

Data Science con R

Instituto de Estadística PUCV - Magister en Estadística

Dae-Jin Lee < dlee@bcamath.org >

```
install.packages("e1071")
install.packages("ROCR")
```

Máquinas de Vector Soporte (Support Vector Machines) en R

Existen numerosas librerías en R para implementar las SVM (ver)

En este capítulo utilizaremos `e1071` y `caret`.

Support Vector Machines (SVM) ó máquinas de vector soporte son un conjunto de algoritmos de aprendizaje supervisado relacionados directamente con problemas de clasificación y regresión. Dado un conjunto de entrenamiento podemos etiquetar las clases y entrenar una SVM para construir un modelo que prediga la clase de una nueva muestra. Las SVM intuitivamente separa las muestras en el espacio mediante hiperplanos de separación definidos como el vector entre los dos puntos, de las dos clases, más cercanos se denominan **vectores soporte**.

Más formalmente, una SVM construye un hiperplano o conjunto de hiperplanos en un espacio de dimensionalidad muy alta (o incluso infinita) que puede ser utilizado en problemas de clasificación o regresión. Una buena separación entre las clases permitirá una clasificación correcta. La técnica SVM es generalmente útil para datos que no tienen regularidad, es decir, datos cuya distribución es desconocida.

En este algoritmo, trazamos cada elemento de datos como un punto en el espacio n-dimensional (donde n es el número de características/variables) con el valor de cada característica que es el valor de una coordenada particular. Luego, realizamos la clasificación encontrando el hiperplano que diferencia muy bien las dos clases.

Los **vectores soporte** son simplemente las coordenadas de la observación individual. La SVM es una frontera que mejor separa las dos clases (hiperplano/línea).

¿Cómo funcionan las SVM?

La pregunta es *¿Cómo podemos identificar el hiperplano correcto?*

- Identificar el hiperplano correcto (**Escenario 1**): Aquí tenemos tres hiperplanos (A, B y C). Ahora, identifica el hiperplano derecho para clasificar estrella y círculo.

Recordemos que el criterio consiste en “Seleccionar el hiperplano que mejor separa las dos clases”. En este escenario, el hiperplano “B” ha realizado este trabajo de manera excelente.

- Identificar el hiperplano correcto (**Escenario 2**): Aquí tenemos tres hiperplanos (A, B y C) y todos están segregando bien las clases. ¿Cómo podemos identificar el hiperplano correcto?

Aquí, maximizar las distancias entre el punto de datos más cercano (cualquier clase) y el hiperplano nos ayudará a decidir el hiperplano correcto. Esta distancia se llama Margen.

El margen para el hiperplano C es alto en comparación con A y B. Por lo tanto, el hiperplano correcto es el C. Otra razón para seleccionar el hiperplano con mayor margen es la robustez. Si seleccionamos un hiperplano con un margen bajo, existe una alta probabilidad de que se produzca una clasificación errónea.

- Identificar el hiperplano correcto (**Escenario 3**):

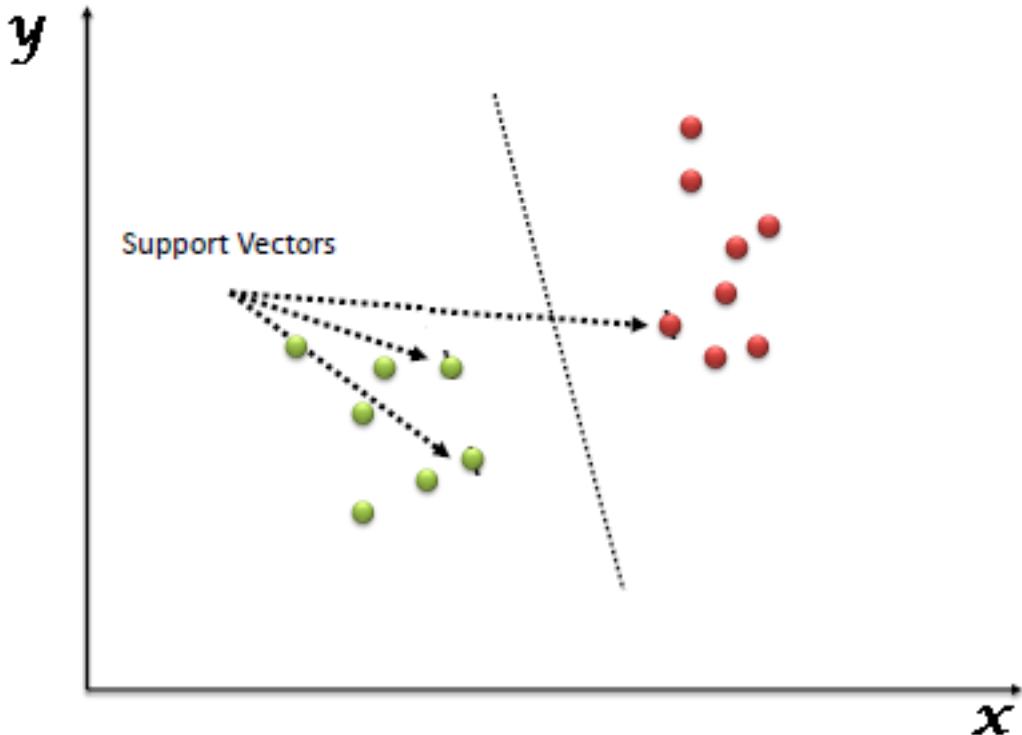


Figure 1: Ilustración de vectores soporte

$\circ A$ ó B ?

SVM selecciona el hiperplano que clasifica las clases con precisión antes de maximizar el margen. Aquí, el hiperplano B tiene un error de clasificación y A ha clasificado todo correctamente. Por lo tanto, el hiperplano derecho es A.

- ¿Podemos clasificar dos clases (**Escenario 4**)? Abajo, no podemos separar las dos clases usando una línea recta, ya que una de las estrellas se encuentra en el territorio de otra clase (círculo) como un *outlier*.

Como ya se ha mencionado, una estrella azul en el otro extremo es como un atípico (outlier) para la clase de estrellas. SVM tiene una característica para ignorar los valores atípicos y encontrar el hiperplano que tiene el máximo margen. Por lo tanto, podemos decir, SVM es robusto a los valores atípicos.

- Encuentra el hiperplano para separar las clases (**Escenario 5**): En el escenario de abajo, no podemos tener un hiperplano lineal entre las dos clases, así que ¿cómo clasifica el SVM estas dos clases? Hasta ahora, sólo hemos mirado el hiperplano lineal.

Las SVM pueden resolver este problema introduciendo una característica adicional. Aquí, añadiremos una nueva característica $z = x^2 + y^2$. Ahora, grafiquemos los puntos de datos en los ejes x y z:

En la gráfica anterior, los puntos a considerar son:

- Todos los valores de z serían siempre positivos porque z es la suma al cuadrado de x e y .
- En el gráfico original, los círculos rojos aparecen cerca del origen de los ejes x e y , lo que lleva a un valor más bajo de z y a una estrella relativamente alejada del resultado de origen a un valor más alto de z .

En las SVM, es fácil tener un hiperplano lineal entre estas dos clases. Pero, la pregunta que surge es, si tenemos que añadir esta característica manualmente para tener un hiper-plano.

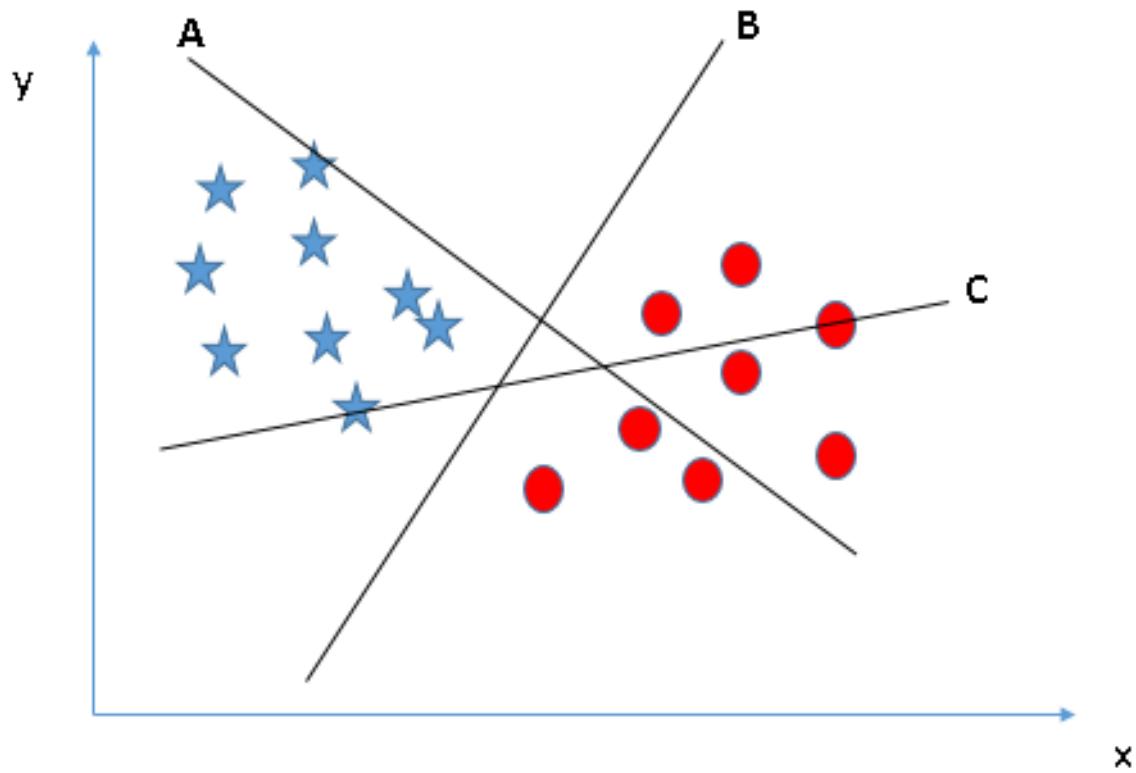


Figure 2: Escenario 1

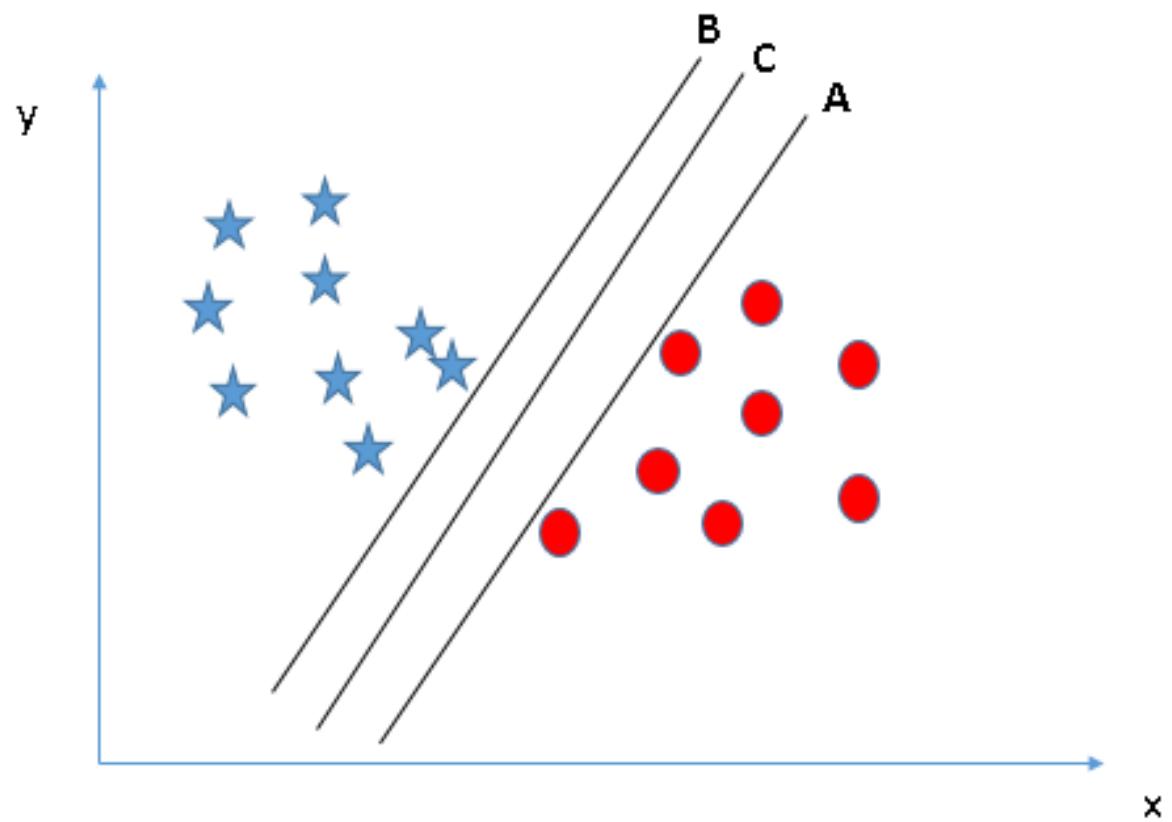


Figure 3: Escenario 2

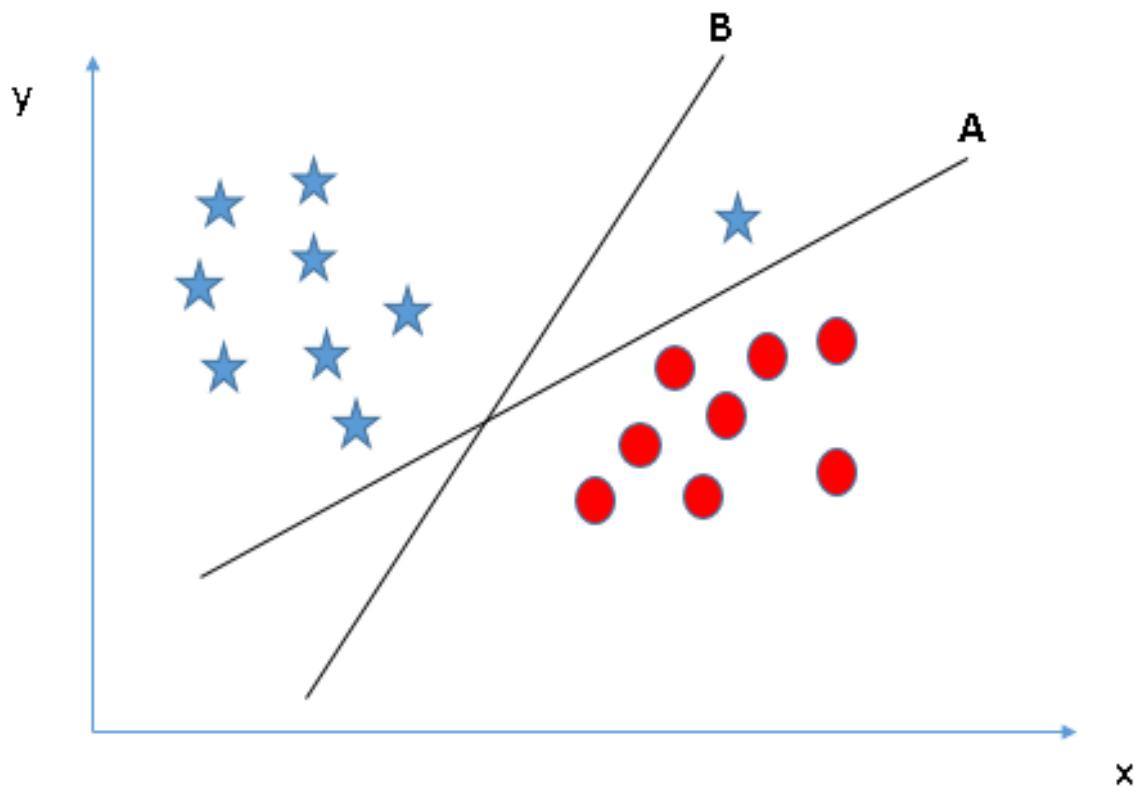


Figure 4: Escenario 3

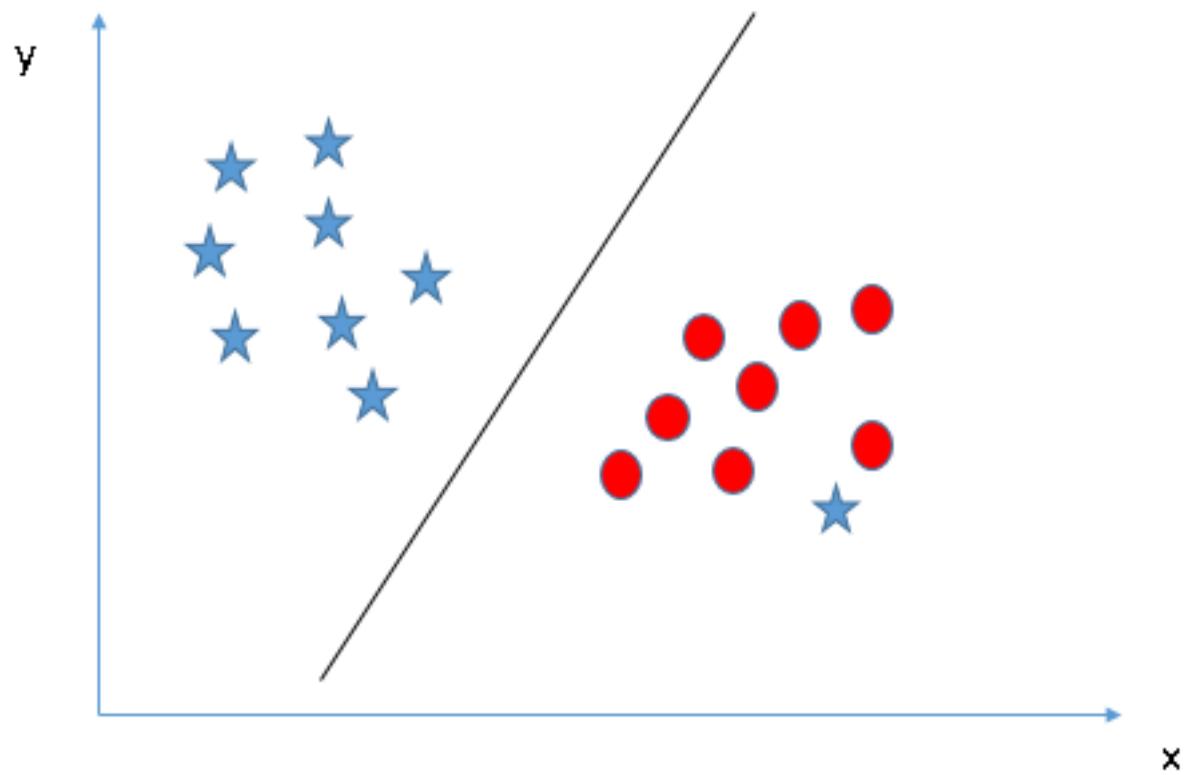


Figure 5: *Escenario 4*

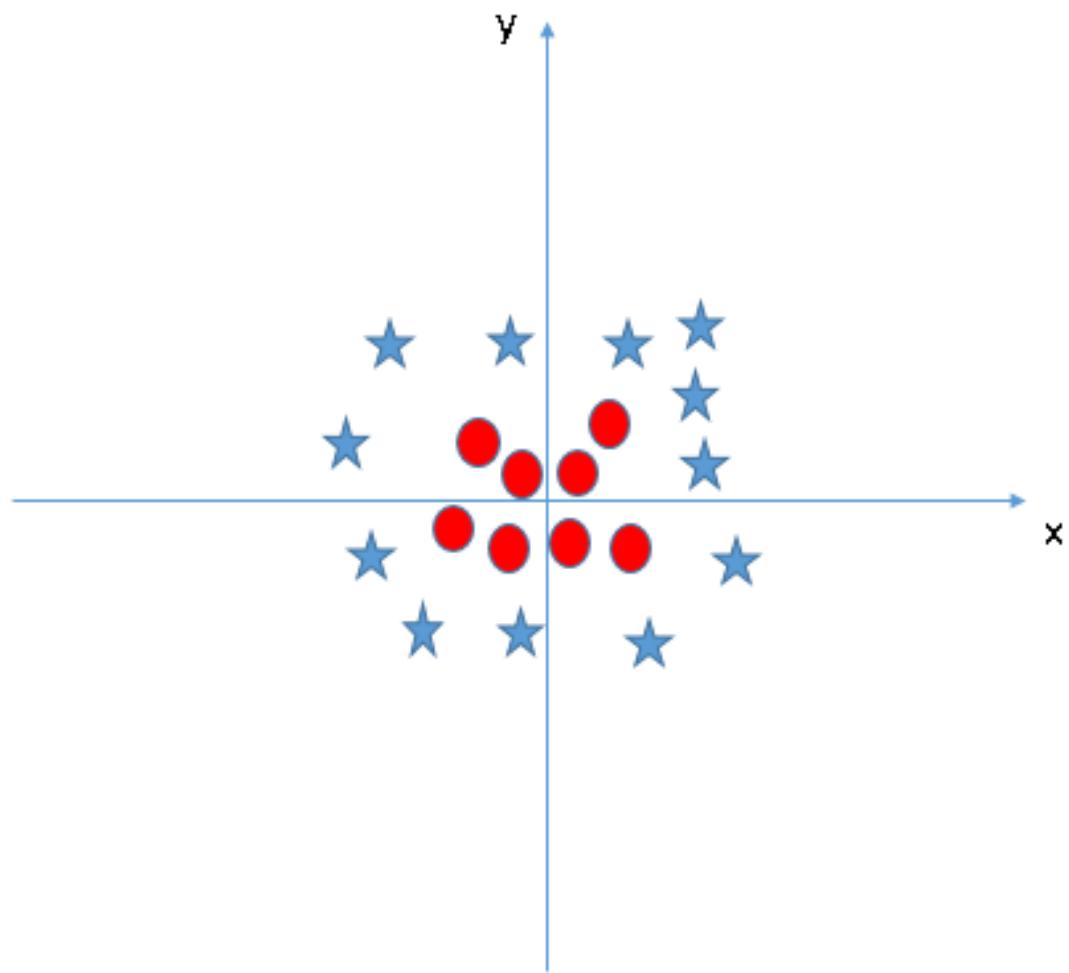


Figure 6: *Escenario 5*

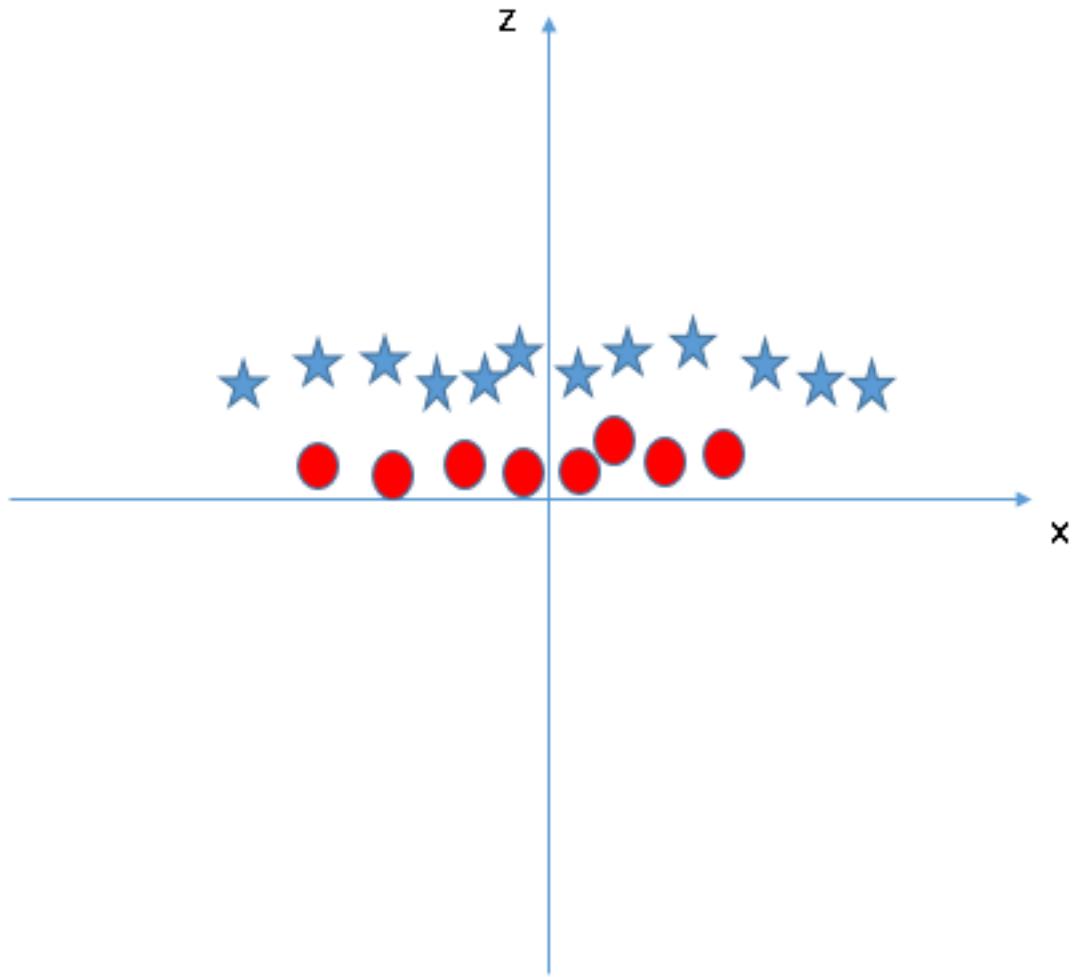


Figure 7: Escenario 6

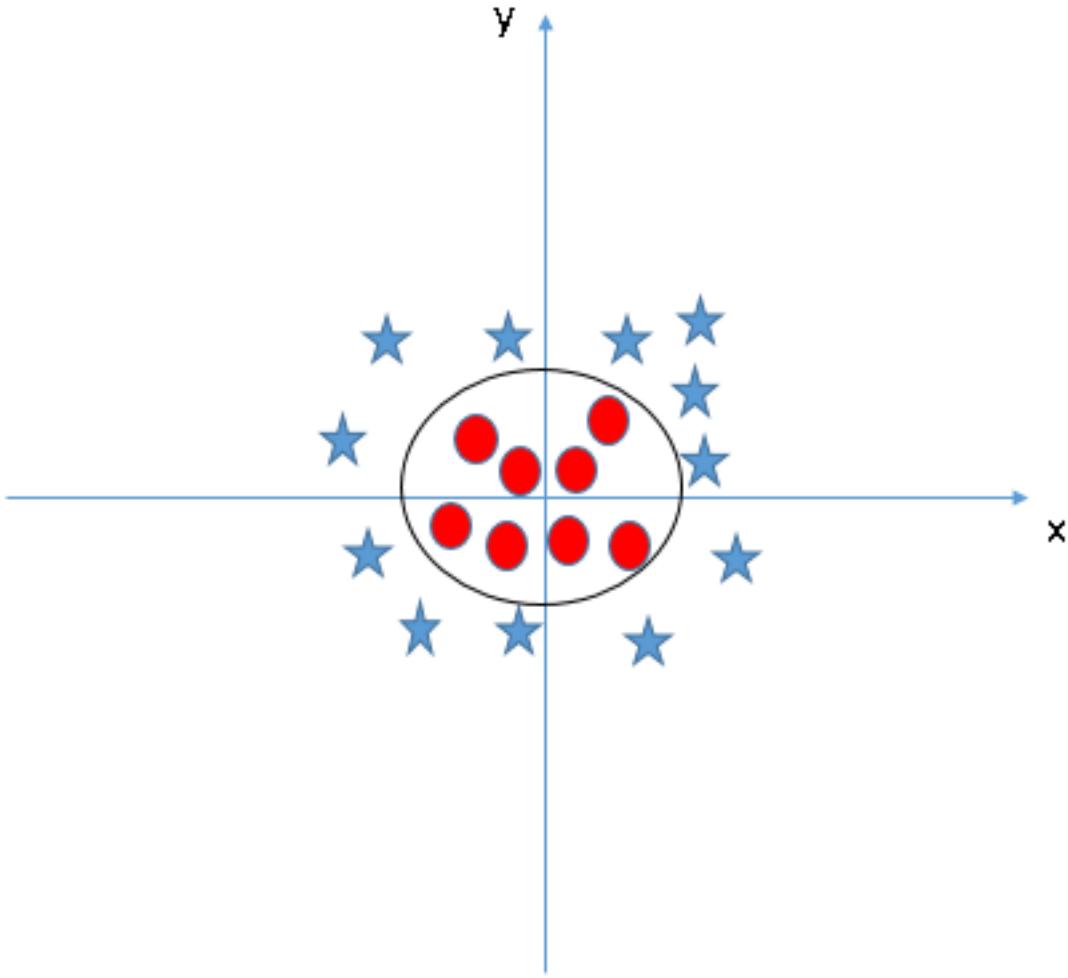


Figure 8: Escenario 6, hiperplano en el espacio de entrada original

La respuesta es **No**, las SVM tienen una técnica llamada el truco del kernel (*kernel trick*). Estas son funciones que toman un espacio de entrada de baja dimensión y lo transforman en un espacio dimensional superior, es decir, convierte un problema no separable en un problema separable, estas funciones se denominan **kernel** ó núcleos. Es principalmente útil en problemas de separación no lineal. En pocas palabras, realiza algunas transformaciones de datos extremadamente complejas, y luego descubre el proceso para separar los datos basándose en las etiquetas o salidas que has definido.

Cuando observamos el hiperplano en el espacio de entrada original, parece un círculo:

Los modelos basados en SVMs están estrechamente relacionados con las redes neuronales. Usando una función kernel, resultan un método de entrenamiento alternativo para clasificadores polinomiales, funciones de base radial y perceptrón multicapa.

Support vector regression

Support Vector Machine también puede ser utilizado como un método de regresión, manteniendo todas las características principales que caracterizan el algoritmo (margen máximo). La Support Vector Regression

(SVR) utiliza los mismos principios que la SVM para la clasificación, con sólo unas pocas diferencias menores. En primer lugar, debido a que la salida es un número real, se hace muy difícil predecir la información a mano, que tiene infinitas posibilidades. En el caso de regresión, se establece un margen de tolerancia (ϵ) en aproximación al SVM que ya habría solicitado del problema. Pero además de este hecho, también hay una razón más complicada, el algoritmo es más complicado por lo tanto a tener en cuenta. Sin embargo, la idea principal es siempre la misma: minimizar el error, individualizando el hiperplano que maximiza el margen, teniendo en cuenta que parte del error es tolerado.

Al igual que en el caso de la SVM, se utiliza el kernel trick (algunos tipos de kernel son el polinomial o Gaussian Radial Basis Functions).

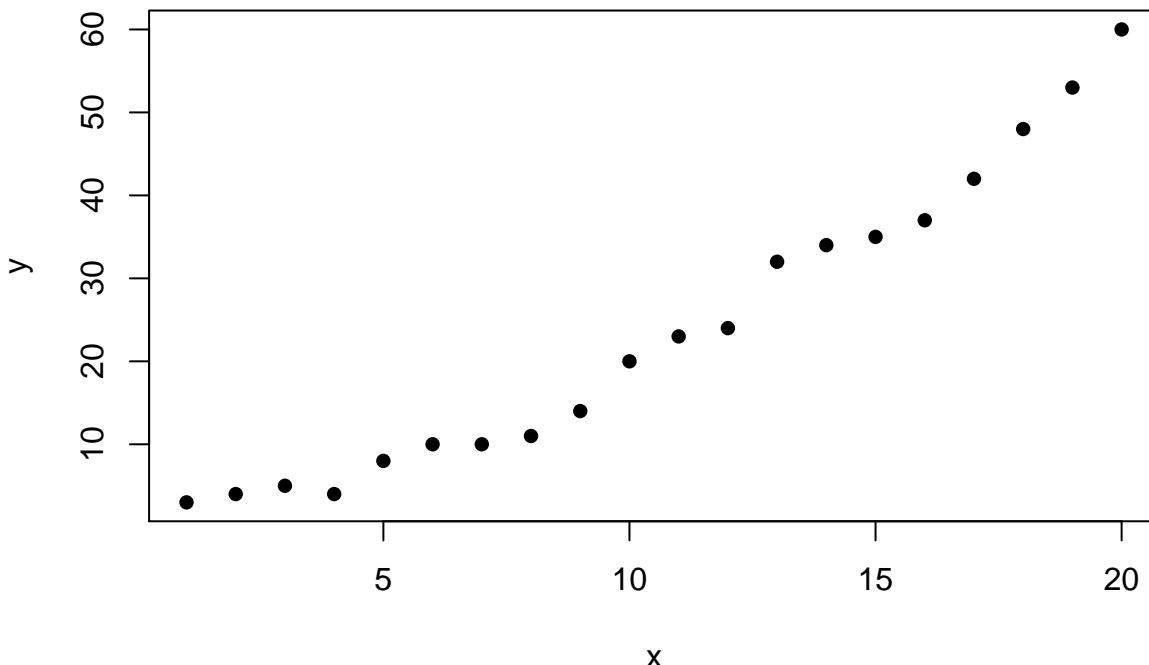
Ejemplo:

```
x=c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
y=c(3,4,5,4,8,10,10,11,14,20,23,24,32,34,35,37,42,48,53,60)

# creamos un data.frame
train=data.frame(x,y)
```

Veamos cómo se ven nuestros datos. Para ello utilizamos la función `plot`.

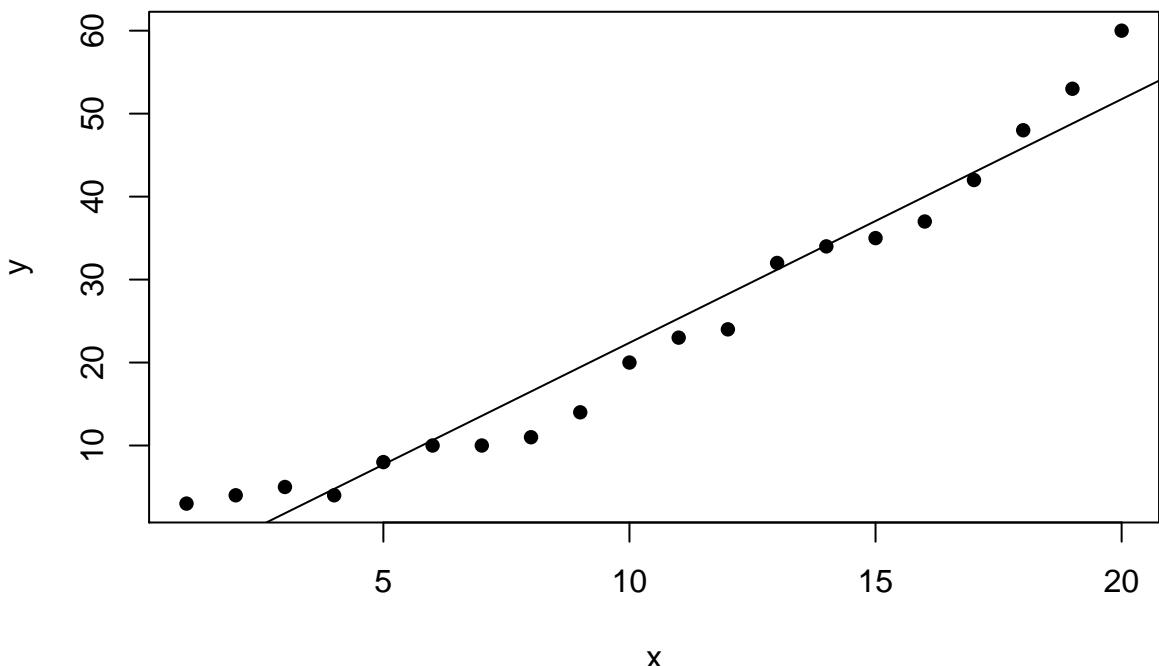
```
plot(train,pch=16)
```



Mirando la gráfica, parece que una regresión lineal también podría ajustar bien en los datos. Usaremos ambos modelos y los compararemos. Primero, el código para la regresión lineal:

```
# Regresión lineal
model <- lm(y ~ x, train)

# abline
plot(train,pch=16)
abline(model)
```



SVM en R con la librería e1071:

```
# SVM

library(e1071)

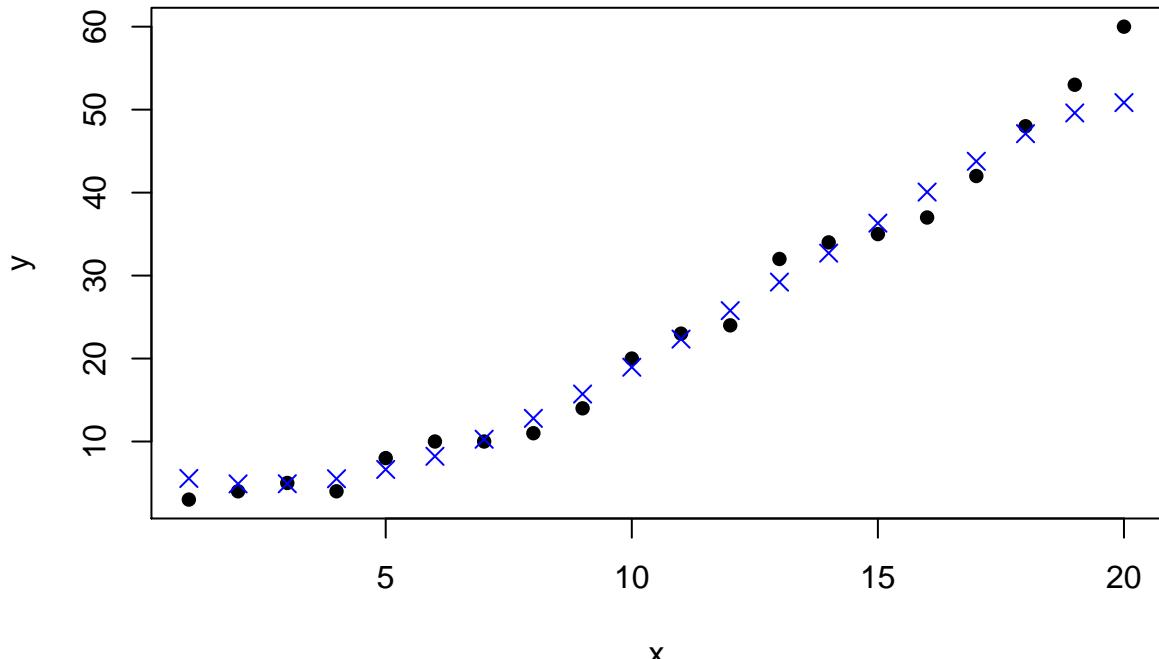
# ajuste SVM

model_svm <- svm(y ~ x , train)

# predict

pred <- predict(model_svm, train)

# Plot
plot(train,pch=16)
points(train$x, pred, col = "blue", pch=4,cex=1.2)
```



Los puntos siguen los valores reales mucho más de cerca que el modelo lineal. ¿Podemos verificar eso? Calculemos los errores RSME para ambos modelos:

```
# residuos del modelo lineal

error <- model$residuals

lm_error <- sqrt(mean(error^2))
lm_error

## [1] 3.832974

# para sum, hay que calcularlo a mano

error_2 <- train$y - pred

svm_error <- sqrt(mean(error_2^2))
svm_error

## [1] 2.696281
```

En este caso, el `rmse` para el modelo lineal es ≈ 3.83 mientras que nuestro modelo svm tiene un error menor de 2.7.

Una implementación directa de SVM tiene una precisión mayor que el modelo de regresión lineal.

Sin embargo, el modelo SVM va mucho más allá. Podemos mejorar aún más nuestro modelo SVM y ajustarlo para que el error sea aún menor. Ahora profundizaremos en la función `svm` y la función de ajuste (*tuning*). Podemos especificar los valores para el parámetro de coste y epsilon que es 0.1 por defecto.

Una manera sencilla es intentar para cada valor de epsilon entre 0 y 1 (tomaré pasos de 0.01) y de manera similar intentar para la función de coste de 4 a 2^9 (por ejemplo pasos exponenciales de 2 aquí). Tomando 101 valores de epsilon y 8 valores de función de coste. Por lo tanto, vamos a probar los modelos 808 y ver cuál de ellos funciona mejor. El código puede tardar un poco en ejecutar todos los modelos y encontrar la mejor versión. El código correspondiente será

```

svm_tune <- tune(svm, y ~ x, data = train,
                  ranges = list(epsilon = seq(0.1, 0.01), cost = 2^(2:9)))
)
print(svm_tune)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   epsilon cost
##       0     8
##
## - best performance: 2.29037

```

Una ventaja del ajuste en R es que nos permite extraer la mejor función directamente. No tenemos que hacer nada y simplemente extraer la mejor función de la lista `svm_tune`. Ahora podemos ver la mejora en nuestro modelo calculando su error RMSE utilizando el siguiente código:

```

# Mejor modelo
best_mod <- svm_tune$best.model
best_mod_pred <- predict(best_mod, train)

error_best_mod <- train$y - best_mod_pred

# este valor puede ser diferente en cada caso
# porque el método tuneado baraja los datos al azar
best_mod_RMSE <- sqrt(mean(error_best_mod^2))
best_mod_RMSE

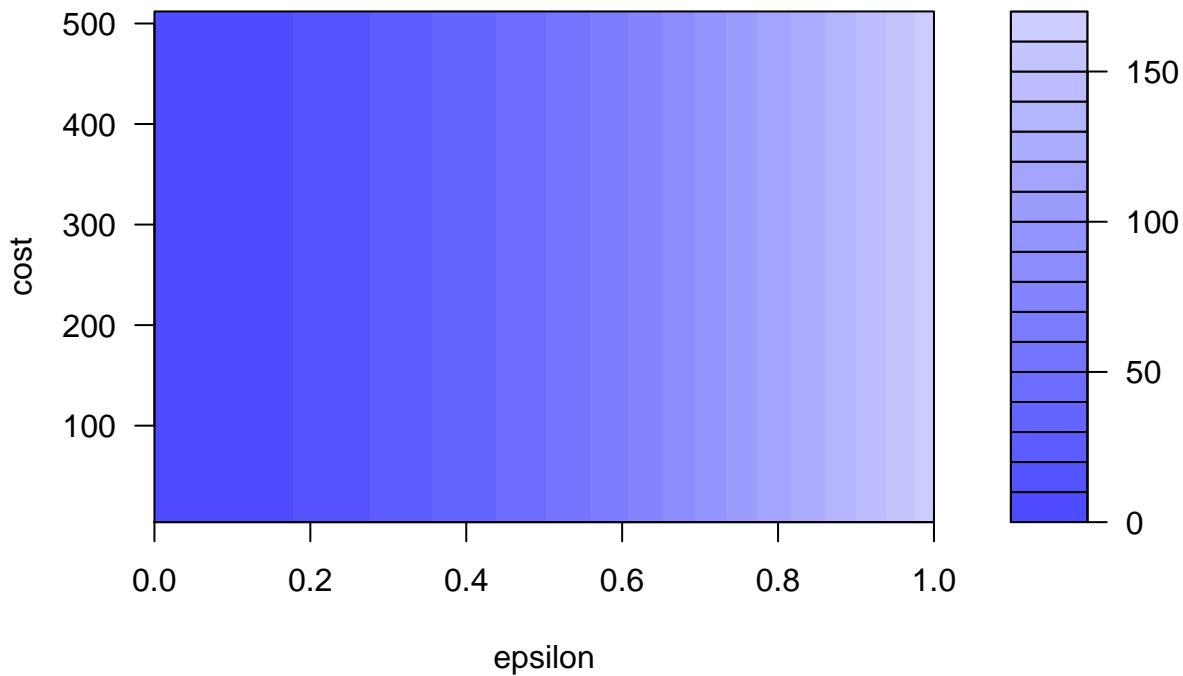
## [1] 1.159186

```

Este método de tuneado (ajuste) se conoce como búsqueda de cuadrícula (grid search). R ejecuta todos los modelos con todos los valores posibles de epsilon y la función de coste en el rango especificado y nos da el modelo que tiene el error más bajo.

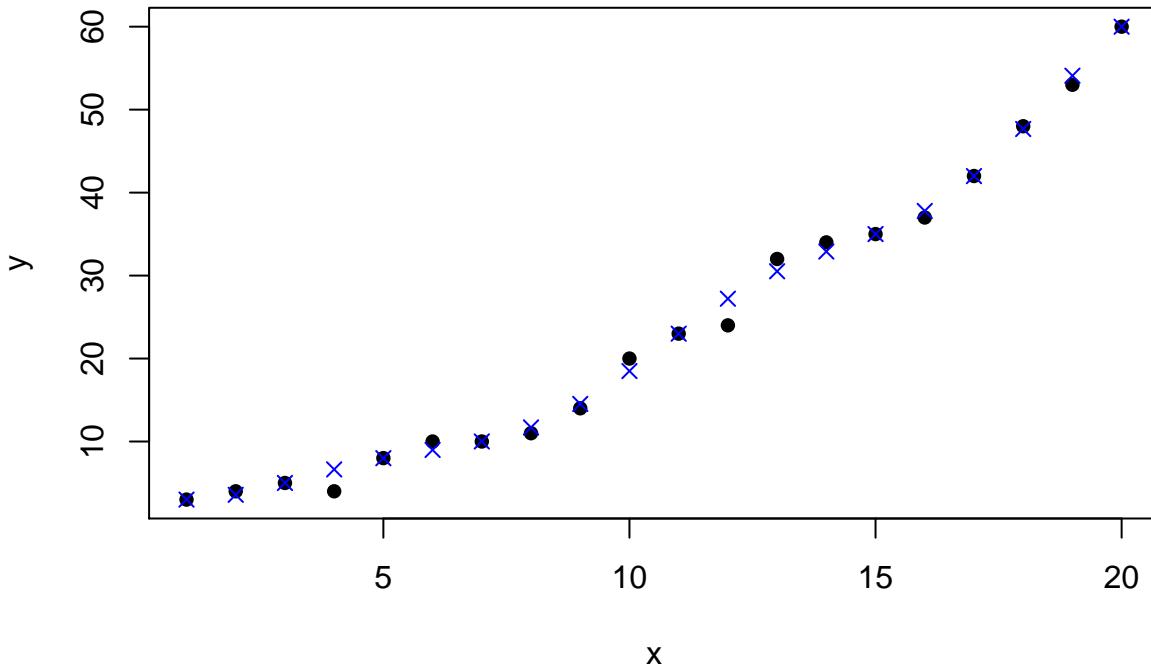
```
plot(svm_tune)
```

Performance of 'svm'



Este gráfico muestra el resultado de varios modelos usando códigos de colores. Las regiones más oscuras implican una mayor precisión. El uso de este gráfico es para determinar el posible rango en el que podemos reducir nuestra búsqueda e intentar afinarla más si es necesario. Por ejemplo, este gráfico muestra que puedo ejecutar el tuneado para epsilon en el nuevo rango de 0 a 0,2 y mientras estoy en ello, puedo moverme en pasos aún más bajos (digamos 0,002), pero ir más allá puede llevar a un sobreajuste, por lo que puedo detenerme en este punto. A partir de este modelo, hemos mejorado nuestro error inicial de 2,69 y nos acercamos a 1,29, que es aproximadamente la mitad de nuestro error original en SVM.

```
plot(train,pch=16)
points(train$x, best_mod_pred, col = "blue", pch=4)
```



Visualmente, los puntos predichos por nuestro modelo afinado casi siguen los datos. Este es el poder del SVM y lo estamos viendo para datos con dos características.

El SVM es una técnica poderosa y especialmente útil para datos cuya distribución es desconocida (también conocida como no regularidad en los datos). Debido a que el ejemplo considerado aquí consistió en sólo dos características, el SVM instalado por R aquí también se conoce como SVM lineal. El SVM está alimentado por un kernel para tratar varios tipos de datos y su kernel también puede ser configurado durante el ajuste del modelo. Algunos de estos ejemplos incluyen el gaussiano y el radial. Por lo tanto, el SVM también puede ser utilizado para datos no lineales y no requiere ninguna suposición sobre su forma funcional. Debido a que separamos los datos con el máximo margen posible, el modelo se vuelve muy robusto y puede manejar incongruencias como datos de pruebas ruidosas o datos sesgados del tren. También podemos interpretar los resultados producidos por SVM a través de la visualización. Una desventaja común con el SVM está asociada con su tuneado.

El nivel de precisión en la predicción de los datos de entrenamiento debe ser definido en nuestros datos. Debido a que nuestro ejemplo eran datos generados a medida, seguimos adelante e intentamos conseguir que la precisión de nuestro modelo fuera lo más alta posible reduciendo el error.

Sin embargo, en situaciones reales en las que se necesita entrenar el modelo y predecir continuamente sobre los datos de prueba, el SVM puede caer en la trampa del sobreajuste. Esta es la razón por la cual el SVM necesita ser modelado cuidadosamente - de lo contrario la precisión del modelo puede no ser satisfactoria.

Como se ilustra en el ejemplo, la técnica SVM está estrechamente relacionada con la técnica de regresión. Para los datos lineales, podemos comparar el SVM con la regresión lineal, mientras que el SVM no lineal es comparable con la regresión logística. A medida que los datos se vuelven más y más lineales, la regresión lineal se vuelve más y más precisa. Al mismo tiempo, el SVM tuneado también puede ajustar los datos. Sin embargo, el ruido y el sesgo pueden afectar severamente la capacidad de regresión.

Parámetros de control de la SVM

Cuando clasificamos datos con SVM es necesario fijar un margen de separación entre observaciones, si no fijamos este margen nuestro modelo sería tan bueno tan bueno que sólo serviría para esos datos, estaría sobreestimando y eso es malo. El coste C y el gamma son los dos parámetros con los que contamos en los

SVM. El parámetro C es el peso que le damos a cada observación a la hora de clasificar un mayor coste implicaría un mayor peso de una observación y el SVM sería más estricto.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.2
```

```
library(e1071)
```

```
#Datos iniciales
```

```
long = 20000
```

```
x <- runif(long,1,100)
```

```
y <- runif(long,1,100)
```

```
datos <- data.frame(x,y)
```

```
#La mitad entrenamiento la mitad test
```

```
indices <- sample(1:long,long/2)
```

```
entrenamiento <- datos[indices,]
```

```
test <- datos[-indices,]
```

```
#letra O
```

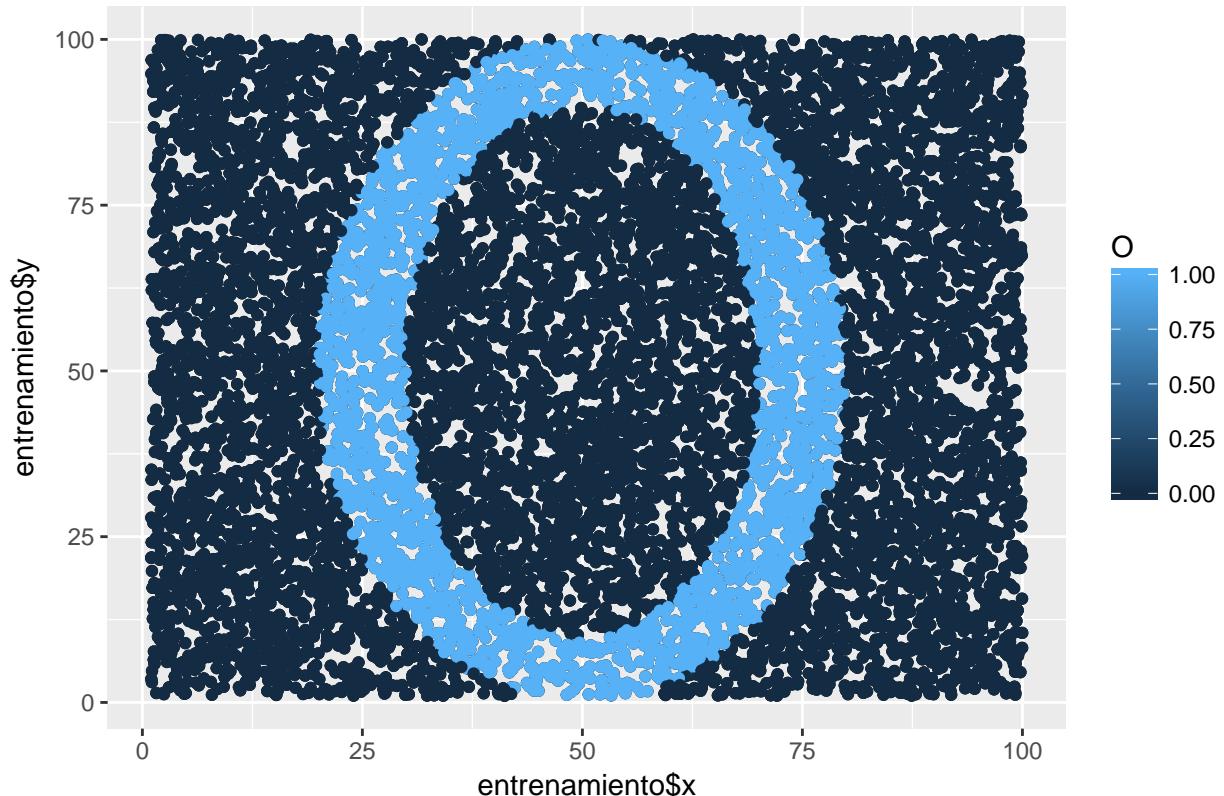
```
O <- ifelse((entrenamiento$x-50)^2/20^2 + (entrenamiento$y-50)^2/40^2 >1 , 1,0)
```

```
O <- ifelse((entrenamiento$x-50)^2/30^2 + (entrenamiento$y-50)^2/50^2 >1 , 0,0)
```

```
g.train <- ggplot(entrenamiento,aes(entrenamiento$x,entrenamiento$y)) + geom_point()
```

```
g.train + geom_point(aes(colour = O)) + labs(title="DATOS DE ENTRENAMIENTO PARA LA LETRA O")
```

DATOS DE ENTRENAMIENTO PARA LA LETRA O

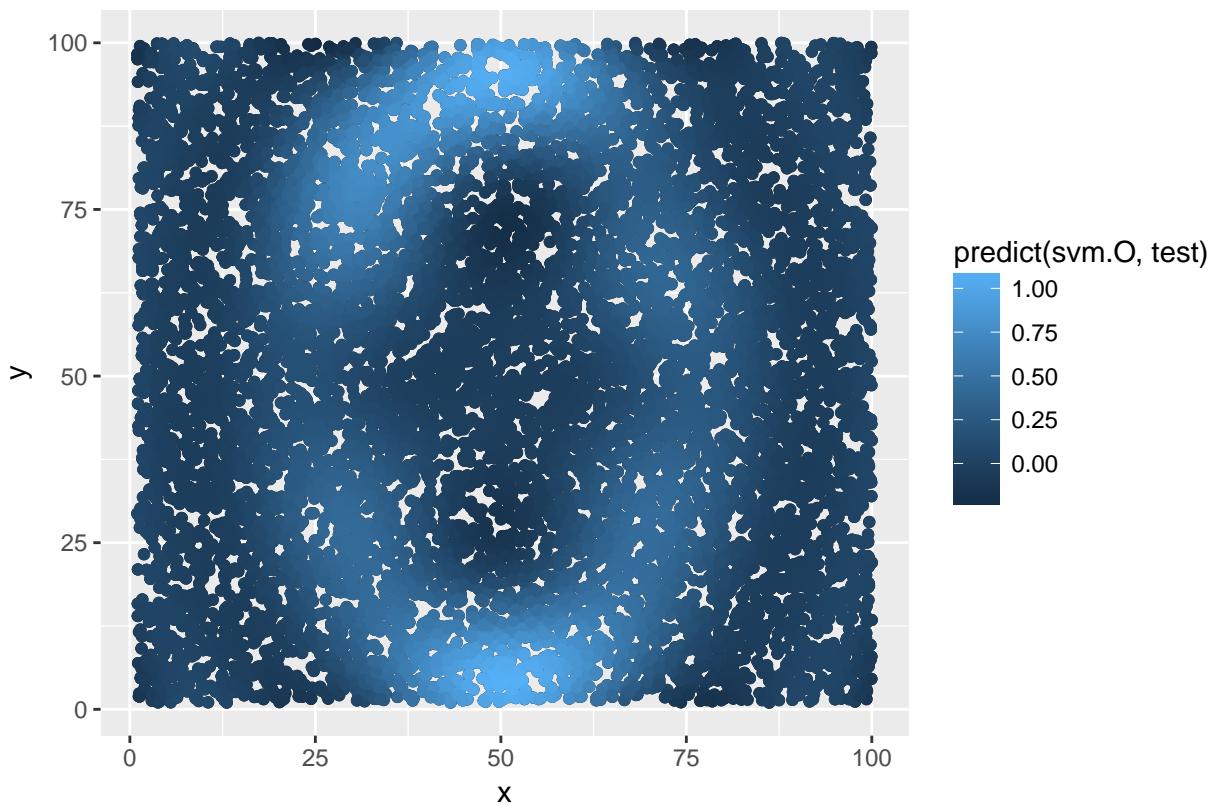


Supongamos un modelo para clasificar observaciones en el plano como una letra **O** podemos ver como se modifica la estimación en esta secuencia en la que se ha modificado el parámetro C :

```
#Gráfico de test
g.test <- ggplot(test,aes(x, y)) + geom_point()

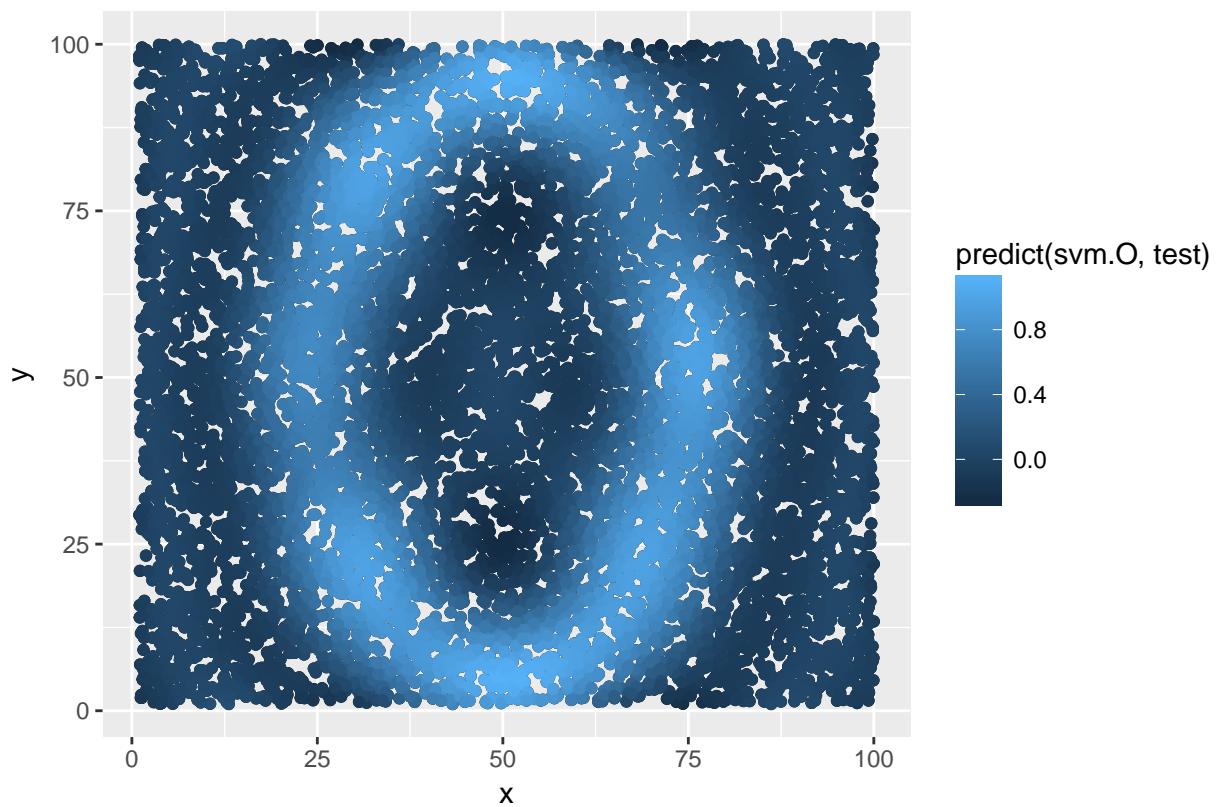
svm.0=svm(0 ~ x + y,entrenamiento,method="C-classification",
          kernel="radial",cost=10,gamma=1)
g.test + geom_point(aes(colour = predict(svm.0,test))) +
  labs(title="SVM LETRA O Coste 10 Gamma 1")
```

SVM LETRA O Coste 10 Gamma 1



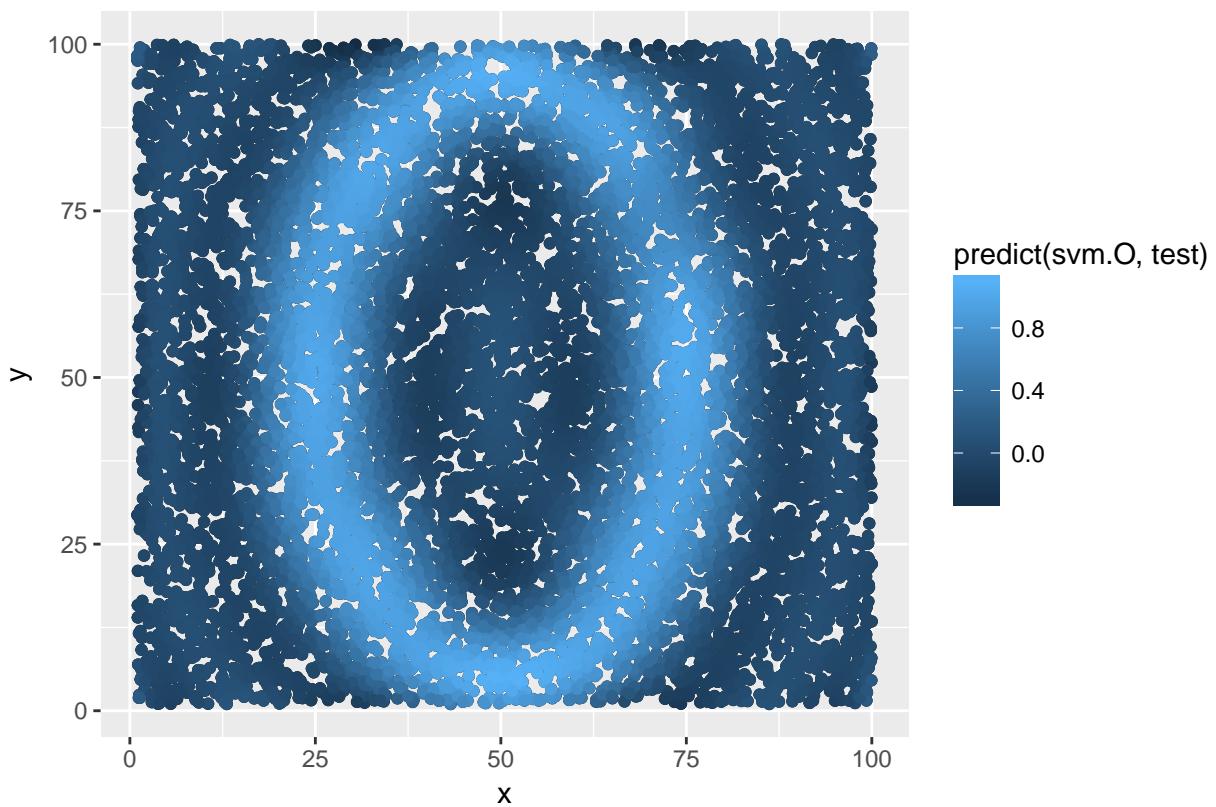
```
svm.0=svm(0 ~ x + y,entrenamiento,method="C-classification",
          kernel="radial",cost=100,gamma=1)
g.test + geom_point(aes(colour = predict(svm.0,test))) +
  labs(title="SVM LETRA O Coste 100 Gamma 1")
```

SVM LETRA O Coste 100 Gamma 1



```
svm.O=svm(0 ~ x + y,entrenamiento,method="C-classification",
          kernel="radial",cost=1000,gamma=1)
g.test + geom_point(aes(colour = predict(svm.O,test))) +
  labs(title="SVM LETRA O Coste 1000 Gamma 1")
```

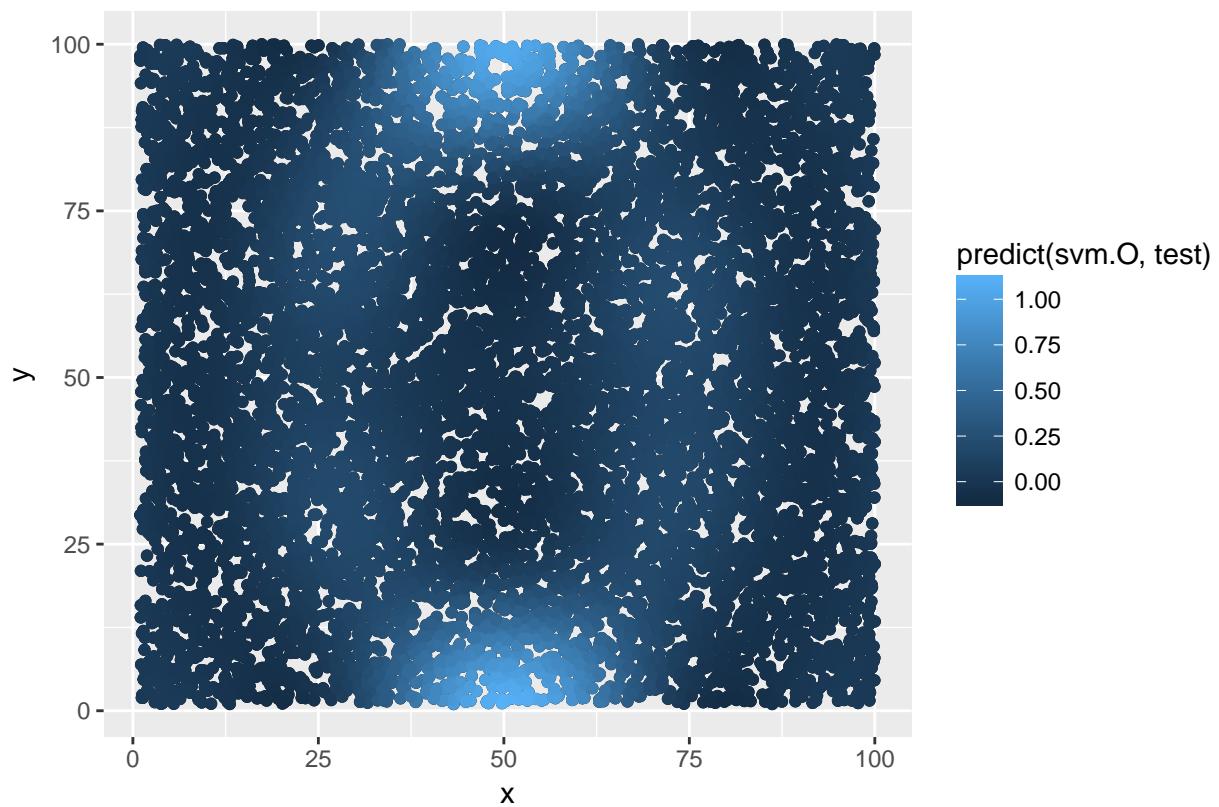
SVM LETRA O Coste 1000 Gamma 1



Se puede ver como la predicción es más conservadora cuando ponemos costes más bajos, pero no es el único parámetro que tenemos para suavizar la sobreestimación también tenemos el parámetro gamma que le da otra vuelta de tuerca a la relación entre las observaciones ya no es una relación entre puntos del espacio, ahora esta relación es una función kernel, lo que nos facilita encontrar los subespacios que puedan diferenciar los puntos en el espacio y también nos permite añadir mayor complejidad a la hora de separar observaciones. Veamos el ejemplo anterior pero modificando el parámetro Gamma y dejando fijo el C:

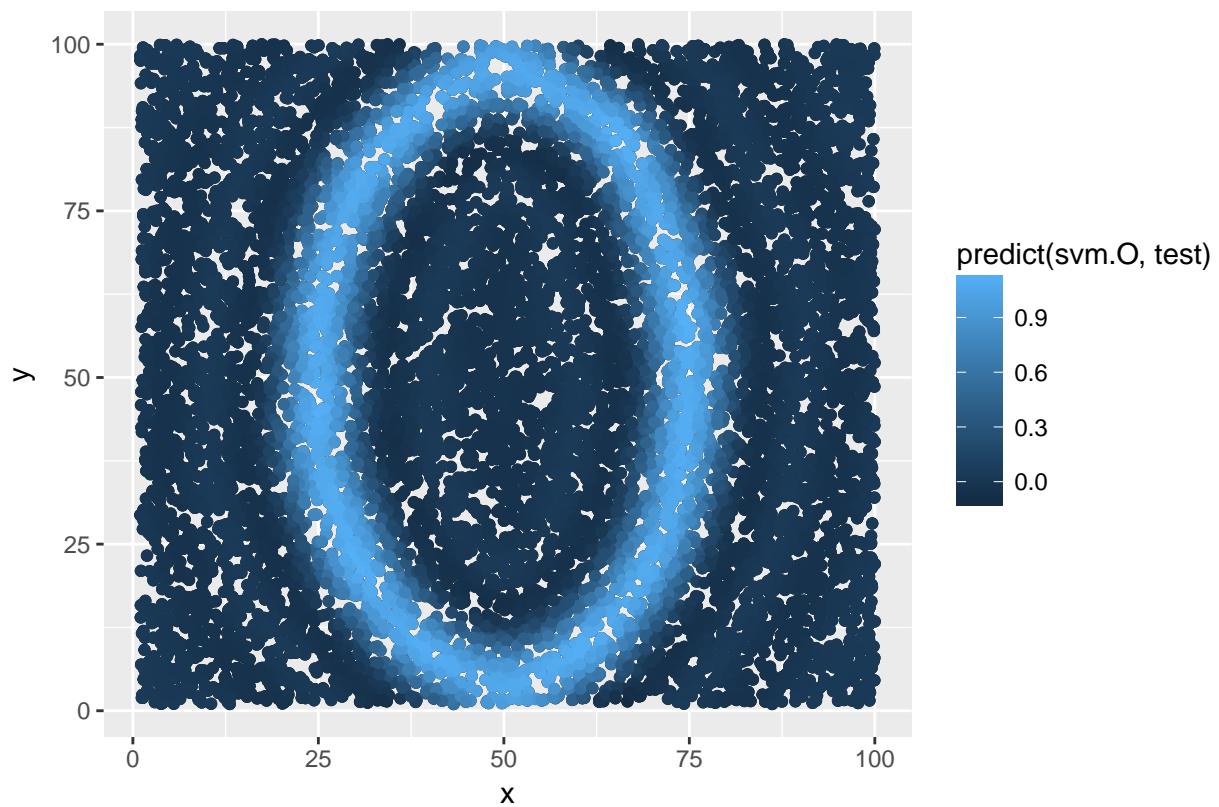
```
#####
svm.O=svm(O ~ x + y,entrenamiento,method="C-classification",
           kernel="radial",cost=1,gamma=1)
g.test + geom_point(aes(colour = predict(svm.O,test))) +
  labs(title="SVM LETRA O Coste 1 Gamma 1")
```

SVM LETRA O Coste 1 Gamma 1



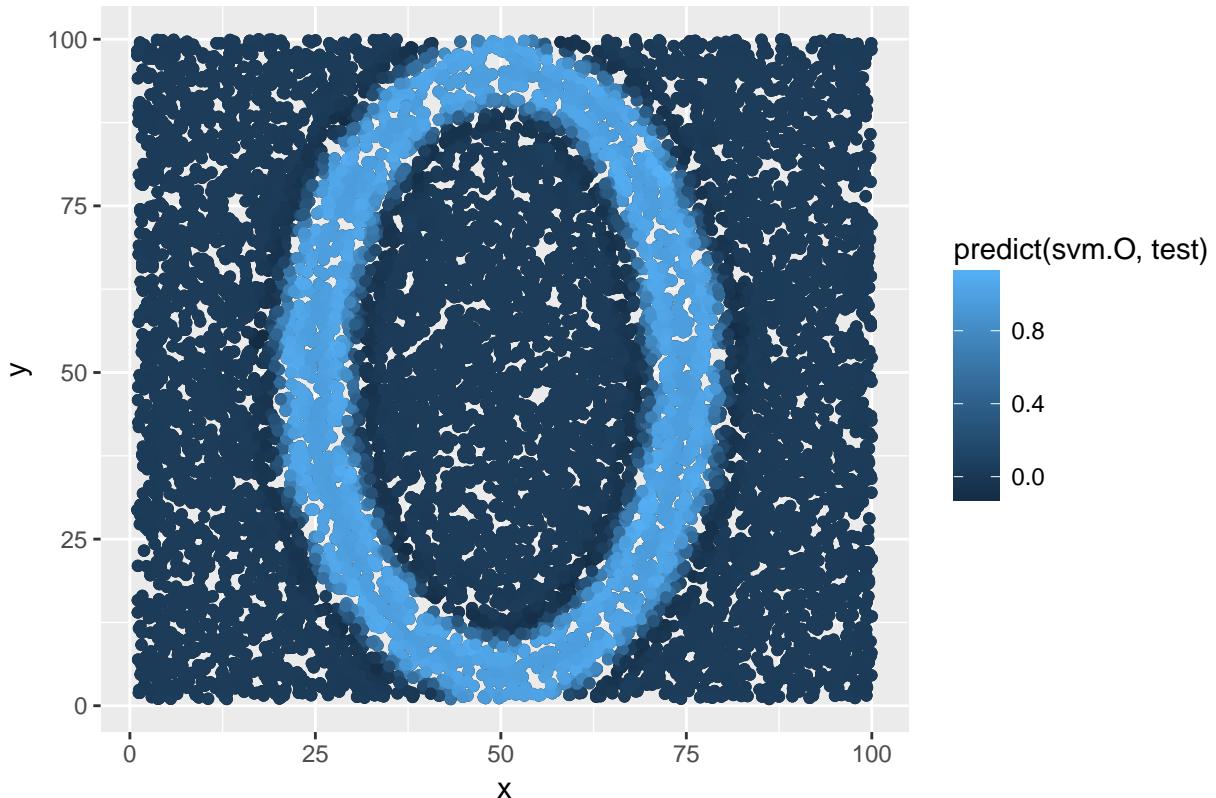
```
svm.0=svm(0 ~ x + y,entrenamiento,method="C-classification",
           kernel="radial",cost=1,gamma=10)
g.test + geom_point(aes(colour = predict(svm.0,test))) +
  labs(title="SVM LETRA O Coste 1 Gamma 10")
```

SVM LETRA O Coste 1 Gamma 10



```
svm.0=svm(0 ~ x + y,entrenamiento,method="C-classification",
           kernel="radial",cost=1,gamma=100)
g.test + geom_point(aes(colour = predict(svm.0,test))) +
  labs(title="SVM LETRA O Coste 1 Gamma 100")
```

SVM LETRA O Coste 1 Gamma 100

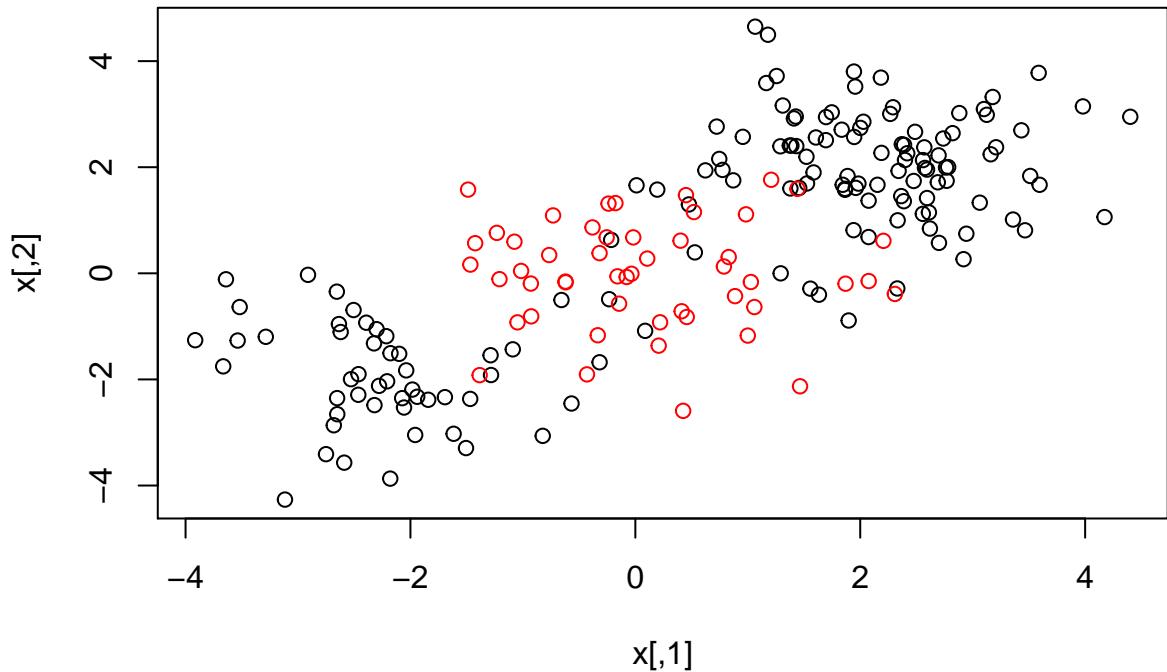


Un menor gamma implica una mayor distancia entre las observaciones que separan los subespacios del SVM luego la estimación es más conservadora, sin embargo un mayor parámetro “fastidio” a la función kernel. También observamos como a mayor gamma las predicciones están menos suavizadas. Tenemos estos dos parámetros y la pregunta sería ¿cuál es la mejor elección? Dónde está el mejor punto para obtener un buen resultado en este balance de sesgo-varianza.

La solución está en utilizar la función `tune`, veamos mejor el siguiente ejemplo.

Generar datos

```
set.seed (1)
x = matrix (rnorm(200*2), ncol = 2)
x[1:100, ] = x[1:100 , ] + 2
x[101:150, ] = x[101:150, ] - 2
y = c(rep(1 , 150), rep(2, 50))
dat = data.frame(x = x, y = as.factor(y))
plot(x, col=y)
```



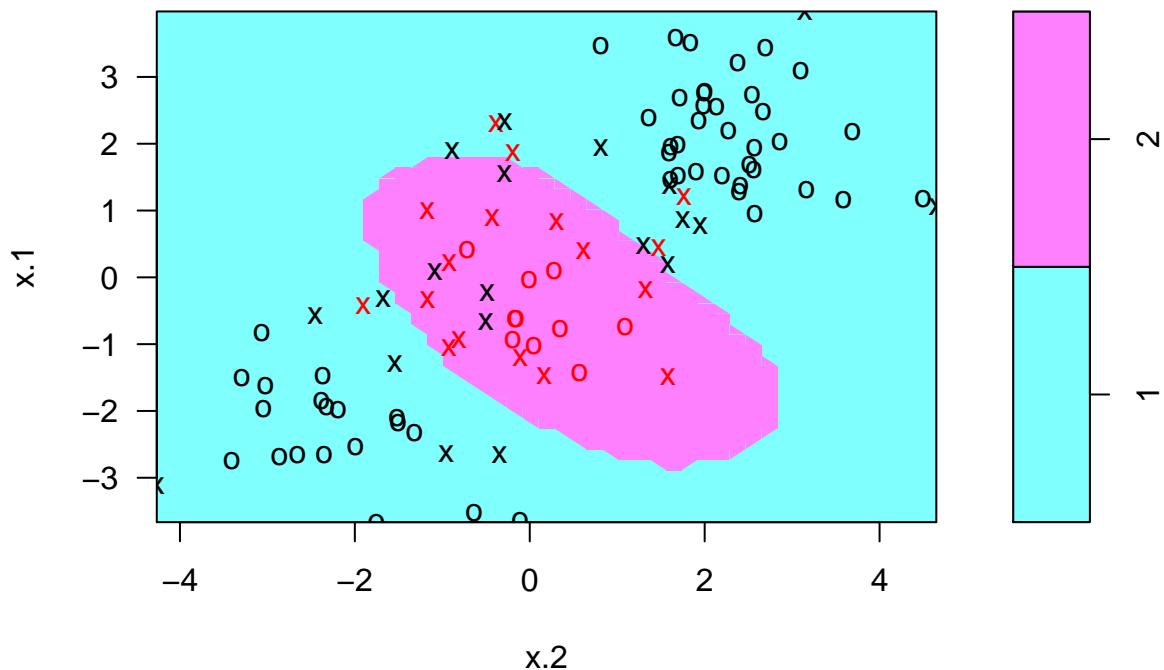
SVM using radial kernel with gamma = 1

```
train = sample(200, 100) # traing set indexes
svmfit = svm(y~.,
              data = dat[train, ],
              kernel = "radial", # for radial kernel
              gamma = 1,
              cost = 1)
```

Modelo

```
plot(svmfit , dat[train ,])
```

SVM classification plot



```
summary (svmfit)
```

```
##  
## Call:  
## svm(formula = y ~ ., data = dat[train, ], kernel = "radial",  
##       gamma = 1, cost = 1)  
##  
##  
## Parameters:  
##   SVM-Type: C-classification  
##   SVM-Kernel: radial  
##         cost: 1  
##        gamma: 1  
##  
## Number of Support Vectors: 37  
##  
##  ( 17 20 )  
##  
##  
## Number of Classes: 2  
##  
## Levels:  
##  1 2
```

k-fold cross-validation

```
set.seed(1)  
tune.out = tune(svm,  
                 y~.,  
                 data = dat[train, ],  
                 kernel = "radial",
```

```

            ranges = list(cost = c(0.1, 1, 10, 100, 1000),
                           gamma = c(0.5, 1, 2, 3, 4)))
summary (tune.out)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost gamma
##     1      2
##
## - best performance: 0.12
##
## - Detailed performance results:
##   cost gamma error dispersion
## 1 1e-01  0.5  0.27 0.11595018
## 2 1e+00  0.5  0.13 0.08232726
## 3 1e+01  0.5  0.15 0.07071068
## 4 1e+02  0.5  0.17 0.08232726
## 5 1e+03  0.5  0.21 0.09944289
## 6 1e-01  1.0  0.25 0.13540064
## 7 1e+00  1.0  0.13 0.08232726
## 8 1e+01  1.0  0.16 0.06992059
## 9 1e+02  1.0  0.20 0.09428090
## 10 1e+03 1.0  0.20 0.08164966
## 11 1e-01  2.0  0.25 0.12692955
## 12 1e+00  2.0  0.12 0.09189366
## 13 1e+01  2.0  0.17 0.09486833
## 14 1e+02  2.0  0.19 0.09944289
## 15 1e+03  2.0  0.20 0.09428090
## 16 1e-01  3.0  0.27 0.11595018
## 17 1e+00  3.0  0.13 0.09486833
## 18 1e+01  3.0  0.18 0.10327956
## 19 1e+02  3.0  0.21 0.08755950
## 20 1e+03  3.0  0.22 0.10327956
## 21 1e-01  4.0  0.27 0.11595018
## 22 1e+00  4.0  0.15 0.10801234
## 23 1e+01  4.0  0.18 0.11352924
## 24 1e+02  4.0  0.21 0.08755950
## 25 1e+03  4.0  0.24 0.10749677

testing
table(true = dat[-train, "y"],
      pred = predict(tune.out$best.model, newx=dat[-train, ]))

##
##   pred
## true 1 2
##   1 56 21
##   2 18 5

```

Curva ROC

```

library(ROCR)
# Creating function to plot ROC curve
rocplot = function(pred, truth, ...) {
  predob = prediction(pred, truth)
  perf = performance(predob, "tpr", "fpr")
  plot(perf, ...)
}

# Optimal model based on cross-validation
svmfit.opt = svm(y ~.,
                  data = dat[train, ],
                  kernel = "radial",
                  gamma = 2,
                  cost = 1,
                  decision.values = T) # to obtain the fitted values for a given SVM model

# By increasing gamma we can produce a more flexible
# fit and generate further improvements in accuracy
svmfit.flex = svm(y~.,
                  data = dat[train, ],
                  kernel = "radial",
                  gamma = 50,
                  cost = 1,
                  decision.values = T)

fitted1 = attributes(predict(svmfit.opt, dat[train, ],
                             decision.values = TRUE))$decision.values
fitted2 = attributes(predict(svmfit.flex, dat[train, ],
                             decision.values =TRUE))$decision.values
par(mfrow = c(1,2))

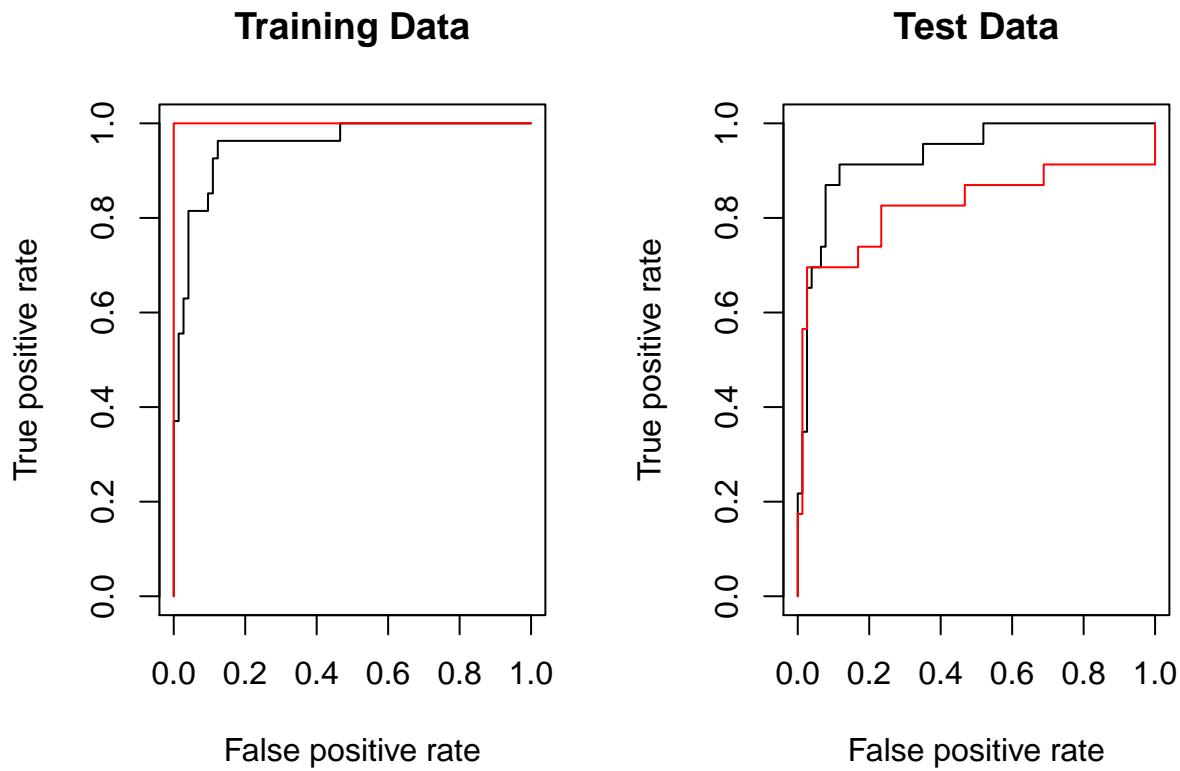
# ROC plot for optimal model
rocplot(fitted1,
        dat[train, "y"],
        main = "Training Data")
# ROC model of flexible model
rocplot(fitted2,
        dat[train, "y"],
        add = T,
        col = "red")

fitted3 = attributes(predict(svmfit.opt, dat[-train, ],
                             decision.values = T))$decision.values
fitted4 = attributes(predict(svmfit.flex, dat[-train, ],
                             decision.values =T))$decision.values

rocplot(fitted3,
        dat[-train, "y"],
        main = "Test Data")

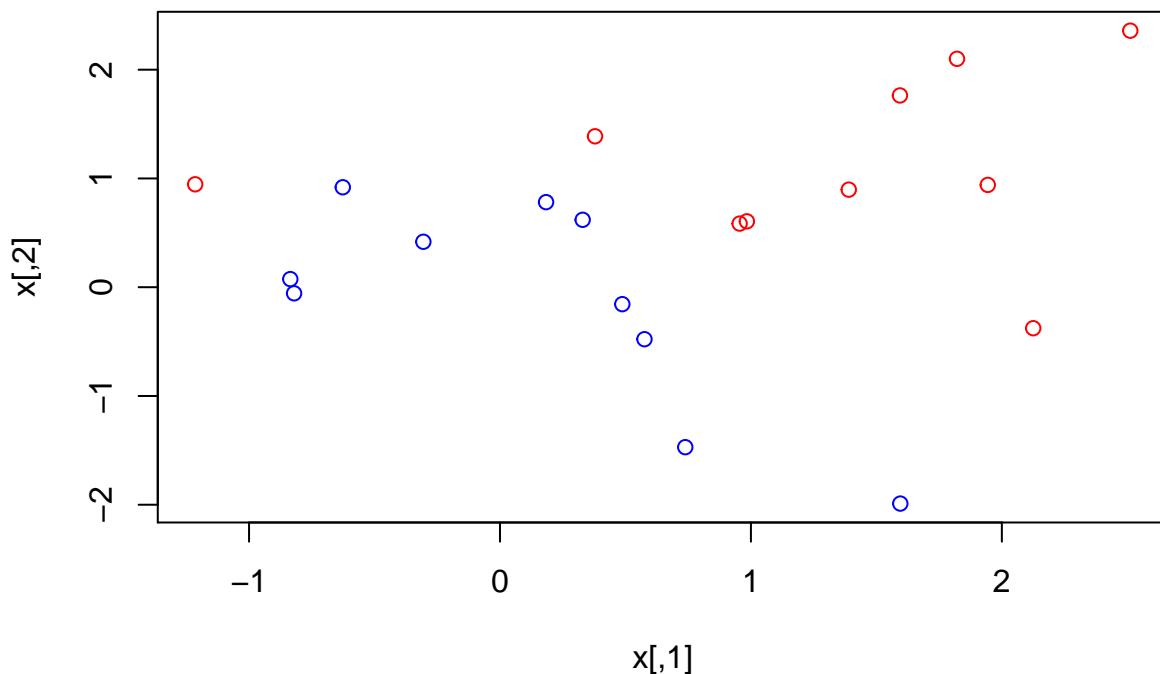
rocplot(fitted4,
        dat[-train, "y"],
        add = T,
        col = "red")

```



Generar datos

```
set.seed(1)
x = matrix(rnorm(20*2), ncol = 2)
y = c(rep(-1,10), rep(1 ,10))
x[y == 1, ] = x[ y== 1, ] + 1 # Separating observation by 1
plot(x, col = (3-y)) # plotting data
```



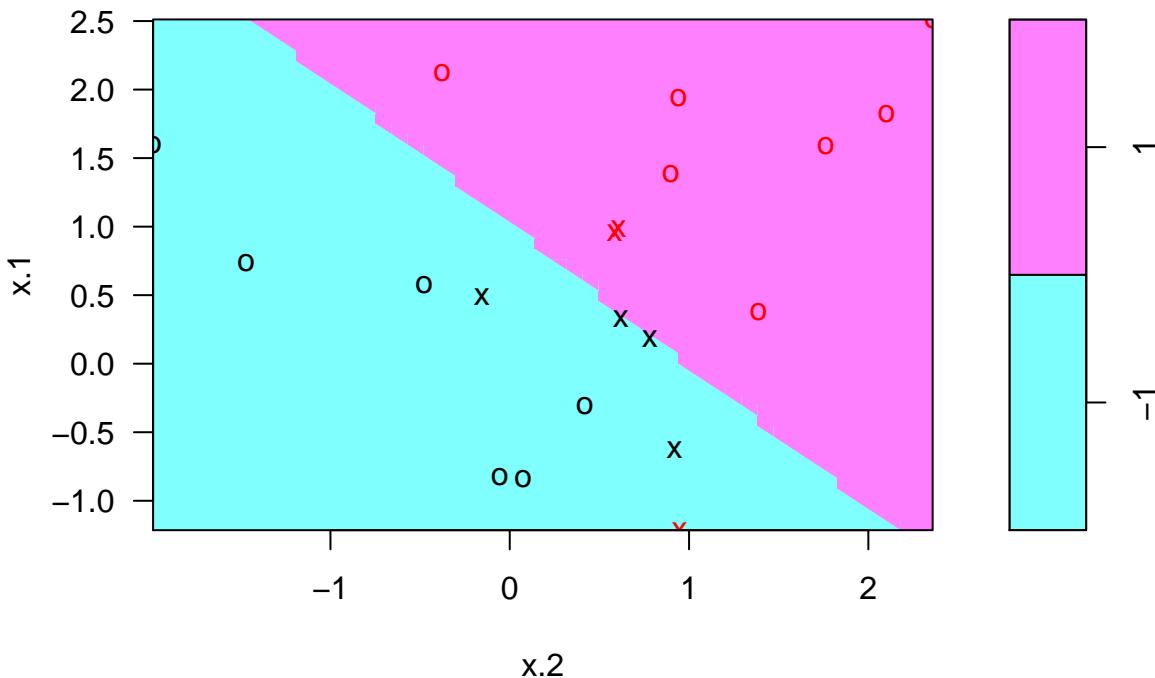
```
library(e1071)
dat = data.frame(x = x, y = as.factor(y))

svmfit1 = svm(y ~ .,
               data = dat,
               kernel = "linear", # support vector classifier
               cost = 10, # cost of a violation to the margin.
               scale = FALSE) # not to scale each feature to have mean zero or standard deviation one
```

Resultados

```
plot(svmfit1, dat) # plot support vector classifier
```

SVM classification plot



```
svmfit1$index # return support vector  
  
## [1] 1 2 5 7 14 16 17  
summary(svmfit1) # summary of the model  
  
##  
## Call:  
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10,  
##       scale = FALSE)  
##  
##  
## Parameters:  
##   SVM-Type: C-classification  
##   SVM-Kernel: linear  
##         cost: 10  
##        gamma: 0.5  
##  
## Number of Support Vectors: 7  
##  
## ( 4 3 )  
##  
##  
## Number of Classes: 2  
##  
## Levels:  
## -1 1  
cost = 0.1 ?  
svmfit2 = svm(y~.,  
              data=dat,
```

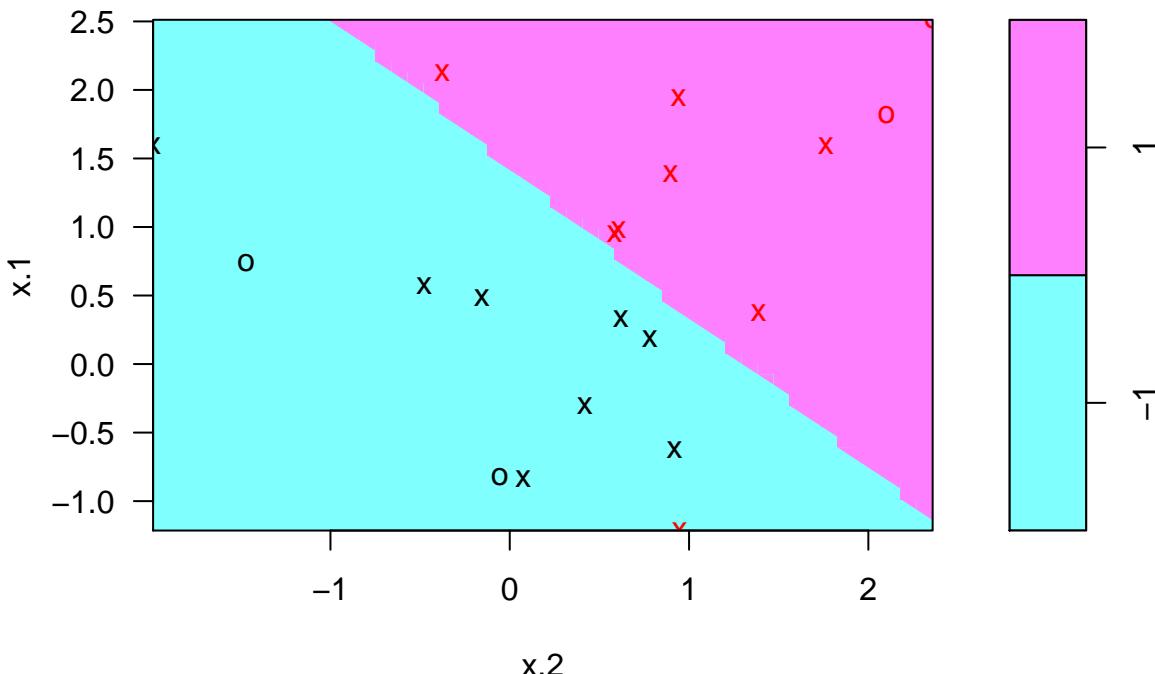
```

    kernel ="linear",
    cost =0.1,
    scale =FALSE )

plot(svmfit2, dat)

```

SVM classification plot



k-fold cross-validation

```

tune.out = tune(svm, #ten-fold cross-validation on a set of models of interest.
                y~.,
                data = dat,
                kernel = "linear",
                ranges = list(cost=c(0.001, 0.01, 0.1, 1, 5, 10, 100))) # Different models
summary(tune.out)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost
##   0.1
##
## - best performance: 0.05
##
## - Detailed performance results:
##   cost error dispersion
## 1 1e-03 0.65 0.3374743
## 2 1e-02 0.65 0.3374743

```

```

## 3 1e-01 0.05 0.1581139
## 4 1e+00 0.10 0.2108185
## 5 5e+00 0.15 0.2415229
## 6 1e+01 0.15 0.2415229
## 7 1e+02 0.15 0.2415229

bestmod = tune.out$best.model # Assigning the best model
summary(bestmod)

##
## Call:
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
## 0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##
##
## Parameters:
##   SVM-Type: C-classification
##   SVM-Kernel: linear
##   cost: 0.1
##   gamma: 0.5
##
## Number of Support Vectors: 16
##
## ( 8 8 )
##
##
## Number of Classes: 2
##
## Levels:
## -1 1

```

Test del modelo

```

xtest = matrix(rnorm(20*2), ncol = 2)
ytest = sample(c(-1,1), 20, rep=TRUE)
xtest[ytest == 1, ] = xtest[ytest== 1, ] + 1
testdat = data.frame(x=xtest, y=as.factor(ytest)) # Creating a testing set
ypred = predict(bestmod, testdat) # predicting using best model
table(predict = ypred, truth = testdat$y) # confusion matrix

##
##      truth
## predict -1 1
##        -1 6 5
##        1  2 7

```

Librería caret

Ver más información

Como ejemplo, utilizaremos el conjunto de datos de segmentación desde el paquete caret y usando la función `createDataPartition()` de `caret` para producir conjuntos de datos de entrenamiento y prueba.

```

# Training SVM Models
library(caret)

## Loading required package: lattice

```

```

## Warning: package 'lattice' was built under R version 3.3.2
library(dplyr)          # Used by caret

## Warning: package 'dplyr' was built under R version 3.3.2
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##   filter, lag
## The following objects are masked from 'package:base':
##   intersect, setdiff, setequal, union
library(kernlab)         # support vector machine

##
## Attaching package: 'kernlab'
## The following object is masked from 'package:ggplot2':
##   alpha
library(pROC)            # plot the ROC curves

## Type 'citation("pROC")' for a citation.
##
## Attaching package: 'pROC'
## The following objects are masked from 'package:stats':
##   cov, smooth, var
#### Get the Data
# Load the data and construct indices to divide it into training and test data sets.
data(segmentationData)      # Load the segmentation data set
trainIndex <- createDataPartition(segmentationData$Case,p=.5,list=FALSE)
trainData <- segmentationData[trainIndex,]
testData  <- segmentationData[-trainIndex,]
trainX <- trainData[,4:61]      # Pull out the variables for training
sapply(trainX,summary)        # Look at a summary of the training data

##           AngleCh1 AreaCh1 AvgIntenCh1 AvgIntenCh2 AvgIntenCh3 AvgIntenCh4
## Min.    0.03184    150.0     15.16      1.00      0.12     0.6316
## 1st Qu. 53.35000   192.0     36.27     43.53     35.26    40.0700
## Median  90.12000   256.0     64.27    173.20     68.95    91.8800
## Mean    89.60000   323.4     124.40    188.30    100.60   140.2000
## 3rd Qu. 124.70000  364.8     144.80    284.10    133.10   194.4000
## Max.   179.90000  2186.0    1419.00   862.70    725.20   809.1000
##           ConvexHullAreaRatioCh1 ConvexHullPerimRatioCh1 DiffIntenDensityCh1
## Min.            1.008             0.5708            25.76
## 1st Qu.          1.069             0.8576            43.60
## Median          1.149             0.9138            56.18
## Mean            1.208             0.8956            72.86
## 3rd Qu.          1.284             0.9556            81.32
## Max.            2.900             0.9965            383.40

```

```

##          DiffIntenDensityCh3 DiffIntenDensityCh4 EntropyIntenCh1
## Min.           1.501           2.711           4.708
## 1st Qu.        30.510          31.680          6.049
## Median         54.720          61.120          6.589
## Mean           77.610          87.310          6.587
## 3rd Qu.        101.700         116.300          7.057
## Max.          446.900         479.600          9.476
##          EntropyIntenCh3 EntropyIntenCh4 EqCircDiamCh1 EqEllipseLWRCh1
## Min.           0.2167          0.512           13.86          1.009
## 1st Qu.        4.7540          4.631           15.67          1.409
## Median         5.8680          5.875           18.08          1.832
## Mean           5.5220          5.497           19.55          2.102
## 3rd Qu.        6.6130          6.650           21.57          2.414
## Max.          7.9960          8.081           52.76          9.131
##          EqEllipseOblateVolCh1 EqEllipseProlateVolCh1 EqSphereAreaCh1
## Min.           149.5            63.85          603.8
## 1st Qu.        276.9            150.80          771.5
## Median         434.7            221.60          1027.0
## Mean           731.3            368.40          1296.0
## 3rd Qu.        793.9            394.40          1462.0
## Max.          11250.0          6315.00          8746.0
##          EqSphereVolCh1 FiberAlign2Ch3 FiberAlign2Ch4 FiberLengthCh1
## Min.           1395             1.000           1.000          11.87
## 1st Qu.        2015             1.286           1.252          20.76
## Median         3096             1.459           1.447          29.76
## Mean           5027             1.446           1.426          34.96
## 3rd Qu.        5258             1.640           1.617          41.31
## Max.          76910            2.000           2.000          220.20
##          FiberWidthCh1 IntenCoocASMCh3 IntenCoocASMCh4 IntenCoocContrastCh3
## Min.           4.530            0.004926          0.004514          0.01627
## 1st Qu.        7.357            0.009734          0.017700          4.43300
## Median         9.649            0.034670          0.049760          8.75300
## Mean           10.240            0.096780          0.100200          9.95300
## 3rd Qu.        12.590            0.117800          0.112900          13.92000
## Max.          27.430            0.936600          0.823100          58.92000
##          IntenCoocContrastCh4 IntenCoocEntropyCh3 IntenCoocEntropyCh4
## Min.           0.09459           0.2546          0.6198
## 1st Qu.        4.13100           5.1690          5.0920
## Median         6.39700           6.4040          6.0850
## Mean           7.76700           5.9310          5.7330
## 3rd Qu.        9.83300           7.1180          6.7920
## Max.          58.01000           8.0680          8.0710
##          IntenCoocMaxCh3 IntenCoocMaxCh4 KurtIntenCh1 KurtIntenCh3
## Min.           0.01429           0.01342          -1.34200          -1.3520
## 1st Qu.        0.05006           0.10790          -0.47610          0.1798
## Median         0.17310           0.21510          0.01946          1.4650
## Mean           0.22670           0.24900          0.82940          3.2280
## 3rd Qu.        0.33900           0.33650          0.99880          3.8920
## Max.          0.96830           0.91110          38.97000          78.7000
##          KurtIntenCh4 LengthCh1 NeighborAvgDistCh1 NeighborMinDistCh1
## Min.           -1.4990          15.77           146.1           10.08
## 1st Qu.        -0.8340          21.98           197.0           22.53
## Median         -0.2360          27.61           227.4           27.66
## Mean           1.0560          30.51           230.5           29.46

```

```

## 3rd Qu.      0.8189    35.33        259.8       33.89
## Max.       82.7200   103.00       375.8      121.80
##           NeighborVarDistCh1 PerimCh1 ShapeBFRCh1 ShapeLWRCh1 ShapeP2ACh1
## Min.          56.89     47.49     0.2501     1.003     1.089
## 1st Qu.      88.41     63.79     0.5252     1.334     1.383
## Median      107.30    78.40     0.6037     1.630     1.780
## Mean         105.20    90.41     0.5953     1.808     2.051
## 3rd Qu.     122.10   101.10     0.6735     2.058     2.411
## Max.        154.10   459.80     0.8681     5.924     7.912
##           SkewIntenCh1 SkewIntenCh3 SkewIntenCh4 SpotFiberCountCh3
## Min.        -2.6660   -1.1140   -1.0040     0.000
## 1st Qu.     0.3040    0.8458    0.4014     1.000
## Median      0.6272    1.3210    0.7480     2.000
## Mean         0.7080    1.4780    0.9480     1.863
## 3rd Qu.     1.0380    1.9010    1.2370     3.000
## Max.        5.2740    8.5010    8.0690    13.000
##           SpotFiberCountCh4 TotalIntenCh1 TotalIntenCh2 TotalIntenCh3
## Min.        2.000      2450        1        24
## 1st Qu.     4.000      9742    15080     8818
## Median      6.000      19530    49470    19560
## Mean         6.887      36510    51860    27420
## 3rd Qu.     8.000      36300    72690    36660
## Max.        40.000     735100   362500   205100
##           TotalIntenCh4 VarIntenCh1 VarIntenCh3 VarIntenCh4 WidthCh1
## Min.        96        11.47     0.8693     2.44     6.553
## 1st Qu.    9890      25.63     36.5700    47.73    13.850
## Median     25530     43.51     71.0800    86.79    16.220
## Mean        40350     72.48    102.3000   121.00   17.640
## 3rd Qu.    56340     82.38    127.8000   161.00   19.770
## Max.       519600    575.80   757.0000   933.50   54.740
##           XCentroid YCentroid
## Min.        9.0       10.0
## 1st Qu.    150.0     85.0
## Median     268.5     160.5
## Mean        265.5     175.5
## 3rd Qu.    383.8     249.0
## Max.       501.0     495.0

```

A continuación, llevamos a cabo un proceso de entrenamiento y ajuste en dos fases.

En el primer paso, que se muestra en el bloque de código de abajo, seleccionamos arbitrariamente algunos parámetros de ajuste y usamos la configuración predeterminada para otros.

En la función `trainControl()` especificamos 5 repeticiones de 10 veces la validación cruzada. En la función `train()` que realmente realiza el trabajo, especificamos el kernel radial utilizando el parámetro de método y el ROC como métrica para evaluar el rendimiento. El parámetro `tuneLength` se establece en 9 valores arbitrarios para C , el “coste” del kernel radial. Este parámetro controla la complejidad del límite entre los vectores soporte. El núcleo radial también requiere establecer un parámetro de suavizado, `sigma`. En este primer paso, dejamos que `train()` use su método predeterminado de calcular una estimación derivada analíticamente para `sigma`. También tenga en cuenta que instruimos a `train()` a centrar y escalar los datos antes de ejecutar el análisis con el parámetro `preProc`.

```

## SUPPORT VECTOR MACHINE MODEL
# First pass
set.seed(1492)
# Setup for cross validation

```

```

ctrl <- trainControl(method="repeatedcv",    # 10 fold cross validation
                      repeats=5,           # do 5 repetitions of cv
                      summaryFunction=twoClassSummary,   # Use AUC to pick the best model
                      classProbs=TRUE)

#Train and Tune the SVM
svm.tune <- train(x=trainX,
                    y= trainData$Class,
                    method = "svmRadial",    # Radial kernel
                    tuneLength = 9,           # 9 values of the cost function
                    preProc = c("center","scale"), # Center and scale data
                    metric="ROC",
                    trControl=ctrl)

svm.tune

## Support Vector Machines with Radial Basis Function Kernel
##
## 1010 samples
##    58 predictor
##    2 classes: 'PS', 'WS'
##
## Pre-processing: centered (58), scaled (58)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 909, 909, 908, 910, 909, 909, ...
## Resampling results across tuning parameters:
##
##     C      ROC      Sens      Spec
## 0.25  0.8652080  0.8613706  0.6762857
## 0.50  0.8692852  0.8629138  0.6868413
## 1.00  0.8739402  0.8687552  0.6863333
## 2.00  0.8733172  0.8706247  0.6991429
## 4.00  0.8672762  0.8752401  0.6908413
## 8.00  0.8573317  0.8758741  0.6629524
## 16.00 0.8467945  0.8684942  0.6368095
## 32.00 0.8365960  0.8682005  0.6127778
## 64.00 0.8304087  0.8651422  0.6133651
##
## Tuning parameter 'sigma' was held constant at a value of 0.01514804
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.01514804 and C = 1.

```

En la segunda etapa, una vez vistos los valores de los parámetros seleccionados en la primera etapa, utilizamos el parámetro `tuneGrid` de `train()` para realizar un análisis de sensibilidad alrededor de los valores `C = 1` y `sigma = 0.015` que produjeron el modelo con el mejor valor ROC. La función `expand.grid()` de R se utiliza para construir `undata.frame` que contiene todas las combinaciones de `C` y `sigma` que queremos probar.

```

# Second pass
# Look at the results of svm.tune and refine the parameter space

set.seed(1492)
# Use the expand.grid to specify the search space
grid <- expand.grid(sigma = c(.01, .015, .02),

```

```

C = c(0.75, 0.9, 1, 1.1, 1.25)
)

# Train and Tune the SVM
svm.tune <- train(x=trainX,
                    y= trainData$Class,
                    method = "svmRadial",
                    preProc = c("center","scale"),
                    metric="ROC",
                    tuneGrid = grid,
                    trControl=ctrl)

svm.tune

## Support Vector Machines with Radial Basis Function Kernel
##
## 1010 samples
##    58 predictor
##    2 classes: 'PS', 'WS'
##
## Pre-processing: centered (58), scaled (58)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 909, 909, 908, 910, 909, 909, ...
## Resampling results across tuning parameters:
##
##     sigma   C      ROC      Sens      Spec
##     0.010  0.75  0.8712074  0.8659860  0.6807460
##     0.010  0.90  0.8722613  0.8678322  0.6784762
##     0.010  1.00  0.8729553  0.8669044  0.6852063
##     0.010  1.10  0.8735111  0.8675245  0.6835397
##     0.010  1.25  0.8739225  0.8684569  0.6813333
##     0.015  0.75  0.8724925  0.8687599  0.6857302
##     0.015  0.90  0.8735299  0.8693753  0.6829841
##     0.015  1.00  0.8738545  0.8690676  0.6874762
##     0.015  1.10  0.8741588  0.8702984  0.6924921
##     0.015  1.25  0.8743568  0.8706154  0.6930476
##     0.200  0.75  0.8288399  0.8952121  0.4851111
##     0.200  0.90  0.8292120  0.8985874  0.4817619
##     0.200  1.00  0.8299083  0.8970443  0.4823016
##     0.200  1.10  0.8310755  0.9025828  0.4728571
##     0.200  1.25  0.8315255  0.9047319  0.4600476
##
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.015 and C = 1.25.

```

Esto fue un cálculo muy intensivo para una mejora de 0,0003247 en la puntuación de la curva ROC, pero muestra algo de lo que `caret` puede hacer.

Para terminar, hemos construido un modelo con un kernel diferente. El kernel lineal es la forma más sencilla de hacerlo. Sólo hay que establecer el parámetro la `C` para este kernel y `train()` con un valor de `C = 1`. El valor de curva ROC resultante de 0.87 no es demasiado bajo.

```
#Linear Kernel
set.seed(1492)
```

```

#Train and Tune the SVM
svm.tune2 <- train(x=trainX,
                     y= trainData$Class,
                     method = "svmLinear",
                     preProc = c("center","scale"),
                     metric="ROC",
                     trControl=ctrl)

svm.tune2

## Support Vector Machines with Linear Kernel
##
## 1010 samples
##   58 predictor
##   2 classes: 'PS', 'WS'
##
## Pre-processing: centered (58), scaled (58)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 909, 909, 908, 910, 909, 909, ...
## Resampling results:
##
##    ROC      Sens     Spec
##    0.8635253 0.874  0.6094603
##
## Tuning parameter 'C' was held constant at a value of 1
##

```

Podemos usar la función `resample()` de `caret` para comparar los resultados generados por los modelos de núcleo radial y núcleo lineal. El siguiente bloque de código y resultados muestra sólo las primeras cinco líneas de la tabla de comparación, pero incluye el resumen de la comparación.

```

rValues <- resamples(list(svm=svm.tune,svm.tune2))
rValues$values

##          Resample    svm~ROC    svm~Sens    svm~Spec Model2~ROC Model2~Sens
## 1 Fold01.Rep1 0.8166667 0.7538462 0.7222222 0.8217949 0.8153846
## 2 Fold01.Rep2 0.8611111 0.8769231 0.7222222 0.8645299 0.8923077
## 3 Fold01.Rep3 0.8380342 0.8615385 0.6944444 0.7918803 0.7846154
## 4 Fold01.Rep4 0.8858974 0.8307692 0.7500000 0.9017094 0.8307692
## 5 Fold01.Rep5 0.9376068 0.9230769 0.7500000 0.9282051 0.8923077
## 6 Fold02.Rep1 0.8807692 0.9230769 0.6111111 0.8688034 0.9230769
## 7 Fold02.Rep2 0.8914141 0.8939394 0.6388889 0.8766835 0.9242424
## 8 Fold02.Rep3 0.7594017 0.6923077 0.6388889 0.7961538 0.7384615
## 9 Fold02.Rep4 0.8688034 0.8153846 0.7777778 0.8918803 0.9076923
## 10 Fold02.Rep5 0.8705128 0.8461538 0.6388889 0.8470085 0.8769231
## 11 Fold03.Rep1 0.9023569 0.9090909 0.7222222 0.8884680 0.8939394
## 12 Fold03.Rep2 0.8681319 0.8923077 0.7142857 0.8325275 0.8923077
## 13 Fold03.Rep3 0.9354701 0.9384615 0.6944444 0.9316239 0.9076923
## 14 Fold03.Rep4 0.8649573 0.8923077 0.5555556 0.8670940 0.8769231
## 15 Fold03.Rep5 0.8952991 0.8153846 0.6666667 0.8491453 0.8461538
## 16 Fold04.Rep1 0.8580220 0.8153846 0.7714286 0.8545055 0.8153846
## 17 Fold04.Rep2 0.9029915 0.9230769 0.5833333 0.8935897 0.9384615
## 18 Fold04.Rep3 0.9440171 0.9384615 0.7500000 0.9440171 0.9230769
## 19 Fold04.Rep4 0.8662393 0.8769231 0.6944444 0.8448718 0.8615385

```

```

## 20 Fold04.Rep5 0.8254945 0.8615385 0.6857143 0.8149451 0.8000000
## 21 Fold05.Rep1 0.8089744 0.8307692 0.6388889 0.7760684 0.8307692
## 22 Fold05.Rep2 0.8957265 0.8769231 0.6666667 0.9021368 0.8923077
## 23 Fold05.Rep3 0.9280303 0.8939394 0.8333333 0.9137205 0.8939394
## 24 Fold05.Rep4 0.9111111 0.8615385 0.7500000 0.8897436 0.8615385
## 25 Fold05.Rep5 0.8316239 0.8615385 0.6111111 0.8264957 0.8461538
## 26 Fold06.Rep1 0.9303419 0.8769231 0.6666667 0.9277778 0.9230769
## 27 Fold06.Rep2 0.8876068 0.8769231 0.7222222 0.8376068 0.8307692
## 28 Fold06.Rep3 0.8606838 0.8615385 0.7500000 0.8444444 0.8923077
## 29 Fold06.Rep4 0.8400000 0.9076923 0.6571429 0.8553846 0.9538462
## 30 Fold06.Rep5 0.8880342 0.8615385 0.8333333 0.8739316 0.8461538
## 31 Fold07.Rep1 0.8594017 0.9230769 0.6666667 0.8555556 0.8923077
## 32 Fold07.Rep2 0.8824786 0.8153846 0.7777778 0.8743590 0.8307692
## 33 Fold07.Rep3 0.8752137 0.8615385 0.7222222 0.8555556 0.8923077
## 34 Fold07.Rep4 0.8577441 0.8939394 0.6666667 0.8379630 0.8787879
## 35 Fold07.Rep5 0.8824786 0.8769231 0.6944444 0.8846154 0.8615385
## 36 Fold08.Rep1 0.9094017 0.9692308 0.5833333 0.9111111 0.9384615
## 37 Fold08.Rep2 0.8256410 0.8769231 0.6388889 0.8205128 0.8615385
## 38 Fold08.Rep3 0.8501099 0.8307692 0.6571429 0.7912088 0.8153846
## 39 Fold08.Rep4 0.9106838 0.9076923 0.7222222 0.8880342 0.9230769
## 40 Fold08.Rep5 0.8918350 0.9090909 0.6388889 0.8775253 0.9090909
## 41 Fold09.Rep1 0.8952991 0.8923077 0.7222222 0.8576923 0.8615385
## 42 Fold09.Rep2 0.8423077 0.8153846 0.6666667 0.8423077 0.8307692
## 43 Fold09.Rep3 0.8910256 0.9384615 0.7222222 0.8700855 0.9538462
## 44 Fold09.Rep4 0.8478632 0.8000000 0.6944444 0.8286325 0.8461538
## 45 Fold09.Rep5 0.8602564 0.8461538 0.7500000 0.8653846 0.8923077
## 46 Fold10.Rep1 0.8829060 0.8461538 0.7222222 0.8876068 0.8615385
## 47 Fold10.Rep2 0.8829060 0.8769231 0.7500000 0.8897436 0.8769231
## 48 Fold10.Rep3 0.8572650 0.8769231 0.6111111 0.8688034 0.9076923
## 49 Fold10.Rep4 0.8944444 0.8615385 0.7222222 0.8405983 0.8923077
## 50 Fold10.Rep5 0.8632479 0.9230769 0.6111111 0.8722222 0.8615385

## Model2-Spec
## 1 0.6111111
## 2 0.6111111
## 3 0.5277778
## 4 0.8055556
## 5 0.6666667
## 6 0.5277778
## 7 0.5277778
## 8 0.6666667
## 9 0.7500000
## 10 0.5555556
## 11 0.6111111
## 12 0.6571429
## 13 0.6388889
## 14 0.5277778
## 15 0.6111111
## 16 0.6857143
## 17 0.6388889
## 18 0.8055556
## 19 0.5833333
## 20 0.6857143
## 21 0.5555556
## 22 0.5555556

```

```

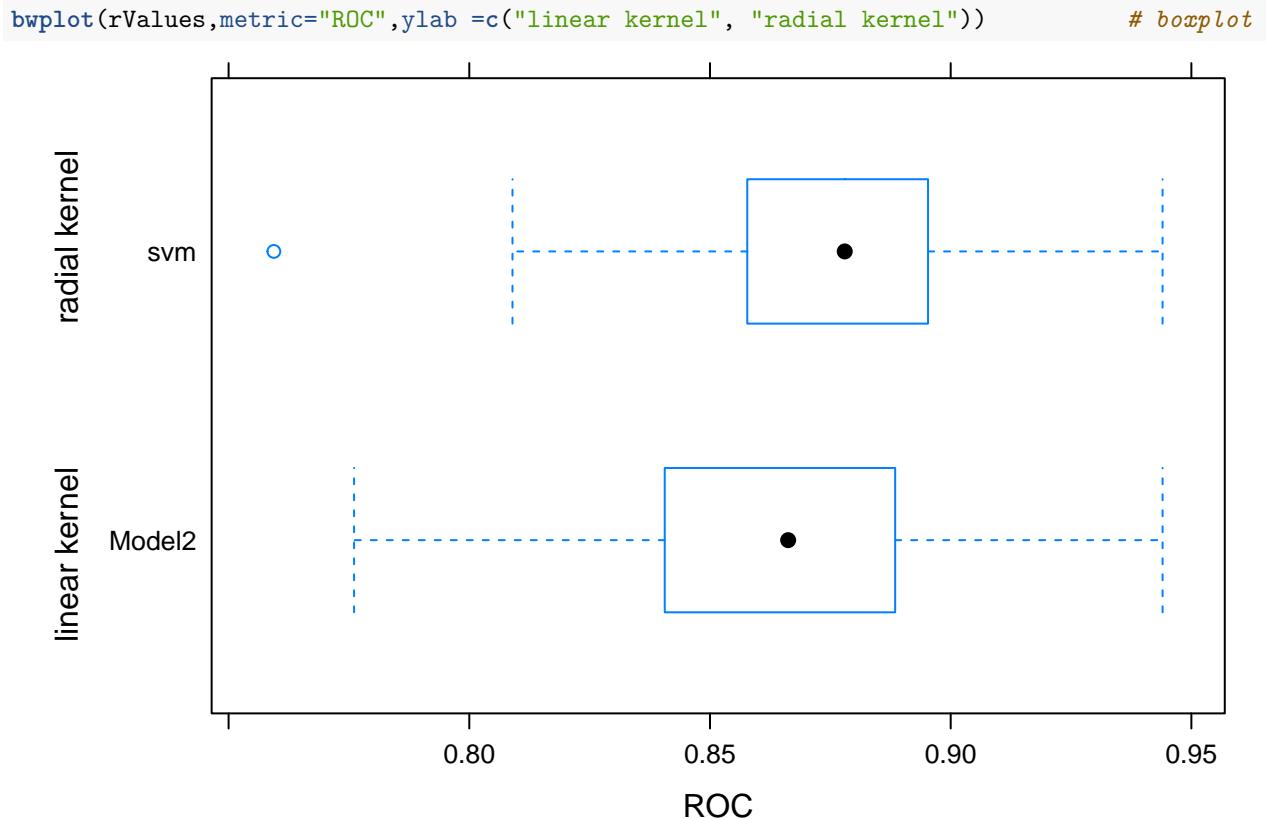
## 23 0.6944444
## 24 0.6388889
## 25 0.5833333
## 26 0.5833333
## 27 0.6388889
## 28 0.6666667
## 29 0.5428571
## 30 0.6944444
## 31 0.5555556
## 32 0.7222222
## 33 0.5000000
## 34 0.5000000
## 35 0.6666667
## 36 0.6388889
## 37 0.4444444
## 38 0.4571429
## 39 0.6666667
## 40 0.4722222
## 41 0.6111111
## 42 0.6388889
## 43 0.5555556
## 44 0.5833333
## 45 0.6944444
## 46 0.6666667
## 47 0.6944444
## 48 0.5555556
## 49 0.4722222
## 50 0.5277778

summary(rValues)

##
## Call:
## summary.resamples(object = rValues)
##
## Models: svm, Model2
## Number of resamples: 50
##
## ROC
##      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## svm 0.7594 0.8578 0.8780 0.8744 0.8951 0.944 0
## Model2 0.7761 0.8410 0.8662 0.8635 0.8884 0.944 0
##
## Sens
##      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## svm 0.6923 0.8462 0.8769 0.8706 0.9043 0.9692 0
## Model2 0.7385 0.8462 0.8779 0.8740 0.9043 0.9538 0
##
## Spec
##      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
## svm 0.5556 0.6435 0.6944 0.6930 0.7222 0.8333 0
## Model2 0.4444 0.5556 0.6111 0.6095 0.6667 0.8056 0

```

La gráfica de abajo parecería dar al kernel radial la ventaja de ser el mejor modelo.



En la práctica se recomienda:

- Transformar datos al formato de un paquete SVM.
- Llevar a cabo un escalado simple de los datos.
- Considere el kernel RBF (radial basis function) $K(x, y) = \exp(-\gamma \|x - y\|^2)$.
- Utilice la validación cruzada para encontrar el mejor parámetro C y γ .
- Utilice el mejor parámetro C y γ para entrenar todo el conjunto de entrenamiento.
- Validar sobre una muestra de test.

SVM para clasificación multi-clase

Veamos los datos Iris en R

```
data(iris)
?iris
head(iris,5)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         1.4         0.2  setosa
## 2          4.9         3.0         1.4         0.2  setosa
## 3          4.7         3.2         1.3         0.2  setosa
## 4          4.6         3.1         1.5         0.2  setosa
## 5          5.0         3.6         1.4         0.2  setosa
```

Divide los datos del Iris entre x (contiene todas las características) y y sólo las clases

```
attach(iris)
x <- subset(iris, select=-Species)
y <- Species
```

Crear modelo SVM y mostrar resumen

```
svm_model <- svm(Species ~ ., data=iris)
summary(svm_model)

##
## Call:
## svm(formula = Species ~ ., data = iris)
##
##
## Parameters:
##   SVM-Type: C-classification
##   SVM-Kernel: radial
##   cost: 1
##   gamma: 0.25
##
## Number of Support Vectors: 51
##
## ( 8 22 21 )
##
##
## Number of Classes: 3
##
## Levels:
## setosa versicolor virginica
```

O puede usar un comando como este

```
svm_model1 <- svm(x,y)
summary(svm_model1)

##
## Call:
## svm.default(x = x, y = y)
##
##
## Parameters:
##   SVM-Type: C-classification
##   SVM-Kernel: radial
##   cost: 1
##   gamma: 0.25
##
## Number of Support Vectors: 51
##
## ( 8 22 21 )
##
##
## Number of Classes: 3
##
## Levels:
## setosa versicolor virginica
```

```

predict
pred <- predict(svm_model1,x)
system.time(pred <- predict(svm_model1,x))

```

```

##    user  system elapsed
##  0.002  0.000  0.004

```

Veamos el resultado de la matriz de confusión de la predicción, usando la tabla de comandos para comparar el resultado de la predicción SVM y los datos de clase en la variable y.

```
table(pred,y)
```

```

##          y
## pred      setosa versicolor virginica
##   setosa     50        0        0
##   versicolor    0       48        2
##   virginica     0        2       48

```

Tuneado de SVM

```

svm_tune <- tune(svm, train.x=x, train.y=y, probability=TRUE,
                  kernel="radial", ranges=list(cost=10^{(-1:2)}, gamma=c(.5,1,2)))

print(svm_tune)

```

```

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost gamma
##     1     0.5
##
## - best performance: 0.03333333

```

Después de encontrar el mejor coste y gamma, puede crear svm model de nuevo y tratar de ejecutar de nuevo

```
svm_model_after_tune <- svm(Species ~ ., data=iris, kernel="radial", cost=1, gamma=0.5)
summary(svm_model_after_tune)
```

```

##
## Call:
## svm(formula = Species ~ ., data = iris, kernel = "radial", cost = 1,
##      gamma = 0.5)
##
##
## Parameters:
##   SVM-Type: C-classification
##   SVM-Kernel: radial
##   cost: 1
##   gamma: 0.5
##
## Number of Support Vectors: 59
##
## ( 11 23 25 )
##
```

```
##  
## Number of Classes: 3  
##  
## Levels:  
##   setosa versicolor virginica
```

Ejecutar `predict` de nuevo con el nuevo modelo

```
pred <- predict(svm_model_after_tune,x)  
system.time(predict(svm_model_after_tune,x))
```

```
##    user  system elapsed  
##    0.004  0.000  0.004
```

Veamos el resultado de la matriz de confusión de la predicción, usando la `table` para comparar el resultado de la predicción SVM y los datos de clase en la variable `y`.

```
table(pred,y)
```

```
##          y  
## pred      setosa versicolor virginica  
##   setosa     50       0       0  
##   versicolor  0      48       2  
##   virginica   0       2      48
```