntos vector de soporte y la distancia de margen que separa las regiones. Después del entrenamiento, todos los demás puntos de no apoyo no se usará para futuras predicciones.

En el espacio de características lineales, los vectores soporte se suman a un vector de hipótesis general *h*,

$$h = \sum_i c_i x_i \tag{1.3}$$

De modo que las fronteras de clasificación están dadas por las líneas $hx+b = 1$ y $hx + b = -1$ centradas alrededor de $hx + b = 0$.

El código 4.2 en el anexo es una modificación de la implementación de la función `ksvm()` en el paquete `kernlab` de `R`, haciendo uso de los tutoriales de *Jean-Philippe Vert* para representar las líneas de separación de clasificación mediante un kernel lineal.

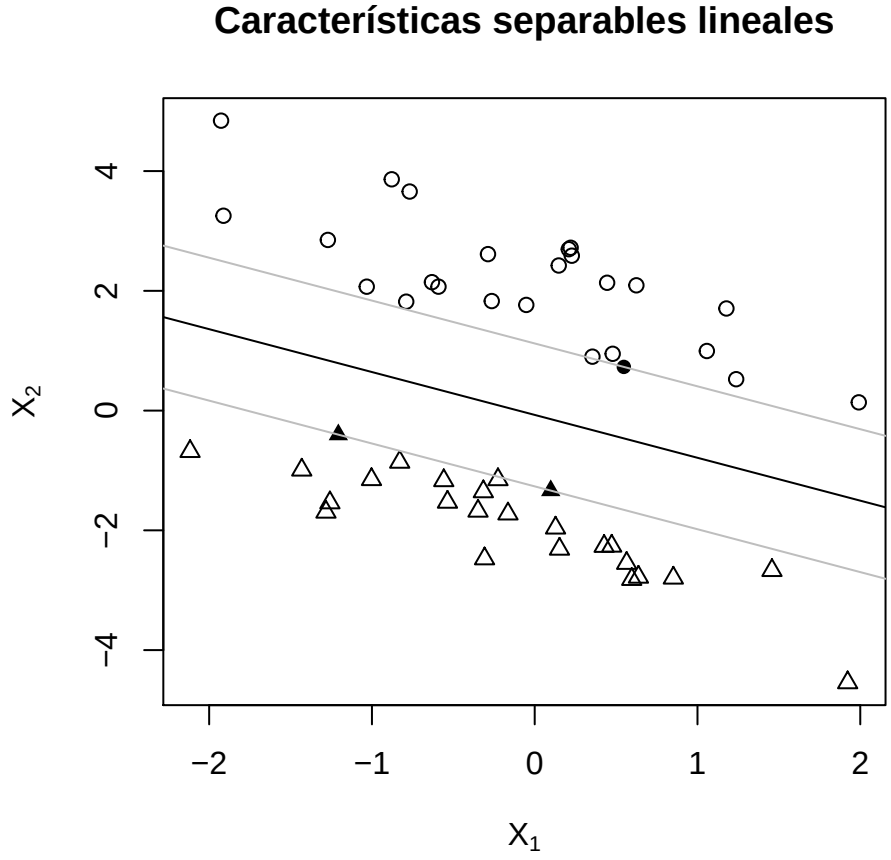


Figura 1.2: *Implementación de la función* `ksvm()` *en el paquete* `kernlab` *de* `R` *para representar las líneas de separación de clasificación.*

En la figura 1.3 se ilustra un ejemplo en el que las observaciones no están separados . Los puntos de la clase $y = 1$ (círculos) se colocan en una región interior rodeada por puntos de clase $y = -1$ (triángulos). En este ejemplo no hay una sola línea recta (lineal) que pueda separar ambas regiones. Sin embargo es posible encontrar un separador lineal mediante la transformación de los puntos $x = \{x_1, x_2\}$ del espacio de características a un espacio cuadrático de núcleos con puntos dados por las correspondientes coordenadas cuadradas $\{x_1^2, x_2^2\}$. El código en R puede consultarse en el código 4.3 en el anexo.

La técnica de transformar el espacio de características en una medida que permite una separación lineal puede formalizarse en términos de *kernel*. Suponiendo que $\Phi()$ sea una función de transformación vectorial de coordenadas, un espacio de coordenadas cuadráticas sería $\{\Phi(x_1), \Phi(x_2)\} = \{x_1^2, x_2^2\}$. La búsqueda de separación de la SVM está actuando ahora en el espacio transformado para encontrar los vectores de soporte que generan la condición:

$$h\Phi(x) + b = \pm 1 \tag{1.4}$$

Para el vector de hipótesis h:

$$h = \sum_i c_i \Phi\left(x_i\right) \tag{1.5}$$

Dada por la suma sobre los puntos vectoriales de soporte x_i. Poniendo ambas expresiones juntas obtenemos

$$\sum_i c_i K\left(x_i, x\right) + b = \pm 1 \tag{1.6}$$

Con la función de kernel escalar $K\left(x_i, x\right) = \Phi\left(x_i\right) \cdot \Phi\left(x\right)$. El kernel se compone del producto escalar entre un vector soporte x_i y otro punto vector x de características en el espacio transformado.

En la práctica, el algoritmo SVM puede expresarse completamente en términos de kernels sin tener que especificar realmente la transformación de espacio de entidad. Los núcleos populares son, por ejemplo, potencias superiores del producto escalar lineal (*kernel* polinomial). Otro ejemplo es una probabilidad pesada de distancia entre dos puntos (*kernel* gaussiano).

La implementación de una función de núcleo cuadrático bidimensional permite al algoritmo SVM encontrar vectores soporte y separar correctamente las regiones. En la figura 1.3 se muestra que regiones no lineales se pueden separar linealmente después de una transformación adecuada.

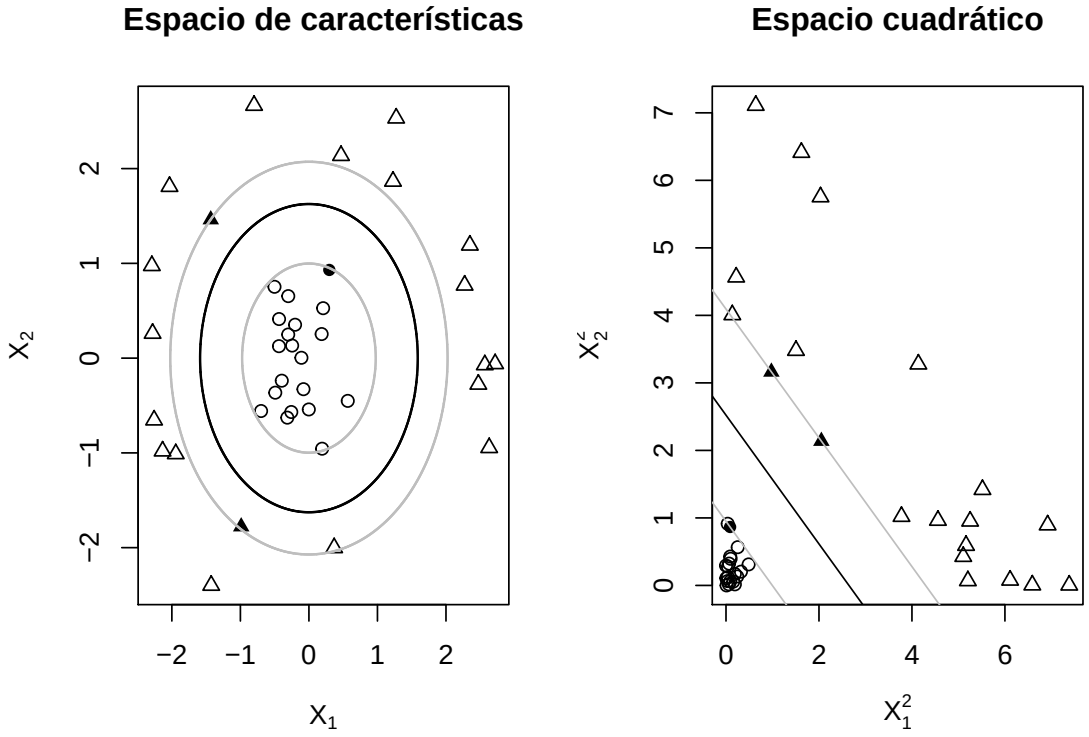


Figura 1.3: *Implementación de una función de núcleo cuadrático bidimensional permite al algoritmo SVM encontrar vectores soporte y separar correctamente las regiones.*

1.3.1.2. Ejemplos de funciones *kernel* más utilizadas

kernel lineal, el más sencillo de los posibles

$$k\left(x, x'\right) = \langle x, x' \rangle \tag{1.7}$$

kernel de base radial (RBF, Laplace Radial Basis Function)

$$k\left(x, x'\right) = \left(e^{-\sigma \cdot ||x - x'||}\right) \tag{1.8}$$

kernel de base radial (RBF, Gaussian Radial Basis Function)

$$k\left(x, x'\right) = \left(e^{-\sigma \cdot ||x - x'||^2}\right) \tag{1.9}$$

kernel polinomial

$$k\left(x, x'\right) = \left(\beta_1 \langle x, x' \rangle + \beta_0\right)^d \tag{1.10}$$

1.4. Aplicaciones

SVMs pueden ser utilizados para resolver varios problemas del mundo real, como por ejemplo:

- La clasificación de las imágenes se puede realizar usando SVMs. Los resultados experimentales muestran que las SVM logran una precisión de búsqueda significativamente mayor que otros algoritmos de clasificación supervisada.

- Los caracteres escritos a mano se pueden reconocer usando SVM. Estos algoritmos son conocidos como algoritmos ocr. Una aplicación muy conocida de reconocimiento de caracteres son los captchas.

1.5. SVM lineales

Dado un conjunto *n* de observaciones de entrenamiento de la forma:

$$(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \tag{1.11}$$

Donde y_i toma los valores -1 o 1, indicando la clase a la que pertenece cada punto $\vec{x}_i$. Cada h $\vec{x}_i$ es un vector de *p* dimensiones. Queremos encontrar el hiperplano de margen máximo que divide el grupo de puntos $\vec{x}_i$ entre los que verifican que $y_i = 1$ de conjunto de puntos que verifican $y_i = -1$. Este hiperplano es definido como aquel cuya distancia a los puntos más cercanos $\vec{x}_i$ de cada clase es máxima.

Un hiperplano puede ser descrito como el conjunto de puntos $\vec{x}$ que satisfacen la siguiente condición:

$$\vec{w} \cdot \vec{x} - b = 0 \tag{1.12}$$

Donde $\vec{w}$ es el vector normal (no necesariamente normalizado) al hiperplano. El parámetro $\frac{b}{\|\vec{w}\|}$ determina el desplazamiento de la hiperplano desde el origen a lo largo del vector normal $\vec{w}$.

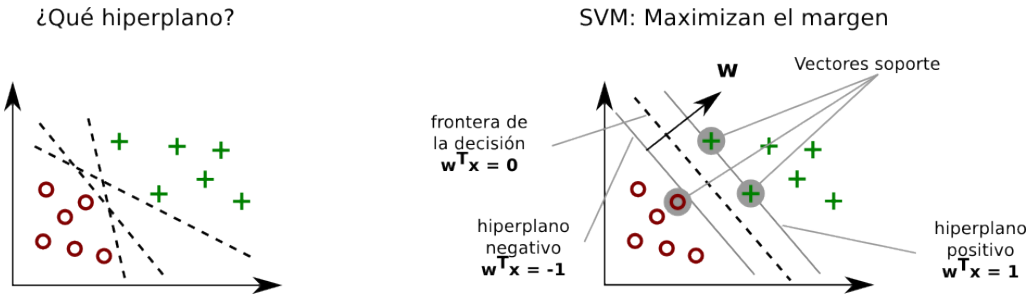


Figura 1.4: *Construcción del hiperplano que maximiza el margen entre las dos clases.*

INPUT		OUTPUT		
A	B	A AND B	A OR B	A XOR B
-1	-1	-1	-1	1
-1	1	-1	1	-1
1	-1	-1	1	-1
1	1	-1	1	1

Tabla 1.1: *Entradas y salidas de los diferentes operadores lógicos.*

1.6. Problemas AND, OR y XOR

1.6.1. Problema AND

AND es un operador lógico cuyo valor de la verdad resulta en cierto sólo si ambas proposiciones son ciertas, y en falso de cualquier otra forma. En la figura 1.5 vemos cómo la maquina vector soporte de kernel lineal, resuelve el problema usando tres soportes.

```
data.and <- data.frame(x = c(-1, -1, 1, 1),
                       y = c(-1, 1, -1, 1),
                       class = c(-1, -1, -1, 1))

modelo <- ksvm(class ~ .,
               data = data.and,
               type = "C-svc",
               kernel = "vanilladot")
```

```
##  Setting default kernel parameters
```

```
# table(predict(modelo), data.and$class)
plot(modelo,
     data = data.and,
     xlim = c(-1.1, 1.1),
     ylim = c(-1.1, 1.1))
```

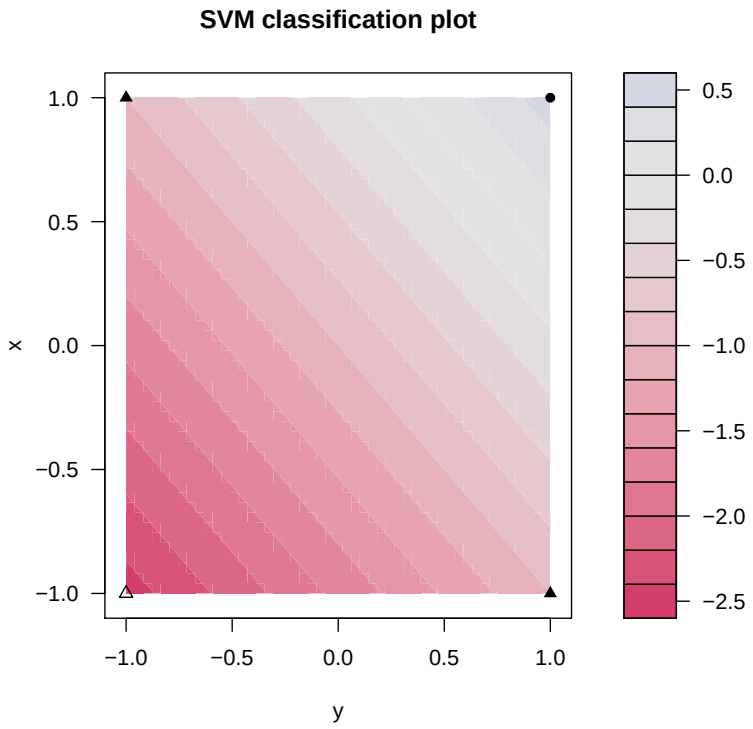


Figura 1.5: *Implementación de una máquina de vector soporte para resolver el problema AND.*

1.6.2. Problema OR

OR es un operador lógico que implementa la disyunción lógica y se comporta de acuerdo a la tabla 1.1.

```
data.or <- data.frame(x = c(-1, -1, 1, 1),
                      y = c(-1, 1, -1, 1),
                      class = c(-1, 1, 1, 1))

modelo <- ksvm(class ~ .,
               data = data.or,
               type = "C-svc",
               kernel = "vanilladot")
```

```
##  Setting default kernel parameters
```

```
# table(predict(modelo), data.or$class)
plot(modelo,
     data = data.or,
     xlim = c(-1.1, 1.1),
     ylim = c(-1.1, 1.1))
```

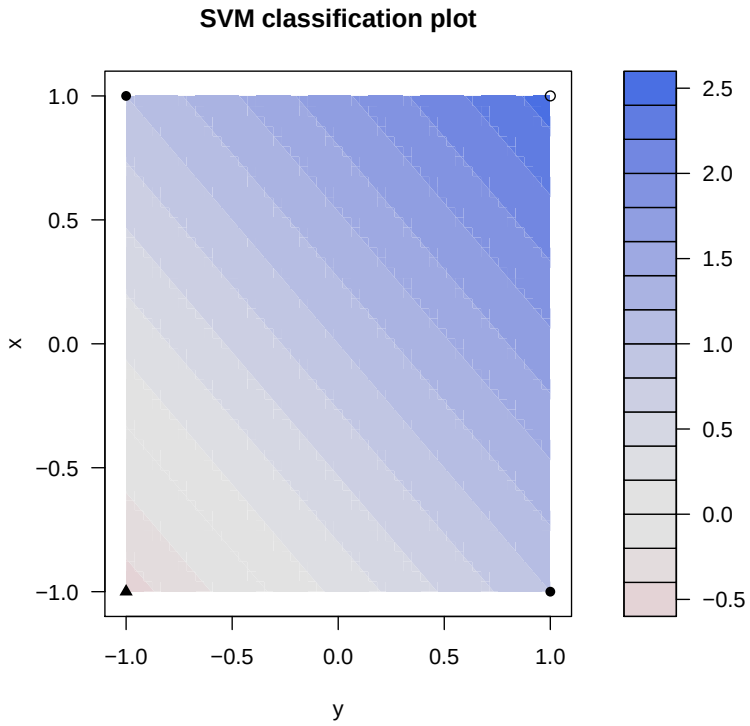


Figura 1.6: *Implementación de una máquina de vector soporte para resolver el problema OR.*

1.6.3. Problema XOR

XOR es un operador lógico que implementa la disyunción exclusiva y se comporta de acuerdo a la tabla 1.1. En este caso, aún siendo un ejemplo de sólo cuatro observaciones, su solución no es trivial aunque se puede resolver mediante un kernel radial.

```
data.xor <- data.frame(x = c(-1, -1, 1, 1),
                       y = c(-1, 1, -1, 1),
```

```
                        class = c(1, -1, -1, 1))

modelo <- ksvm(class ~ .,
               data = data.xor,
               type = "C-svc",
               kernel = "rbfdot")
# table(predict(modelo), data.xor$class)
plot(modelo, data = data.xor,
     xlim = c(-1.1, 1.1),
     ylim = c(-1.1, 1.1))
```

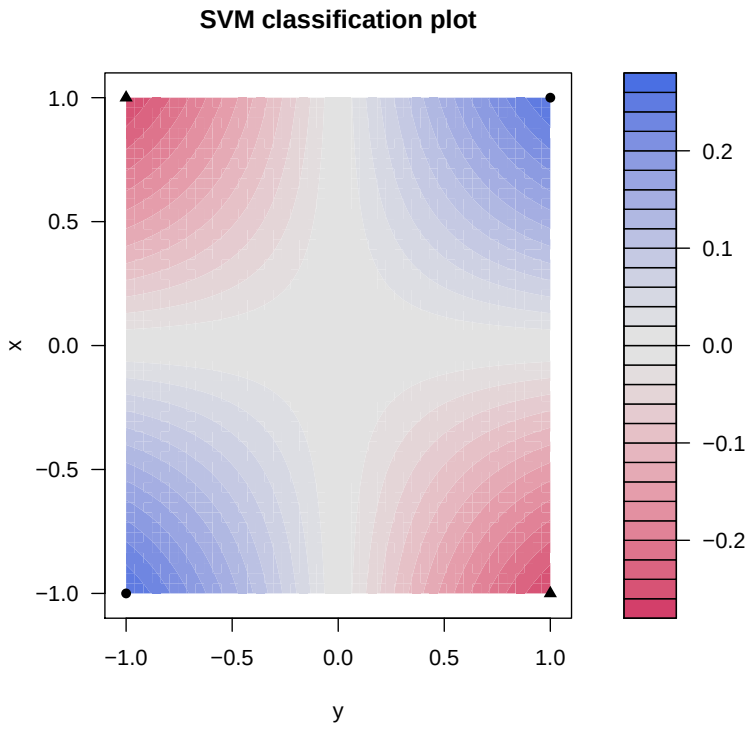


Figura 1.7: *Implementación de una máquina de vector soporte para resolver el problema XOR.*

1.7. Validación de los modelos SVM

Como todos los modelos supervisado, las SVM dependen de las observaciones de entrenamiento. Si cambian estas observaciones, los parámetros del modelo pueden cambiar. En las SVM se observa ademas que la construcción del hiperplano depende de los vectores soporte, si se modifican los vectores soporte se modifica el hiperplano, aunque el resto de observaciones sean las mismas.

Por este motivo, al generar un modelo SVM se debe analizar su robustez mediante un análisis de validación cruzada. Los análisis de validación nos permite valorar el grado de sobreajuste de nuestro modelo a los datos. Un modelo sobreajustado clasificará muy bien las muestras de entrenamiento, pero su exactitud será muy baja cuando pronostique nuevas observaciones.

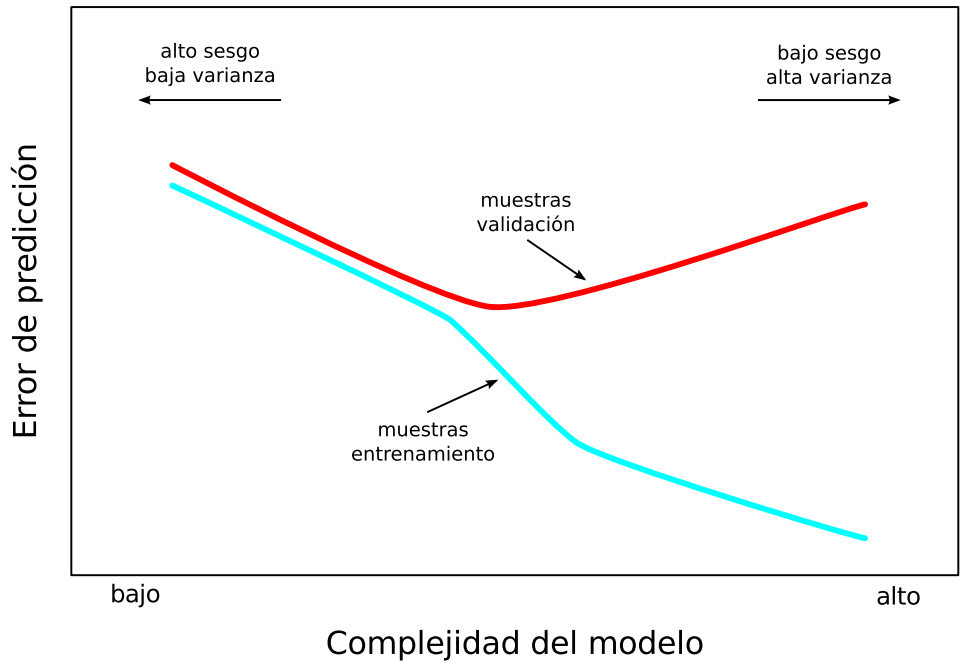


Figura 1.8: *Evolución del error en función en las muestras de entrenamiento y validación en función de la complejidad del modelo. The Elements of Statistical Learning, Trevor Hastie.*

1.7.1. Validación cruzada *leave-one-out*

La validación *leave-one-out* (validación dejando uno fuera) es una validación muy sencilla pero que computacionalmente puede ser muy costosa. El procedimiento de la validación *leave-one-out* se basa en lo siguiente: Si tenemos *n* observaciones entonces construimos *n* Máquinas de Vector Soporte, $\{SVM_1, SVM_2, \ldots, SVM_n\}$, cada una de ellas con *n-1* muestras, $SVM_i = \{1, 2, \ldots, i-1, i+1, \ldots, n\}$ y aplicamos el modelo sobre la muestra no utilizada en la construcción del modelo.

1.7.2. Validación cruzada *k-fold*

La validación *k-fold* consiste en separar las muestras en *k* grupos de mismo tamaño. Se construyen entonces *k* Máquinas de Vector Soporte con *k-1* grupos y se aplica el modelo resultante sobre el grupo de muestras no incluidas en el modelo. Observamos que si *k = n* estamos ante la validación *leave-one-out*.

2. Práctica: Construcción de una Máquina de Vector Soporte con R

Hay muchos paquetes en R que permiten construir SVM como por ejemplo el paquete e1071 o el paquete kernlab. Nosotros vamos a trabajar con el paquete caret un paquete diseñado por *Max Kuhn* precisamente para la búsqueda de modelos, en particular SVM.

La práctica que vamos a desarrollar consiste en la creación y validación de una Máquina de Vector Soporte que sea capaz de distinguir el dígito escrito en una imagen de entre los diez posibles dígitos: 0, 1, 2, . . . , 8 y 9. Los datos se han generado a partir de 130 imágenes por cada dígito en las que en cada imagen el dígito está escrito con una tipografía diferente. El código para generar dichas imágenes está escrito es el código 4.7 en el anexo.

En total tenemos 1300 imágenes de 16x16 píxeles cada una, por lo que tenemos 256 píxeles de información en cada imagen que podemos considerar como 256 variables. Mediante el programa imageMagick hemos pasado cada imagen a información en texto que podemos leer fácilmente con R. En el código 2.1 están descritas las primeras 25 líneas de uno de los ficheros generados.

```
1  # ImageMagick pixel enumeration: 16,16,255,srgb
2  0,0: (255,255,255)  #FFFFFF  white
3  1,0: (255,255,255)  #FFFFFF  white
4  2,0: (255,255,255)  #FFFFFF  white
5  3,0: (255,255,255)  #FFFFFF  white
6  4,0: (255,255,255)  #FFFFFF  white
7  5,0: (255,255,255)  #FFFFFF  white
8  6,0: (255,255,255)  #FFFFFF  white
9  7,0: (255,255,255)  #FFFFFF  white
10 8,0: (255,255,255)  #FFFFFF  white
11 9,0: (255,255,255)  #FFFFFF  white
12 10,0: (255,255,255)  #FFFFFF  white
13 11,0: (255,255,255)  #FFFFFF  white
14 12,0: (255,255,255)  #FFFFFF  white
15 13,0: (255,255,255)  #FFFFFF  white
```

```
14,0: (255,255,255)  #FFFFFF  white
15,0: (255,255,255)  #FFFFFF  white
0,1: (255,255,255)  #FFFFFF  white
1,1: (255,255,255)  #FFFFFF  white
2,1: (255,255,255)  #FFFFFF  white
3,1: (255,255,255)  #FFFFFF  white
4,1: (255,255,255)  #FFFFFF  white
5,1: (255,255,255)  #FFFFFF  white
6,1: (255,255,255)  #FFFFFF  white
7,1: (255,255,255)  #FFFFFF  white
```

Código 2.1: *ImageMagick pixel enumeration: 30, 30, 255, srgb.*

El código anterior son las primeras líneas del fichero en texto de una imagen. La primera fila es la descripción del fichero, la segunda línea nos informa que el punto (0,0) de la imagen es de color blanco. La información está en *rgb*, *hexadecimal* y en modo texto (*white*, *black*). Nosotros utilizaremos la última columna como la información de nuestras variables.

En la figura 2.9 se muestra como ejemplo las 130 imágenes del dígito 8 que utilizaremos para entrenar y validar el modelo de Máquinas de Vector Soporte.

Dibujamos el perfil promedio de cada dígito. Las figuras, generadas a 16x16 pixeles de resolución nos dan un total de 256 variables en las que tenemos los valores *white* o *black*, una variable por cada píxel de la imagen. Nosotros pasaremos estos valores a -1 y 1. Cuanto mayor sea la resolución de la imagen mayor será también el número de variables. Las diferencias claras entre los distintos números, que hace que seamos capaces de distinguirlos, tienen que verse también en el perfil generado por las 256 variables y estas diferencias son las que tiene que encontrar la Máquina de Vector Soporte. En la figura 2.10 se muestran la distribución promedio de cada píxel para cada dígito.

2.1. Support Vector Machine Learning

Como ya hemos comentado, las máquinas de vector soporte dependen de la función kernel que consideremos. Las más usuales son las funciones kernel lineales y las funciones kernel de base radial [1]. Con los datos de las 1300 imágenes vamos a construir diferentes Máquinas de Vector Soporte utilizando diferentes argumentos y parámetros y compararemos sus resultados.

2.1.1. Linear Support Vector Machine

Estimamos la exactitud de una máquina de vector soporte con función kernel lineal. La forma de atacar este problema por parte de la librería `caret` [2,3] es evaluando cada punto de la malla generada por los diferentes valores de los parámetros *sigma* (σ) y *costo* (c). Una vez evaluados todos los paŕametros elegiremos aquél par (*sigma*, *costo*) que haya maximizado la exactitud de las predicciones. Valoramos la construcción del modelo considerando 5 repeticiones con el 80 % del total de la muestra (`trainControl(method = "cv", p = 0.8, digito = 5, repeats = 5, search = "grid")`). Para evaluar un modelo de máquinas de vector soporte con kernel lineal, le pasamos a la función `train` el argumento `method = "svmLinear2"` que será evaluada internamente con las funciones del paquete `kernlab` [4].

En metabolómica, es habitual realizar diferentes transformaciones en los datos para conseguir distribuciones normales o distribuciones tipificadas entre otras muchas [5]. La función `train` permite considerar diferentes transformaciones sobre los datos. Las más comunes son:

- **Datos originales**. Trabajaremos con los datos originales, sin ningún tipo de preprocesado previo utilizando el comando `preProcess = NULL`.
- **Datos escalados y centrados**. Si las variables tienen diferentes escalas y variabilidad es interesante considerar el centrado de variables y su escalado, para que no influya en el análisis. Utilizaremos el comando `preProcess =`

`c("scale", "center")`.

$$X' = \frac{X - \bar{X}}{\sigma_X} \tag{2.1}$$

- **Datos transformados según potencias *Box-Cox***. Las transformacion es *Box-Cox* son una familia de transformaciones cuya finalidad principal es normalizar una variable. El comando que podemos utilizar es `preProcess = "BoxCox"`.

$$y_i^{(\lambda)} = \begin{cases} \dfrac{y_i^{\lambda} - 1}{\lambda} & \text{si } \lambda \neq 0, \\ \ln(y_i) & \text{si } \lambda = 0, \end{cases} \tag{2.2}$$

La figura 2.12 muestra la proyección sobre las dos primeras componentes las observaciones que utilizaremos para entrenar al modelo (65 observaciones de cada dígito). Vemos que hay una separación evidente entre algunos dígitos y otros están juntos. Comprobamos que las observaciones de los dígitos 1 y 7 están cercanas, así como las observaciones de los dígitos 0, 6 y 8.

Estas diferencias visibles en el Análisis de Componentes Principales ya nos hacen intuir que un modelo SVM puede tener un gran capacidad de pronóstico. El código en `R` para generar el PCA puede consultarse en el código 4.4 en el anexo.

Vamos a aplicar la función `train()` con el mínimo número de argumentos:

- **x**: Matriz de datos.
- **y**: Clasificación de las muestras.
- **method**: Una cadena de texto en la que se especifica el modelo de clasificación o regresión a considerar. En nuestro caso vamos a utilizar `svmLinear2` que es un *kernel* disponible desde la librería `e1071`.

```
X <- entrenamiento[, 1:256]
Y <- entrenamiento$digito
modelo <- train(x = X,
                y = Y,
                method = "svmLinear2")
```

La función `train()` devuelve un objeto de la clase `train` que es una lista que contiene los siguientes elementos:

- **method**: El modelo elegido. Es el que hemos pasado como argumento.
- **modelType**: Un identificador del tipo de modelo.
- **results**: Un `data.frame` de datos la tasa de error de entrenamiento y los valores de los parámetros de ajuste.
- **bestTune**: Un `data.frame` de datos con los parámetros finales.
- **metric**: Una cadena de texto que especifica qué métrica de resumen se utilizará para seleccionar el modelo óptimo.
- **control**: La lista de parámetros de control.
- **preProcess**: `NULL` o un objeto de clase `preProcess`.
- **finalModel**: Un objeto de ajuste utilizando los mejores parámetros.
- **trainingData**: Un `data.frame`.
- **resample**: Un `data.frame` con columnas para cada métrica de rendimiento. Cada fila corresponde a cada re-muestreo. Si se solicitan los métodos de validación cruzada o de validación fuera de bolsa, estos valores serán `NULL`.
- **perfNames**: Un vector de caracteres de métricas de rendimiento que se producen mediante la función `summary()`.

- **maximize**: Un valor lógico que proviene de los argumentos de la función.
- **yLimits**: El rango de los resultados del conjunto de entrenamiento.

Primero, entrenamos una SVM con un *kernel* lineal. En este caso utilizamos la función `svmLinear2` del paquete `e1071`. En modelos disponibles se pueden consultar los modelos disponibles. También se puede ejecutar el comando `names(getModelInfo())`. Los modelos específicos para construir Máquinas de Vector Soporte pueden consultarse en train models.

En la tabla 2.2 se muestran 16 descritos los métodos para trabajar con la función `train()` para construir Máquinas de Vector Soporte.

Tras aplicar la función `train()` sobre nuestros datos obtenos el objeto `modelo`.

```
print(modelo)
```

```
## Support Vector Machines with Linear Kernel
##
## 650 samples
## 256 predictors
##  10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 650, 650, 650, 650, 650, 650, ...
## Resampling results across tuning parameters:
##
##   cost  Accuracy   Kappa
##   0.25  0.9892014  0.9879686
##   0.50  0.9892014  0.9879686
##   1.00  0.9892014  0.9879686
##
## Accuracy was used to select the optimal model using the largest valu
```

```
## The final value used for the model was cost = 0.25.
```

El modelo con estructura de SVM guardado en `modelo` es una lista con varios argumentos.

```
modelo$method
```

```
## [1] "svmLinear2"
```

```
modelo$modelType
```

```
## [1] "Classification"
```

```
modelo$results
```

```
##   cost  Accuracy     Kappa  AccuracySD     KappaSD
## 1 0.25 0.9892014 0.9879686 0.006443373 0.007176749
## 2 0.50 0.9892014 0.9879686 0.006443373 0.007176749
## 3 1.00 0.9892014 0.9879686 0.006443373 0.007176749
```

```
modelo$bestTune
```

```
##   cost
## 1 0.25
```

```
modelo$call
```

```
## train.default(x = X, y = Y, method = "svmLinear2")
```

```
modelo$metric
```

```
## [1] "Accuracy"
```

```
names(modelo$control)
```

```
##  [1] "method"          "number"          "repeats"
##  [4] "search"          "p"               "initialWindow"
##  [7] "horizon"         "fixedWindow"     "skip"
## [10] "verboseIter"     "returnData"      "returnResamp"
```

```
## [13] "savePredictions"    "classProbs"         "summaryFunction"
## [16] "selectionFunction" "preProcOptions"     "sampling"
## [19] "index"              "indexOut"           "indexFinal"
## [22] "timingSamps"        "predictionBounds"   "seeds"
## [25] "adaptive"           "trim"               "allowParallel"
```

```
modelo$preProcess
```

```
## NULL
```

```
modelo$finalModel
```

```
##
## Call:
## svm.default(x = as.matrix(x), y = y, kernel = "linear", cost = param
##     probability = classProbs)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.25
##       gamma:  0.00390625
##
## Number of Support Vectors:  405
```

En la tabla 2.3 se muestra la matriz de confusión entre los dígitos observados (columnas) y los dígitos pronosticados (filas) de los 1000 dígitos de los datos utilizados en el entrenamiento del modelo.

En la tabla 2.4 se muestra la matriz de confusión entre los dígitos observados (columnas) y los dígitos pronosticados (filas) de los 300 dígitos de los datos utilizados en la validación del modelo.

Observamos que hay una tasa de error en la validación del 0.0767.

Utilizamos la función `varImp()` del paquete `caret`. Esta función tiene como argumento la salida de la función `train`, es decir, el modelo construido. La salida de la función `varImp()` depende del modelo que sele haya pasado como argumento. En nuestro caso la salida es una lista de tres elementos:

- **importance**: Es un `data.frame` con tantas filas como variables y tantas columnas como clases a separar que contiene un índice de importancia para discriminar entre clases que va de 0 a 100 (`scaled = TRUE`) .
- **model**: Modelo que permite calcular el índice de importancia entre clases. Por defecto se utiliza un análisis ROC.
- **calledFrom**: Nos informa de la llamada a la función.

En la figura 2.15 se muestra el gráfico por defecto al dibujar el objeto devuelto por la función `varImp()`.

```
a <- varImp(modelo)
colnames(a$importance) <- paste("dígito",
                                seq(0, 9),
                                sep = " ")
plot(a, top = 20,
     xlab = "Variables (píxeles) más importantes")
```

En la figura 2.16 representamos la importancia de cada variable como si fueran los píxeles de una imagen. Como era evidente, los márgenes de las imágenes no son importantes para discriminar entre los diferentes dígitos (blanco y gris) y si son más importantes los píxeles centrales de las imágenes (negro).

Figura 2.9: *130 gráficos del número ocho con cada fuente.*

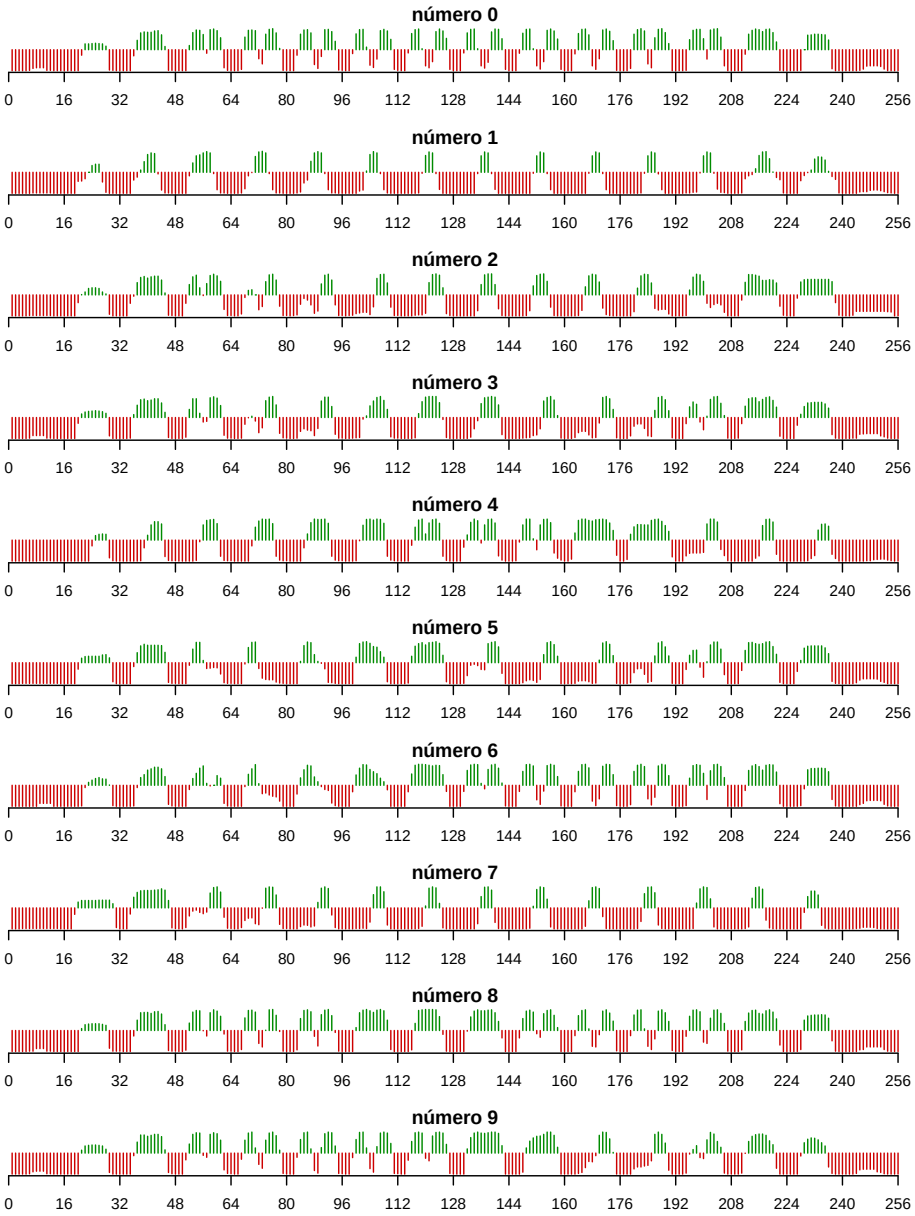


Figura 2.10: *Distribución.*

Figura 2.11: *Ejemplo de los dígitos a reconocer tras haber añadido ruido a la imagen.*

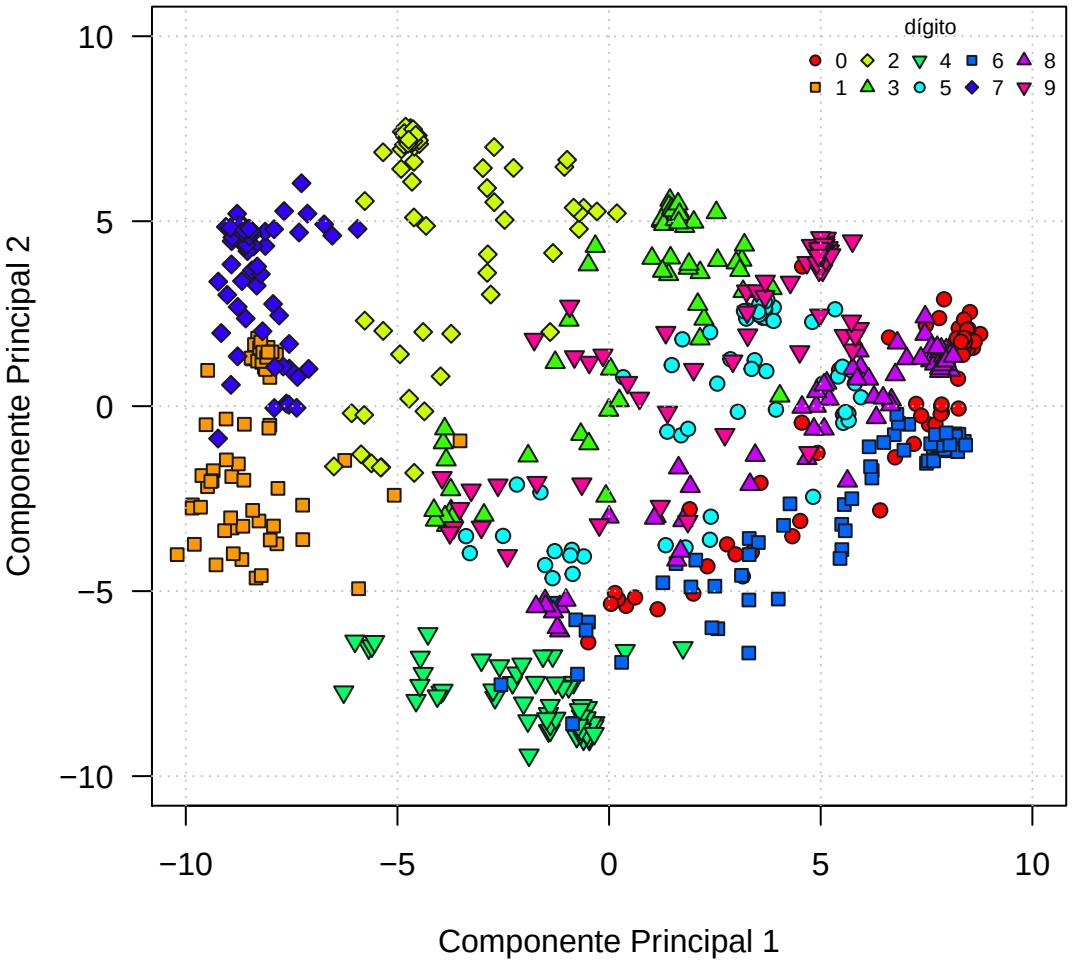


Figura 2.12: *Análisis de Componentes Principales con 100 observaciones por cada dígito. Comprobamos que las observaciones de los dígitos 1 y 7 están cercanas, así las observaciones de los dígitos 0, 6 y 8. Además, estos dos grupos de observaciones están situados en grupos opuestos en la primera componente, así que podríamos interpretar que la primera componente está relacionada con el volumen del número.*

Model	Method	Type	Libraries	Tuning parameters
L2 Regularized Linear Support Vector Machines with Class Weights	svmLinearWeights2	Classification	LiblineaR	cost, Loss, weight
L2 Regularized Support Vector Machine (dual) with Linear Kernel	svmLinear3	Classification, Regression	LiblineaR	cost, Loss
Least Squares Support Vector Machine	lssvmLinear	Classification	kernlab	tau
Least Squares Support Vector Machine with Polynomial Kernel	lssvmPoly	Classification	kernlab	degree, scale, tau
Least Squares Support Vector Machine with Radial Basis Function Kernel	lssvmRadial	Classification	kernlab	sigma, tau
Linear Support Vector Machines with Class Weights	svmLinearWeights	Classification	e1071	cost, weight
Support Vector Machines with Boundrange String Kernel	svmBoundrangeString	Classification, Regression	kernlab	length, C
Support Vector Machines with Class Weights	svmRadialWeights	Classification	kernlab	sigma, C, Weight
Support Vector Machines with Exponential String Kernel	svmExpoString	Classification, Regression	kernlab	lambda, C
Support Vector Machines with Linear Kernel	svmLinear	Classification, Regression	kernlab	C
Support Vector Machines with Linear Kernel	svmLinear2	Classification, Regression	e1071	cost
Support Vector Machines with Polynomial Kernel	svmPoly	Classification, Regression	kernlab	degree, scale, C
Support Vector Machines with Radial Basis Function Kernel	svmRadial	Classification, Regression	kernlab	sigma, C
Support Vector Machines with Radial Basis Function Kernel	svmRadialCost	Classification, Regression	kernlab	C
Support Vector Machines with Radial Basis Function Kernel	svmRadialSigma	Classification, Regression	kernlab	sigma, C
Support Vector Machines with Spectrum String Kernel	svmSpectrumString	Classification, Regression	kernlab	length, C

Tabla 2.2: *Modelos disponibles para construir Máquinas de Vector Soporte con la función train().*

	0	1	2	3	4	5	6	7	8	9
0	65	0	0	0	0	0	0	0	0	0
1	0	65	0	0	0	0	0	0	0	0
2	0	0	65	0	0	0	0	0	0	0
3	0	0	0	65	0	0	0	0	0	0
4	0	0	0	0	65	0	0	0	0	0
5	0	0	0	0	0	65	0	0	0	0
6	0	0	0	0	0	0	65	0	0	0
7	0	0	0	0	0	0	0	65	0	0
8	0	0	0	0	0	0	0	0	65	0
9	0	0	0	0	0	0	0	0	0	65

Tabla 2.3: *Matriz de confusión entre los dígitos observados y los dígitos pronosticados por el modelo.*

	0	1	2	3	4	5	6	7	8	9
0	61	0	0	0	0	1	1	0	0	2
1	0	64	1	0	0	0	0	1	1	1
2	0	0	62	0	0	0	0	0	0	0
3	0	0	1	64	0	1	0	0	0	1
4	2	0	0	0	65	0	0	0	1	0
5	0	0	0	0	0	60	0	0	0	0
6	0	0	0	0	0	3	64	0	0	0
7	0	1	0	0	0	0	0	64	1	0
8	2	0	1	1	0	0	0	0	62	0
9	0	0	0	0	0	0	0	0	0	61

Tabla 2.4: *Matriz de confusión entre los dígitos de validación y los dígitos pronosticados por el modelo.*

Figura 2.13: *Casos en los que ha fallado el modelo a la hora de interpretar un dígito. El dígito del centro (negrita) representa el número que la Máquina de Vector Soporte no ha sabido interpretar. El dígito de la esquina inferior (gris) representa el dígito que ha predicho el modelo.*

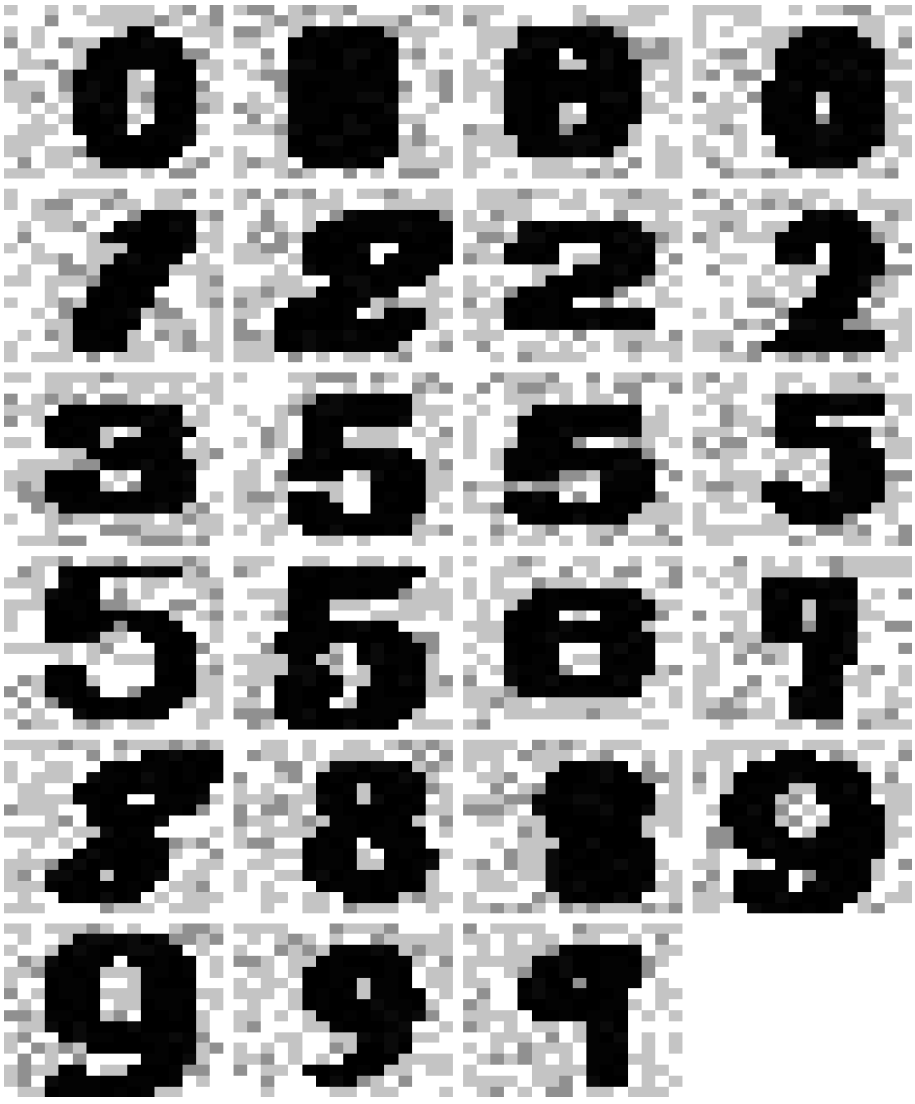


Figura 2.14: *Casos en los que ha fallado el modelo a la hora de interpretar un dígito. Los dígitos se representan tal y como se ven en las imágenes.*

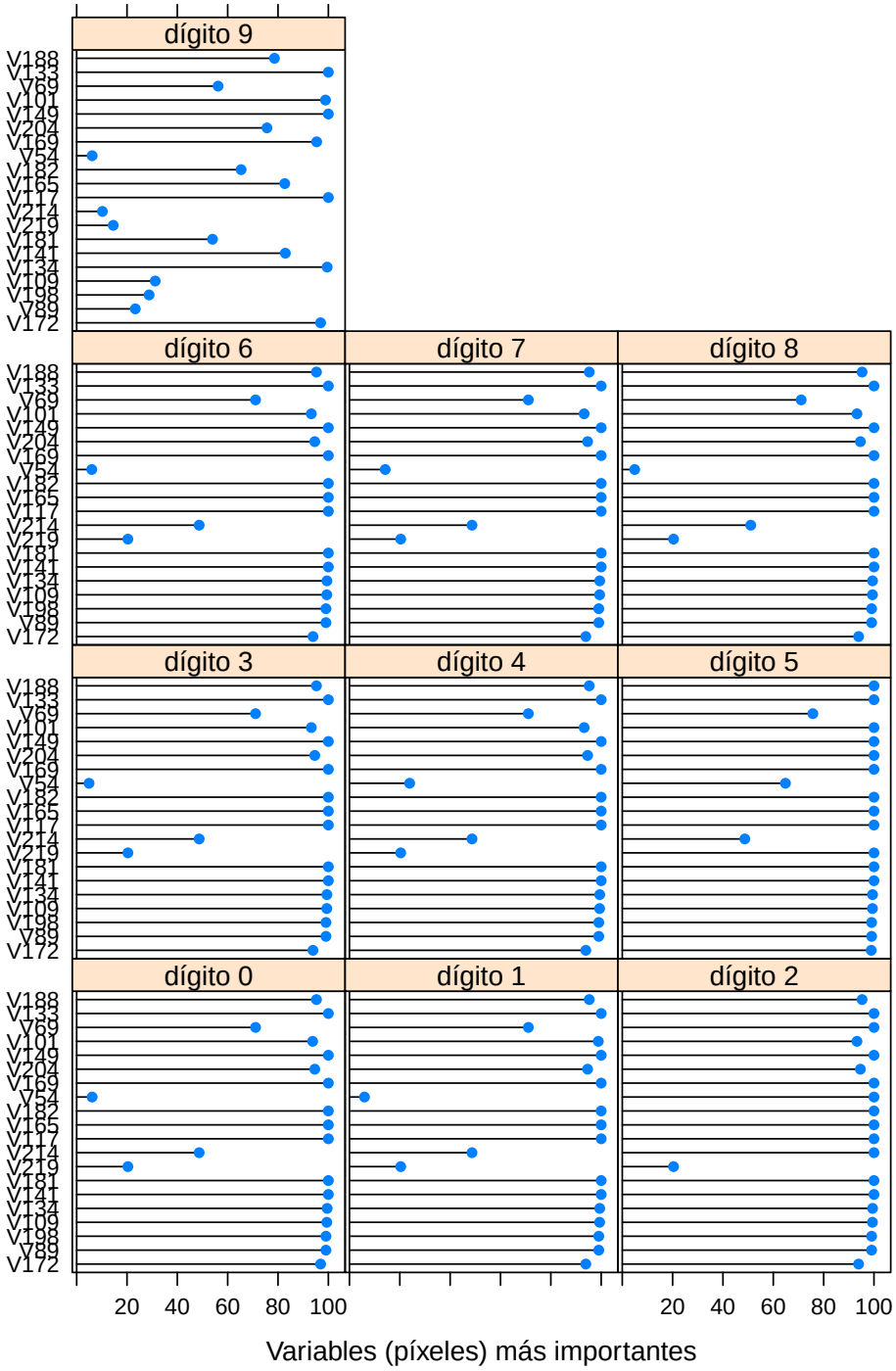


Figura 2.15: *Las 20 variables más importantes para la discriminación de dígitos, es decir, qué pixeles son más importantes para clasificar.*

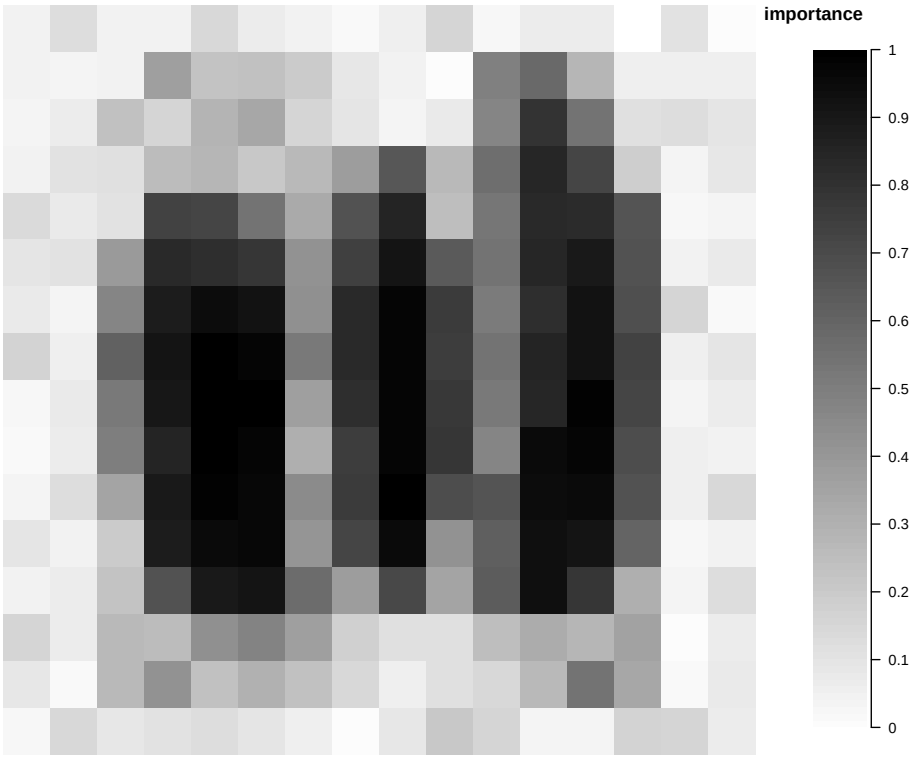


Figura 2.16: *Los píxeles más importantes a la hora de tomar una decisión. En blanco se señalan los píxeles menos significativos y en negro están marcados los más significativos en promedio para discriminar entre dígitos.*

2.1.2. Linear Support Vector Machine. Modificación de argumentos y parámetros

En el ejemplo que hemos realizado no hemos modificado ningún parámetro y todos los argumentos necesarios han sido los que vienen por defecto. En el siguiente ejemplo añadimos varios párametros más:

- **preProcess**: Argumento que indica si existe algún preprocesamiento previo de los datos. En este casos escalamos y centramos las variables. Dada la naturaleza de los datos este proceso no va a mejorar los resultados previos, pero con otro tipo de datos puede ser interesante. Por defecto, este valor es `NULL`.

- **trControl**: Argumento al que se le puede pasar una lista que controla el proceso de búsqueda del modelo. En este caso pasamos los siguientes controles:

  - **method = cv**: Método de remuestreo. Se pueden elegir entre varios métodos, algunos específicos del modelo a buscar: svm, rf, knn, ...

  - **p = 0.8**: Porcentaje de muestras utilizadas en el entrenamiento de la validación cruzada.

  - **number = 5**: Número de iteraciones o número de subgrupos de validación (k-fold).

  - **repeats = 5**: Repetición de k-fold de la validación cruzada. En este caso tomamos $k = 5$.

- **tuneLength**: Número de veces que se ejecutará la búsqueda de un modelo variando los parámetros propios de la función *kernel*. En este caso, se variará el parámetro *cost* del *kernel* lineal.

- **metric**: Argumento que le indica a la función cuál va a ser el criterio para elegir los valores óptimos del modelo. En el caso lineal, nos mostrará cuál es

el costo óptimo para el cuál se alcanzará la exactitud máxima. Las métricas disponibles son:

- **RMSE** y R^2 para regresión.
- **accuracy** y **kappa** para clasificación.

- **maximize**: Valor lógico que indica a la función si debe buscar el máximo o el mínimo de la métrica. Por ejemplo, buscaremos minimizar el *RMSE* en regresión pero maximizaremos el *Accuracy* en clasificación.

Se pueden consultar las opciones de remuestreo.

El siguiente código en `R` construye una Máquina de Vector Soporte realizando un pre-procesado de los datos escalándolos y centrándolos. Además, utiliza un control de validación cruzada, en el que se usa el 80 % de las muestras

```
X <- entrenamiento[, 1:256]
Y <- entrenamiento$digito

modelo <- train(x = X,
                y = Y,
                preProcess = c("scale", "center"),
                trControl = trainControl(method = "cv",
                                         p = 0.8,
                                         number = 5,
                                         repeats = 5),
                method = "svmLinear2",
                tuneLength = 10,
                maximize = TRUE,
                metric = "Accuracy")
```

Tras aplicar la función `train()` sobre nuestros datos obtenos el objeto `modelo`.

```
print(modelo)
```

```
## Support Vector Machines with Linear Kernel
##
## 650 samples
## 256 predictors
##  10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## Pre-processing: scaled (256), centered (256)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 520, 520, 520, 520, 520
## Resampling results across tuning parameters:
##
##   cost    Accuracy   Kappa
##     0.25  0.9892308  0.9880342
##     0.50  0.9892308  0.9880342
##     1.00  0.9892308  0.9880342
##     2.00  0.9892308  0.9880342
##     4.00  0.9892308  0.9880342
##     8.00  0.9892308  0.9880342
##    16.00  0.9892308  0.9880342
##    32.00  0.9892308  0.9880342
##    64.00  0.9892308  0.9880342
##   128.00  0.9892308  0.9880342
##
## Accuracy was used to select the optimal model using the largest valu
## The final value used for the model was cost = 0.25.
```

En la tabla 2.5 se muestra la matriz de confusión entre los dígitos observados (columnas) y los dígitos pronosticados (filas) de los 300 dígitos de los datos utilizados en la validación del modelo.

Observamos que hay una tasa de error en la validación del 0.0767.

	0	1	2	3	4	5	6	7	8	9
0	61	0	0	0	0	1	1	0	0	2
1	0	64	1	0	0	0	0	1	1	1
2	0	0	62	0	0	0	0	0	0	0
3	0	0	1	64	0	1	0	0	0	1
4	2	0	0	0	65	0	0	0	1	0
5	0	0	0	0	0	60	0	0	0	0
6	0	0	0	0	0	3	64	0	0	0
7	0	1	0	0	0	0	0	64	1	0
8	2	0	1	1	0	0	0	0	62	0
9	0	0	0	0	0	0	0	0	0	61

Tabla 2.5: *Matriz de confusión entre los dígitos de validación y los dígitos pronosticados por el modelo.*

Figura 2.17: *Casos en los que ha fallado el modelo a la hora de interpretar un dígito. El dígito del centro (negrita) representa el número que la Máquina de Vector Soporte no ha sabido interpretar. El dígito de la esquina inferior (gris) representa el dígito que ha predicho el modelo.*

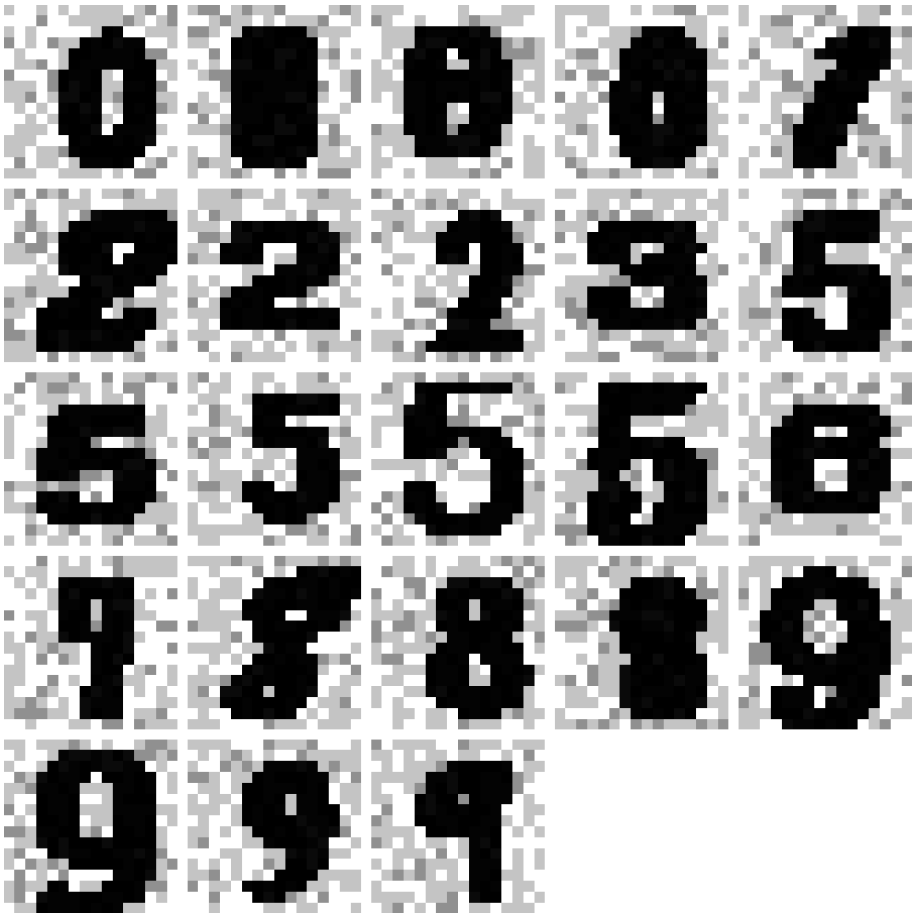


Figura 2.18: *Casos en los que ha fallado el modelo a la hora de interpretar un dígito. Los dígitos se representan tal y como se ven en las imágenes.*

2.2. Conclusiones

2.2.1. Ventajas

- Las Máquinas de Vector Soporte trabajan de forma eficiente con grandes cantidades de datos.
- Permiten realizar cribados iniciales de variables evaluando la importancia de cada una de ellas en la predicción de clases.

2.2.2. Desventajas

- *A priori* no existe un *kernel* óptimo, dependerá de la naturaleza de los datos con los que estemos trabajando y siempre será recomendable el uso de varios *kernel* y comparar sus resultados.
- La experiencia nos lleva a pensar que las Máquinas de Vector Soporte tienden a sobreajustar los datos y siempre es conveniente realizar una validación del modelo.
- Los modelos que se obtienen al aplicar una Máquina de Vector Soporte suelen ser muy opacos y de dificil interpretación.

3. Versión

- R version 3.4.4 (2018-03-15), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=es_ES.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=es_ES.UTF-8`, `LC_COLLATE=es_ES.UTF-8`, `LC_MONETARY=es_ES.UTF-8`, `LC_MESSAGES=es_ES.UTF-8`, `LC_PAPER=es_ES.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=es_ES.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 14.04.5 LTS`
- Matrix products: default
- BLAS: `/usr/lib/libblas/libblas.so.3.0`
- LAPACK: `/usr/lib/lapack/liblapack.so.3.0`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: ca0.71, caret6.0-81, e10711.7-0, extrafont0.17, Formula1.2-3, ggplot23.1.0, Hmisc4.1-1, kernlab0.9-27, knitr1.20, lattice0.20-35, pheatmap1.0.12, pROC1.13.0, randomForest4.6-14, RColorBrewer1.1-2, survival2.42-6, tables0.8.7, xtable1.8-3
- Loaded via a namespace (and not attached): acepack1.4.1, assertthat0.2.0, backports1.1.2, base64enc0.1-3, bindr0.1.1, bindrcpp0.2.2, checkmate1.8.5, class7.3-14, cluster2.0.7-1, codetools0.2-15, colorspace1.2-6, compiler3.4.4, crayon1.3.4, data.table1.11.8, digest0.6.8, dplyr0.7.7, evaluate0.11, extrafontdb1.0, foreach1.4.2, foreign0.8-70, generics0.0.2, glue1.3.0, gower0.1.2, grid3.4.4, gridExtra2.3, gtable0.1.2, htmlTable1.13.1, htmltools0.3.6, htmlwidgets1.3, ipred0.9-8, iterators1.0.7, latticeExtra0.6-28, lava1.6.3, lazyeval0.2.1, lubridate1.7.4, magrittr1.5, MASS7.3-50, Matrix1.2-14, ModelMetrics1.2.2, munsell0.5.0, nlme3.1-137, nnet7.3-12,

pillar1.3.0, pkgconfig2.0.2, plyr1.8.1, prodlim2018.04.18, purrr0.2.5, R62.3.0, Rcpp1.0.0, recipes0.1.4, reshape21.4.1, rlang0.3.0.1, rmarkdown1.10, rpart4.1-13, rprojroot1.3-2, rstudioapi0.8, Rttf2pt11.3.7, scales1.0.0, splines3.4.4, stats43.4.4, stringi1.2.4, stringr1.3.1, tibble1.4.2, tidyselect0.2.5, timeDate3043.102, tools3.4.4, withr2.1.2, yaml2.2.0

Bibliografía

[1] Karatzoglou A, Meyer D, Hornik K. Support vector machines in r. Journal of Statistical Software [Internet]. 2006;15:1–28. Available from: https://www.jstatsoft.org/index.php/jss/article/view/v015i09.

[2] Kuhn M. Building predictive models in r using the caret package. Journal of Statistical Software [Internet]. 2008;28:1–26. Available from: https://www.jstatsoft.org/index.php/jss/article/view/v028i05.

[3] Max Kuhn JW, Weston S, Williams A, et al. Caret: Classification and regression training [Internet]. 2015. Available from: http://CRAN.R-project.org/package=caret.

[4] Karatzoglou A, Smola A, Hornik K, et al. Kernlab – an S4 package for kernel methods in R. Journal of Statistical Software [Internet]. 2004;11:1–20. Available from: http://www.jstatsoft.org/v11/i09/.

[5] Berg RA van den, Hoefsloot HCJ, Westerhuis JA, et al. Centering, scaling, and transformations: Improving the biological information content of metabolomics data. BMC Genomics [Internet]. 2006;7:142. Available from: http://dx.doi.org/10.

1186/1471-2164-7-142.

[6] Krzywinski M, Altman N. Points of significance: Two-factor designs. Nature Methods. 2014;11:1187–1188.

[7] Altman N, Krzywinski M. Simple linear regression. Nature Methods. 2015;12:999–1000.

[8] Altman N, Krzywinski M. Points of significance: Analyzing outliers: Influential or nuisance? Nature Methods. 2016;13:281–282.

[9] Altman N, Krzywinski M. Points of significance: Regression diagnostics. Nature Methods. 2016;13:385–386.

[10] Kuehl R, Osuna M. Diseño de experimentos: Principios estadísticos de diseño y análisis de investigación. International Thomson Editores, S. A. de C. V. 2001.

[11] Pulido H, Vara Salazar R de la, González P, et al. Análisis y diseño de experimentos. McGraw-Hill; 2004.

[12] Martínez-Arranz I, Mayo R, Pérez-Cormenzana M, et al. Enhancing metabolomics research through data mining. Journal of Proteomics. 2015;127, Part B:275–288.

[13] Xie Y. Knitr: A comprehensive tool for reproducible research in R. In: Stodden V, Leisch F, Peng RD, editors. Implementing reproducible computational research [Internet]. Chapman; Hall/CRC; 2014. Available from: http://www.crcpress.

com/product/isbn/9781466561595.

[14] Xie Y. Dynamic documents with R and knitr [Internet]. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC; 2015. Available from: http://yihui.name/knitr/.

[15] Xie Y. Knitr: A general-purpose package for dynamic report generation in r [Internet]. 2016. Available from: http://yihui.name/knitr/.

[16] Armitage EG, Barbas C. Metabolomics in cancer biomarker discovery: Current trends and future perspectives. J Pharm Biomed Anal. 2014;87:1–11.

[17] Čuperlović-Culf M. 5 - metabolomics data analysis – processing and analysis of a dataset. In: Čuperlović-Culf M, editor. {NMR} metabolomics in cancer research [Internet]. Woodhead Publishing; 2013. pp. 261–333. Available from: http://www.sciencedirect.com/science/article/pii/B9781907568848500056.

[18] Fox J. Applied regression analysis, linear models, and related methods. SAGE Publications; 1997.

4. Apéndices

```
require('kernlab')

kfunction <- function(linear = 0, quadratic = 0)
{
  k <- function (x,y)
  {
    linear*sum((x)*(y)) + quadratic*sum((x^2)*(y^2))
  }
  class(k) <- "kernel"
  k
}

n <- 25
a1 <- rnorm(n)
a2 <- 1 - a1 + 2* runif(n)
b1 <- rnorm(n)
b2 <- -1 - b1 - 2*runif(n)
x <- rbind(matrix(cbind(a1, a2), ncol = 2), matrix(cbind(b1, b2), ncol
    = 2))
y <- matrix(c(rep(1, n), rep(-1, n)))

svp <- ksvm(x,
            y,
            type = "C-svc",
            C = 100,
            kernel = kfunction(1, 0),
            scaled = c())

plot(range(x[, 1]),
     range(x[, 2]),
     type = 'n',
     xlab = expression(X[1]),
     ylab = expression(X[2]))
title(main = 'Caracteristicas separables lineales')

ymat <- ymatrix(svp)
points(x = x[-SVindex(svp), 1],
       y = x[-SVindex(svp), 2],
```

```
        pch = ifelse(ymat[-SVindex(svp)] < 0, 2, 1)) # 1: ícrculo 2: átringulo
points(x = x[SVindex(svp), 1],
        y = x[SVindex(svp), 2],
        pch = ifelse(ymat[SVindex(svp)] < 0, 17, 16)) # 16: ícrculo cerrado 17: átringulo cerrado

# Extraemos el vector w y b del modelo
w <- colSums(coef(svp)[[1]] * x[SVindex(svp), ])
b <- b(svp)

# Dibujamos las ílneas
abline(b/w[2], -w[1]/w[2])
abline((b + 1)/w[2], -w[1]/w[2], col = "gray")
abline((b - 1)/w[2], -w[1]/w[2], col = "gray")
```

Código 4.2: *Implementación de la función* `ksvm()` *en el paquete* `kernlab` *de* `R` *para representar las líneas de separación de clasificación.*

```
require('kernlab')

kfunction <- function(linear = 0, quadratic = 0)
{
  k <- function (x,y)
  {
    linear*sum((x)*(y)) + quadratic*sum((x^2)*(y^2))
  }
  class(k) <- "kernel"
  k
}

n <- 20
r <- runif(n)
a <- 2*pi*runif(n)
a1 <- r*sin(a)
a2 <- r*cos(a)
r <- 2 + runif(n)
a <- 2*pi*runif(n)
b1 <- r*sin(a)
b2 <- r*cos(a)
```

```
x <- rbind(matrix(cbind(a1, a2), ncol = 2), matrix(cbind(b1, b2), ncol
    = 2))
y <- matrix(c(rep(1, n), rep(-1, n)))

svp <- ksvm(x,
            y,
            type = "C-svc",
            C = 100,
            kernel = kfunction(0, 1),
            scaled = c())

par(mfrow = c(1, 2))
plot(range(x[, 1]),
     range(x[, 2]),
     type = 'n',
     xlab = expression(X[1]),
     ylab = expression(X[2]))

title(main = 'Espacio de caracteristicas')
ymat <- ymatrix(svp)
points(x = x[-SVindex(svp),1],
       y = x[-SVindex(svp),2],
       pch = ifelse(ymat[-SVindex(svp)] < 0, 2, 1))
points(x = x[SVindex(svp),1],
       y = x[SVindex(svp),2],
       pch = ifelse(ymat[SVindex(svp)] < 0, 17, 16))

# Extraemos el vector w y b del modelo
w2 <- colSums(coef(svp)[[1]] * x[SVindex(svp), ]^2)
b <- b(svp)

x1 <- seq(min(x[, 1]), max(x[, 1]), 0.01)
x2 <- seq(min(x[, 2]), max(x[, 2]), 0.01)

points(-sqrt((b-w2[1]*x2^2)/w2[2]), x2, pch = 16 , cex = .2 )
points(sqrt((b-w2[1]*x2^2)/w2[2]), x2, pch = 16 , cex = .2 )
points(x1, sqrt((b-w2[2]*x1^2)/w2[1]), pch = 16 , cex = .2 )
points(x1, -sqrt((b-w2[2]*x1^2)/w2[1]), pch = 16, cex = .2 )

points(-sqrt((1+ b-w2[1]*x2^2)/w2[2]) , x2, pch = 16 , cex = .2 , col =
    "gray")
```

```
61 points( sqrt((1 + b-w2[1]*x2^2)/w2[2]) , x2,  pch = 16 , cex = .2 , col
       = "gray")
62 points( x1 , sqrt(( 1 + b -w2[2]*x1^2)/w2[1]), pch = 16 , cex = .2 ,
      col = "gray")
63 points( x1 , -sqrt(( 1 + b -w2[2]*x1^2)/w2[1]), pch = 16, cex = .2 ,
      col = "gray")
64
65 points(-sqrt((-1+ b-w2[1]*x2^2)/w2[2]) , x2, pch = 16 , cex = .2 , col
      = "gray")
66 points( sqrt((-1 + b-w2[1]*x2^2)/w2[2]) , x2,  pch = 16 , cex = .2 ,
      col = "gray")
67 points( x1 , sqrt(( -1 + b -w2[2]*x1^2)/w2[1]), pch = 16 , cex = .2 ,
      col = "gray")
68 points( x1 , -sqrt(( -1 + b -w2[2]*x1^2)/w2[1]), pch = 16, cex = .2 ,
      col = "gray")
69
70 xsq <- x^2
71 svp <- ksvm(x = xsq,
72             y = y,
73             type = "C-svc",
74             C = 100,
75             kernel = kfunction(1, 0),
76             scaled = c())
77
78 plot(x = range(xsq[, 1]),
79      y = range(xsq[, 2]),
80      type = 'n',
81      xlab = expression(X[1]^2),
82      ylab = expression(X[2]^2))
83
84 title(main='Espacio cuadratico')
85 ymat <- ymatrix(svp)
86 points(x = xsq[-SVindex(svp), 1],
87        y = xsq[-SVindex(svp), 2],
88        pch = ifelse(ymat[-SVindex(svp)] < 0, 2, 1))
89 points(x = xsq[SVindex(svp), 1],
90        y = xsq[SVindex(svp), 2],
91        pch = ifelse(ymat[SVindex(svp)] < 0, 17, 16))
92
93 # Extraemos el vector w y b del modelo
94 w <- colSums(coef(svp)[[1]] * xsq[SVindex(svp),])
```

```
b <- b(svp)

# Dibujamos las ílneas
abline(b/w[2], -w[1]/w[2])
abline((b + 1)/w[2], -w[1]/w[2], col = "gray")
abline((b - 1)/w[2], -w[1]/w[2], col = "gray")
```

Código 4.3: *Implementación de la función* `ksvm()` *en el paquete* `kernlab` *de* `R` *para representar las líneas de separación de clasificación.*

```
mipc <- prcomp(entrenamiento[, 1:256])
mipch <- c(21:25,21:25)
plot(predict(mipc)[, 1:2],
     xlim = c(-10, 10),
     ylim = c(-10, 10),
     pch = rep(mipch, each = 65),
     col = gray(0.1), #rep(rainbow(10), each = 65),
     bg = rep(rainbow(10), each = 65),
     xaxt = "n", yaxt = "n",
     xlab = "Componente Principal 1",
     ylab = "Componente Principal 2")
axis(1)
axis(2, las = 2)
abline(v = pretty(c(-10, 10)), lty = 3, col = "gray")
abline(h = pretty(c(-10, 10)), lty = 3, col = "gray")

legend(x = "topright", legend = seq(0, 9), pch = c(21:25,21:25),
       col = gray(0.1), #rainbow(10),
       pt.bg = rainbow(10), bty = "n",
       ncol = 5,cex = 0.7,
       title = "ídgito")
```

Código 4.4: *Código para realizar un análisis de componentes principales considerando el dígito que representa cada observación.*

```
X <- entrenamiento[, 1:256]
Y <- entrenamiento$digito
modelo <- train(x = X,
                y = Y,
                method = "svmLinear2")
```

Código 4.5: *Código para construir una máquina de vector soporte con kernel lineal.*

```
X <- entrenamiento[, 1:256]
Y <- entrenamiento$digito

modelo <- train(x = X,
                y = Y,
                preProcess = c("scale", "center"),
                trControl = trainControl(method = "cv",
                                         p = 0.8,
                                         number = 5,
                                         repeats = 5),
                method = "svmLinear2",
                tuneLength = 10,
                maximize = TRUE,
                metric = "Accuracy")
```

Código 4.6: *Código para construir una máquina de vector soporte con kernel lineal utilizando diferentes argumentos de la función train().*

```
suppressPackageStartupMessages(library("extrafont")) # listado de
    fuentes del sistema
# font_import() # extrafont
# fonts() # extrafont
# fonttable() # extrafont

fuentes <- fonts() # extrafont

for (i in 1:length(fonts())) {
  for (j in 0:9) {
    filename <- paste("/numeros/", j, "/", fuentes[i], ".png", sep = ""
    )
    png(filename = filename, width = 30, height = 30)
    par(mfrow = c(1, 1), mar = c(0,0,0,0))
    plot(-1, -1, xlim = c(0, 1), ylim = c(0, 1), axes = FALSE)
    text(x = 0.5, y = 0.5, family = fuentes[i], j, cex = 3)
    dev.off()
  }
```

```
17 }
```

Código 4.7: *Secuencia de comandos para listar todas las fuentes instaladas en el sistema y generar una imagen de cada dígito (0-9) con cada tipografía. La secuencia de comandos utiliza el paquete extrafont que es específico de Linux, en Windows se debe utilizar la función windowFonts() del paquete grDevices.*