

Trabajo 3

Alejandro García Montoro

2 de junio de 2016

Cómo se escribe una fórmula para que el modelo devuelva los pesos que nos interesan, modelando lo que queremos:

- Modelo general: `m1 <- lm(Y~.,data=datos)`
- Modelo con columnas especificadas: `m2 <- lm(Y~X1+X2+X3,data=datos)`
- Modelos más complejos: `m1 <- lm(Y~I(X^2)+X2,data=datos)`
- Combinaciones polinómicas: `log gpoly()`
- Para predecir cosas dependientes (p.ej., dos atributos combinados): `lm(Y~X1+X2+X1:X2,data=datos)`, que es equivalente a escribir `lm(Y~X1*X2,data=datos)`.

Ejercicio 1

Carguemos primero los datos necesarios para realizar el ejercicio y echémosle un primer vistazo a la base de datos:

```
# Cargamos la librería necesaria para usar la base de datos Auto
# Para usarla, hay que instalar con la orden
# install.packages('ISLR')
library(ISLR)

# Usamos Auto por defecto, evitando así poner el prefijo Auto$
# siempre que queramos acceder a una característica de esa base de datos
attach(Auto)
```

Si ejecutamos las órdenes siguientes

```
class(Auto)
dim(Auto)
colnames(Auto)
```

podemos obtener información de la forma que tiene nuestra base de datos. Vemos así que tiene forma de `data.frame`, con 392 filas y 9 columnas, cuyos nombres son los siguientes: `mpg`, `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration`, `year`, `origin`, `name`.

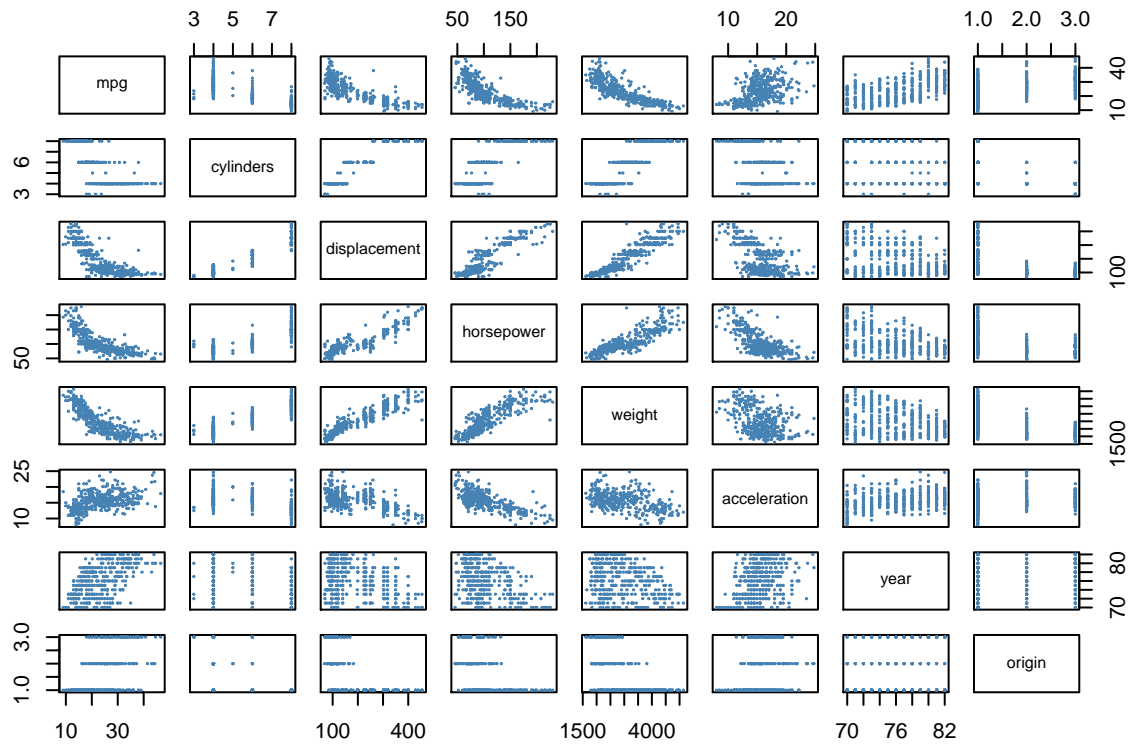
Podemos eliminar la última columna, que es el nombre del vehículo y la única variable no numérica, para poder trabajar con comodidad más adelante:

```
# Eliminamos la última columna
Auto <- Auto[,seq(ncol(Auto)-1)]
```

Apartado a

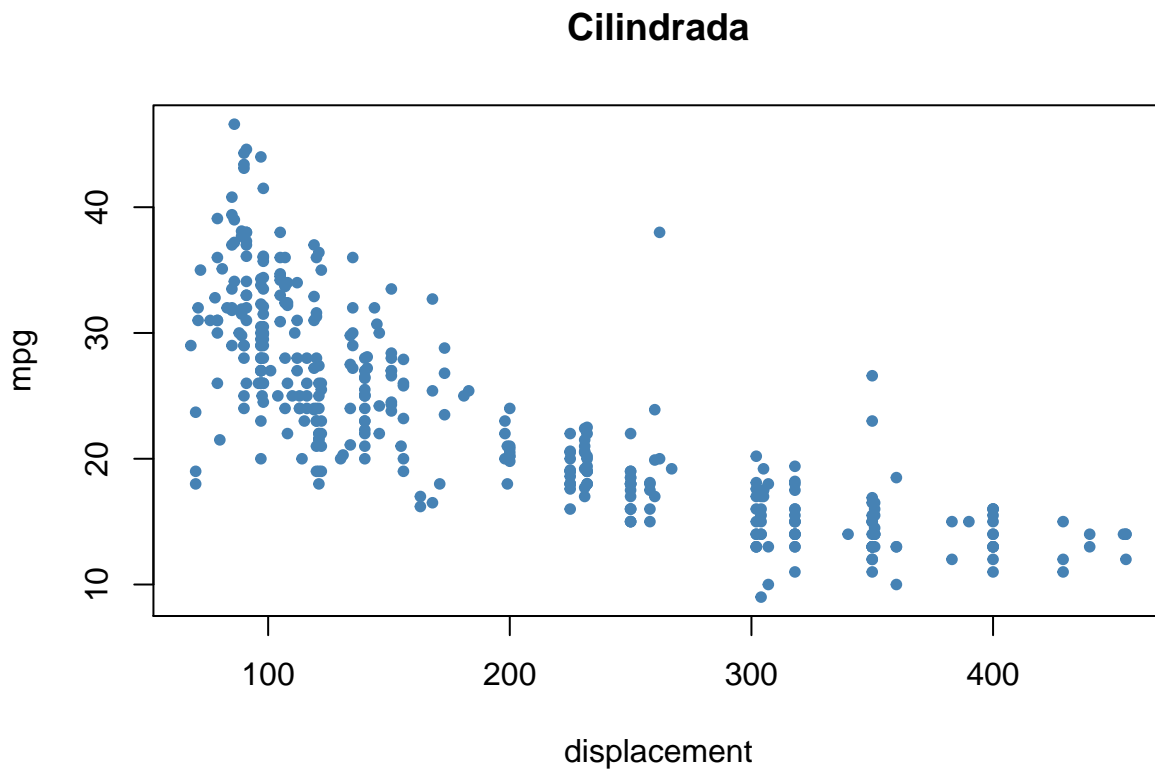
Para visualizar las dependencias entre `mpg` y las otras características podemos usar las funciones `pairs()` y `boxplot()`:

```
# Visualizamos la relación entre todos los pares de variables
pairs(Auto, pch=20, cex=0.2, col="steelblue")
```

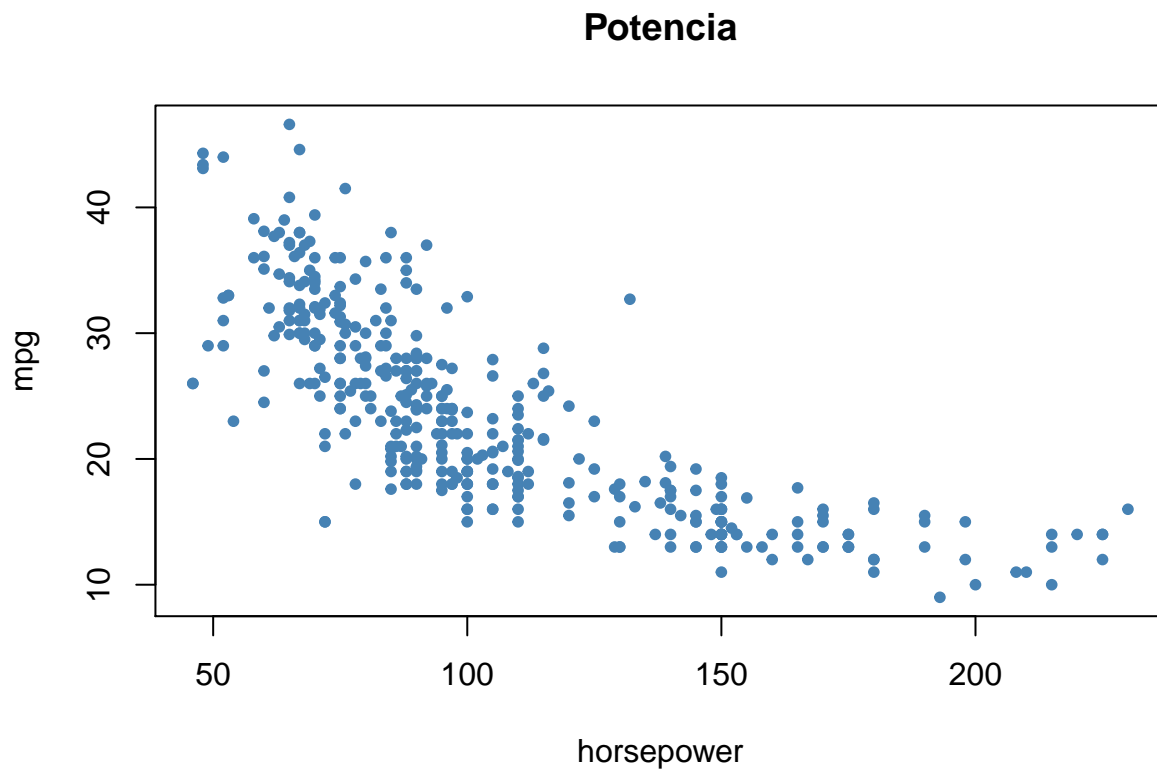


Podemos visualizar más de cerca las variables que parecen más relevantes para predecir la variable `mpg`:

```
# Plot para mpg-displacement
plot(displacement, mpg, pch=20, col="steelblue",
     main="Cilindrada")
```

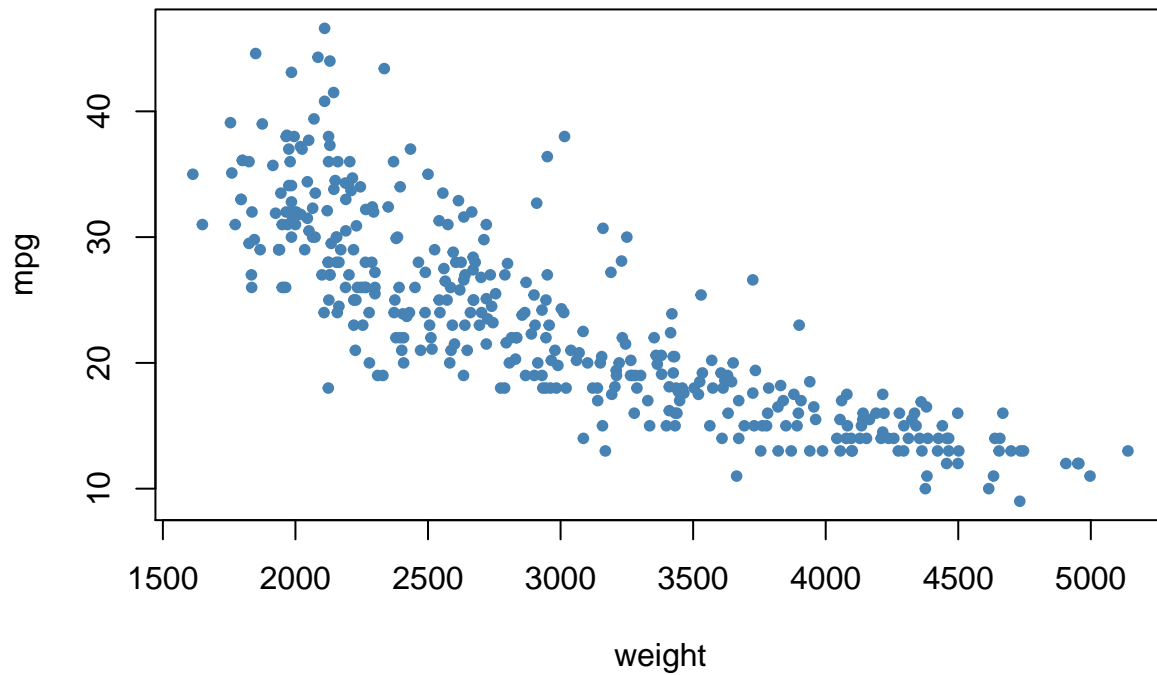


```
# Plot para mpg-horsepower  
plot(horsepower, mpg, pch=20, col="steelblue",  
      main="Potencia")
```



```
# Plot para mpg-weight  
plot(weight, mpg, pch=20, col="steelblue",  
      main="Peso")
```

Peso

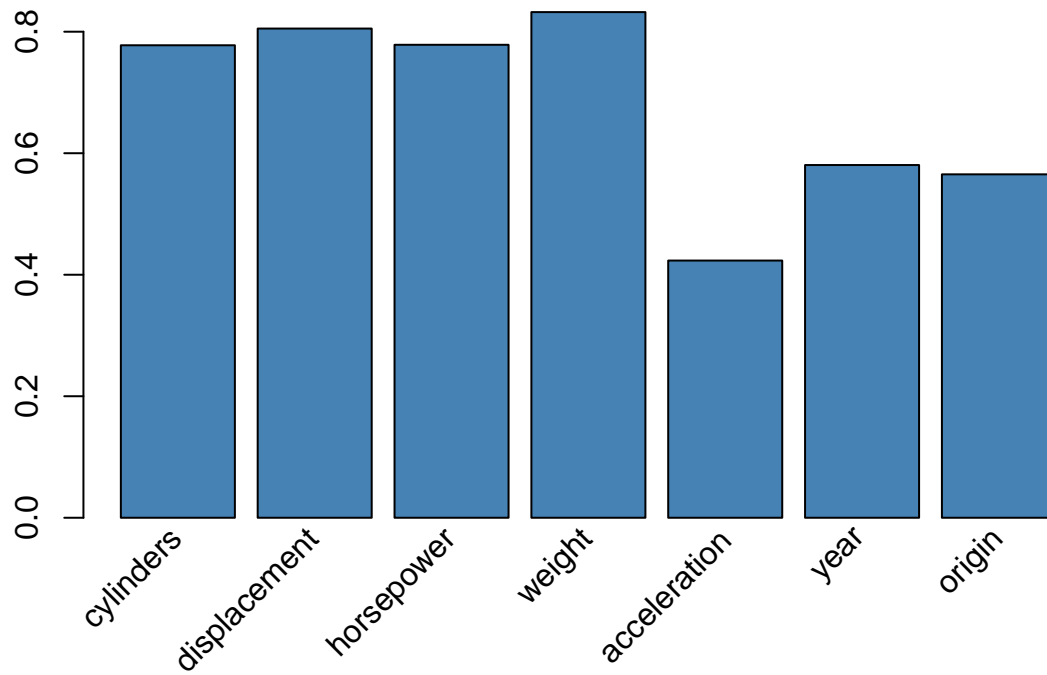


Apartado b

Podemos estudiar de forma numérica la correlación entre mpg y las demás variables:

```
# Tomamos los valores absolutos de la correlación entre mpg y todas las demás variables,  
# sin incluirse a sí misma  
corr <- abs(cor(Auto))["mpg",-1]  
  
# Visualizamos el grado de correlación en un gráfico de barras.  
# Creamos el gráfico.  
bp <- barplot(corr, axes = FALSE, axisnames = FALSE, col = "steelblue",  
              main="Correlación entre mpg y las demás variables")  
  
# Añadimos el texto, girado 45 grados.  
text(bp, par("usr")[3]-0.02, labels = colnames(Auto[-1]),  
      srt = 45, adj = 1, xpd = TRUE)  
  
# Dibujamos los ejes.  
axis(2)
```

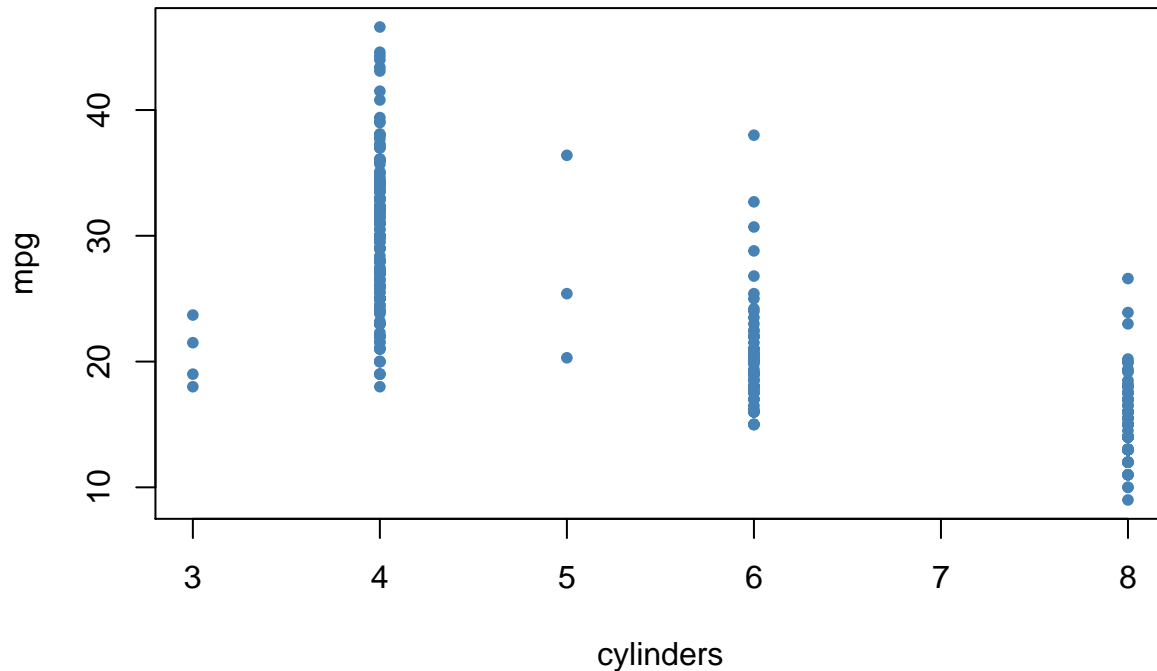
Correlación entre mpg y las demás variables



Como vemos son cylinders, displacement, horsepower y weight las variables que más correlación presentan con mpg. Sin embargo, vamos a estudiar las tres últimas; la correlación entre la primera y mpg es un dato arbitrario que tiene más que ver con la forma de la variable que con la relación entre ambas. Esto se ve claro si ampliamos el gráfico mpg-cylinders:

```
# Plot para mpg-cylinders
plot(cylinders, mpg, pch=20, col="steelblue",
     main="Número de cilindros")
```

Número de cilindros



Seleccionamos las variables escogidas:

```
selec <- c("displacement", "horsepower", "weight")
```

Apartado c

Vamos a crear dos vectores que usaremos para indexar las muestras de entrenamiento y de test:

```
# Vector de índices para la muestra de entrenamiento (80%)
trainIdx <- sample(nrow(Auto), size=0.8*nrow(Auto))

# Vector de índices para la muestra de test
testIdx  <- setdiff(1:nrow(Auto), trainIdx)
```

Apartado d

Creamos la variable booleana mpg01. En vez de usar los valores simbólicos 1 y -1, usamos unos mucho más expresivos: True y False. Una vez añadida la variable, partimos los datos en las muestras de entrenamiento y test con los índices calculados anteriormente:

```
# Creamos una nueva variable booleana, mpg01, en función de la mediana,
# y la añadimos a la base de datos
mpg01 <- ifelse(mpg > median(mpg), T, F)
Auto <- data.frame(mpg01, Auto)

# Obtenemos las muestras de entrenamiento y de test
Auto.train <- Auto[trainIdx,]
Auto.test  <- Auto[testIdx,]
```

Regresión lineal

Para hacer la regresión lineal vamos a usar la función `lm`, que devuelve un modelo lineal $Y \sim X$, donde Y es la variable a predecir y X el conjunto de variables predictoras.

Este modelo lineal lo ajustaremos con la muestra de test:

```
# Ajustamos el modelo con los datos de entrenamiento
mod.lin = lm(mpg01~displacement+horsepower+weight, data=Auto.train)
```

Para ver la efectividad del modelo, intentamos predecir la variable `mpg01` con la función `predict` sobre la muestra de test:

```
# Usamos el modelo lineal ajustado para predecir con la muestra de test
mod.lin.pred <- predict(mod.lin, Auto.test, type = "response")
```

Esto nos devuelve en la variable `Auto.mod.lin.pred` las probabilidades predecidas para cada punto de la muestra de test. Por tanto, vamos a asignar el valor `True` a aquellos puntos donde la probabilidad sea mayor al 50% y `False` a los demás:

```
# Si la predicción tiene probabilidad mayor que el 50%,
# asignamos el valor Verdad; en otro caso, asignamos el valor Falso
mod.lin.mpg01 <- ifelse(mod.lin.pred > 0.5, T, F)
```

Para calcular el error ya basta, tan sólo, contar el porcentaje de puntos mal clasificados en la muestra de test:

```
# Calculamos el error como el porcentaje de muestras mal clasificadas
mod.lin.error <- mean(Auto.test$mpg01 != mod.lin.mpg01)
```

De aquí obtenemos que el error producido por este modelo es del 11.3924051%. Podemos ver exactamente cuántos falsos positivos y falsos negativos tenemos; la siguiente tabla, cuyas columnas son los valores predecidos y cuyas filas los reales, resume la efectividad del método:

```
tab <- table(Real=Auto.test$mpg01, Predicado=mod.lin.mpg01)
kable(tab, caption="Regresión lineal")
```

Table 1: Regresión lineal

	FALSE	TRUE
FALSE	30	7
TRUE	2	40

Vecino más cercano

Para ajustar el modelo de los k vecinos más cercanos necesitamos, primero, normalizar las variables predictoras. Para ello usamos la función `scale()`, que devuelve el conjunto de variables introducido de manera que todas ellas tengan media 0 y varianza 1. Omitimos en este escalado la variable que queremos predecir, `mpg01`, ya que su valor no influirá en las distancias entre los datos, que se mide con las variables predictoras.

El escalado, además lo vamos a hacer de la siguiente manera:

- Escalamos la muestras de entrenamiento.
- Tomamos los valores de medias y varianzas del escalado.
- Escalamos la muestra de test con esos valores.

Esto lo hacemos para que los datos de test no influyan en el proceso de aprendizaje.

```

# Escalado de la muestra de entrenamiento sin la variable mpg01
norm.train <- scale(Auto.train[,-1])

# Escalado de la muestra de test sin mpg01 y en base al escalado anterior
norm.test <- scale(Auto.test[,-1],
                  center = attributes(norm.train)$"scaled:center",
                  scale = attributes(norm.train)$"scaled:scale")

# Creamos una variable con ambas muestras y la reordenamos según el nombre (número) de las filas
norm.full <- rbind(norm.train, norm.test)
norm.full <- norm.full[order(as.numeric(row.names(norm.full))),]

```

La forma más sencilla de ajustar el modelo y predecir con él es llamar a la función `knn` con no más atributos que las muestras de entrenamiento y test y las clases:

```

# Cargamos la librería class, que contiene los modelos del knn
library(class)

# Predecimos de la forma más directa posible y con el típico valor de k=3
knn.pred <- knn(norm.train, norm.test, Auto.train$mpg01, k=3)

# Calculamos el error
knn.error <- mean(Auto.test$mpg01 != knn.pred)

```

Tenemos así un error de 8.8607595%, que mejora al anterior de regresión lineal. Sin embargo, podemos intentar mejorarlo, ajustando el parámetro k a su óptimo. Esta regularización la podemos hacer con la función `tune.knn` y con el método de validación cruzada:

```

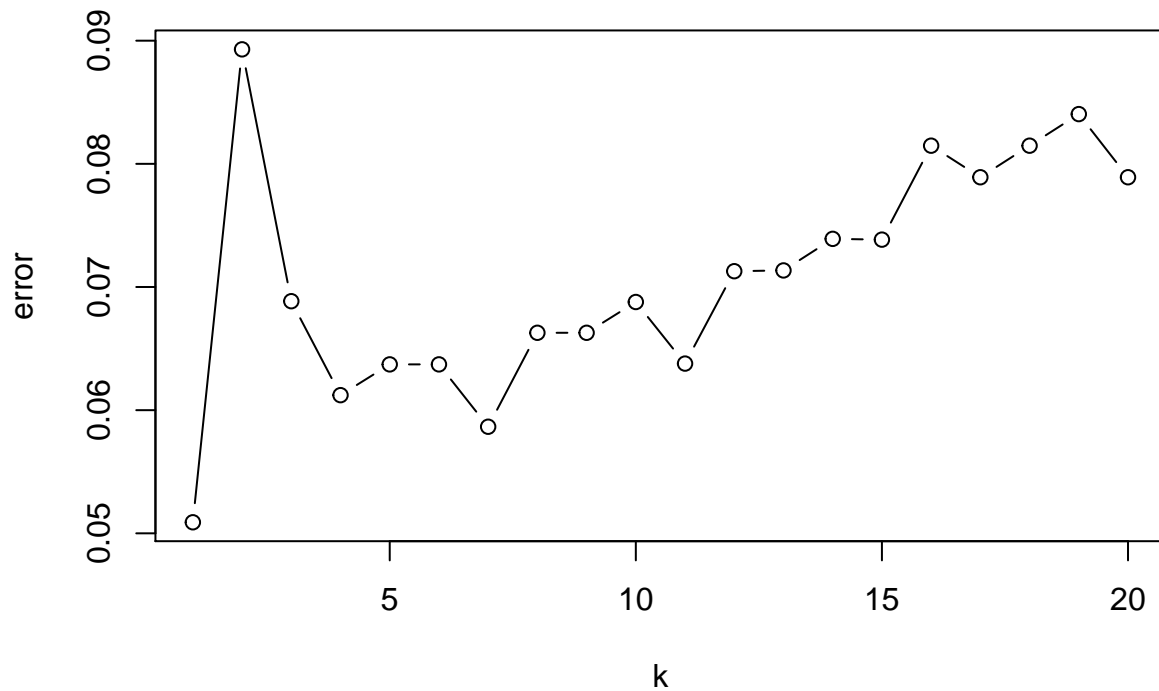
# Necesitamos la librería e1071, que podemos instalar con la orden
# install.packages("e1071")
library(e1071)

# Para la validación cruzada usamos el conjunto completo de datos
# y el conjunto completo de clases
knn.cross <- tune.knn(x = norm.full, y = Auto$mpg01, k = 1:20,
                    tunecontrol = tune.control(sampling = "cross"), cross=10)

# Podemos estudiar visualmente la búsqueda del mejor k:
plot(knn.cross)

```


Performance of 'knn.wrapper'



```
# Nos quedamos con el mejor parámetro  
knn.k <- knn.cross$best.parameters$k
```

Ajustamos de nuevo el modelo con el valor de $k = 1$ obtenido del anterior estudio:

```
# Predecimos con el k calculado  
knn.pred <- knn(norm.train, norm.test, Auto.train$mpg01, k = knn.k)  
  
# Calculamos de nuevo el error  
knn.k.error <- mean(Auto.test$mpg01 != knn.pred)
```

consiguiendo ahora un 7.5949367% de error, 1.2658228 puntos mejor que el anterior.