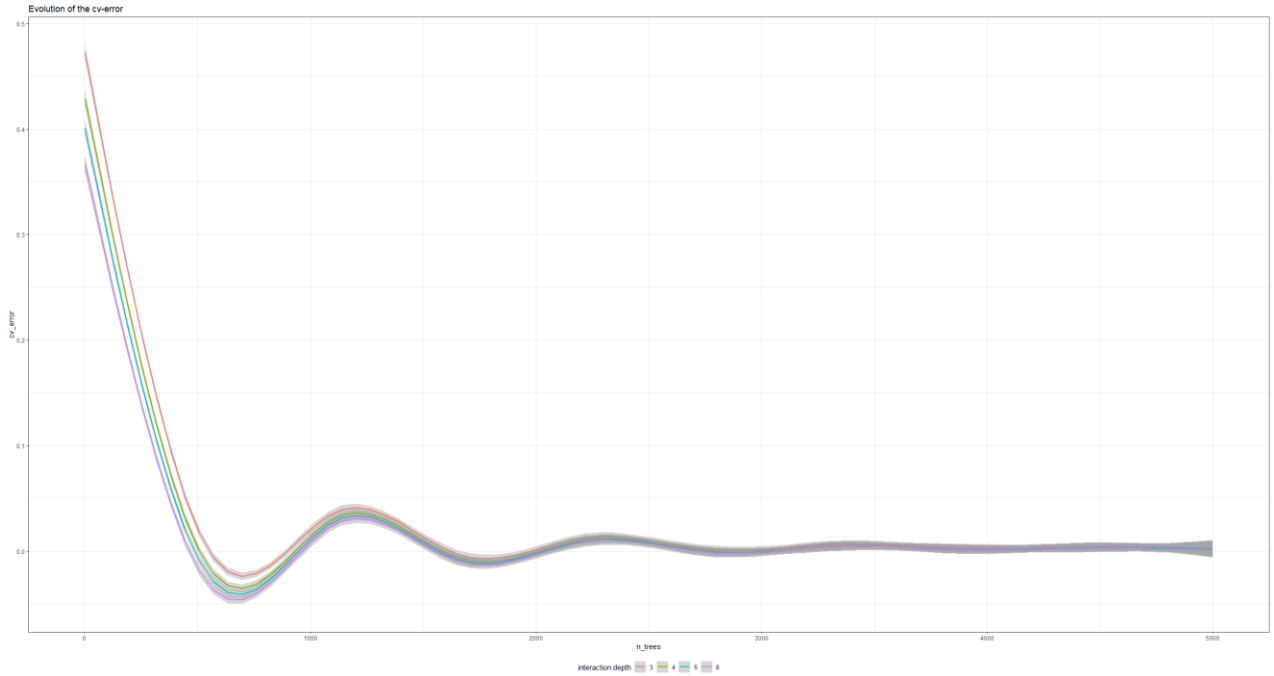


Práctica Decision Trees y Ensembles

MACHINE LEARNING I

Yago Tobio | Profesor: Miguel Ángel Sanz Bobi | 18/11/2021

Introducción

En esta práctica se nos ha encargado seleccionar el dataset del repositorio UCI llamado Letter en el que hay información sobre cajas. El objetivo es predecir la letra asignada en cada caja. Al igual que en la práctica anterior tenemos un dataset con 26 letras, pero en nuestro caso, solo modelaremos casos de la letra X, Y e Z.

Para dicha práctica vamos a usar los siguientes métodos de árboles de decisión y ensembles:

- ID3
- C5.0
- Cart

- Arboles de Regresión
- Bagging
- Random Forest
- Boosting

Preparación de los datos

Cargamos los datos y los filtramos de la siguiente manera:

- Leemos los datos
- Quitamos los NA's
- Sacamos los índices para únicamente las letras X, Y y Z
- Extraemos los sets de entrenamiento y prueba mediante una ratio 80:20

```
data <- read.csv("letter-recognition.data", header = FALSE, sep = ",",
na.strings = "NA")
#Tenemos que sacar las propiedades del archivo de datos
summary(data)
str(data)

#Se nos ha encargado hacer el estudio para los casos de las letras X, Y
y Z.
data <- na.omit(data)

#Por lo tanto, debemos de quitar todos los datos que no sean X, Y o Z
#Paso 1.- Obtener el indice para los valores de X, Y o Z
index_x <- grep("X",data$V1)
index_y <- grep("Y", data$V1)
index_z <- grep("Z", data$V1)
```

```r
index_filter <- c(index_x, index_y, index_z)

#Ahora ya tenemos todos los datos filtrados tal que T solo valdr? X,Y
o Z
data_filtered <- data[index_filter,]


#Al solo tener 3 niveles en T, vamos a convertirlo a una variable
factor
data_filtered$V1 <- as.factor(data_filtered$V1)
str(data_filtered)

names(data_filtered) <- c('lettr','x-box','y-
box','width','high','onpix','x-bar','y-bar'
                          ,'x2bar','y2bar','xybar', 'x2ybr','xy2br','x-
ege','xegvy','y-ege','yegvx')

data_filtered <- data_filtered %>% relocate(lettr, .after = yegvx)

set.seed(123)

#Creation of training and test datasets
index <- sample(nrow(data_filtered), round(0.8*nrow(data_filtered)))
train <- data_filtered[index, ]
test <- data_filtered[-index, ]
train_label <- data_filtered[index, 17]
test_label <- data_filtered[-index, 17]
```

Debemos de comprobar que los conjuntos de Train y Test tengan una proporción similar de cada letra. Esto causará que sea más fiable la validación y que evitemos posible sesgo en la predicción.

```
> table(train$lettr == "X")/nrow(train)

    FALSE      TRUE
0.6576381 0.3423619
> table(test$lettr == "X")/nrow(test)

    FALSE      TRUE
0.6637744 0.3362256
> table(train$lettr == "Y")/nrow(train)

   FALSE     TRUE
0.659805 0.340195
> table(test$lettr == "Y")/nrow(test)

    FALSE      TRUE
0.6572668 0.3427332
> table(train$lettr == "Z")/nrow(train)

    FALSE      TRUE
0.6825569 0.3174431
> table(test$lettr == "Z")/nrow(test)

    FALSE      TRUE
0.6789588 0.3210412
```

Método 1.- ID3

Comenzamos con nuestro primer método para clasificar nuestros datos mediante arboles de decisión. Primero definimos las siguientes funciones:

- Puridad
- Entropía
- Cambio de Entropía

```
################### Pasamos ID3 ###################################
library(data.tree)

#Definimos puridad
IsPure <- function(data){
  length(unique(data[, ncol(data)])) == 1
}

#Definition of entropy
Entropy <- function( vls ) {
  res <- vls/sum(vls) * log2(vls/sum(vls))
  res[vls == 0] <- 0
  -sum(res)
}

Entropy(c(10, 0))
Entropy(c(0, 7))
Entropy(c(3, 7))


#Information Gain
InformationGain <- function( tble ) {
  tble <- as.data.frame.matrix(tble)
  entropyBefore <- Entropy(colSums(tble))
  s <- rowSums(tble)
  entropyAfter <- sum (s / sum(s) * apply(tble,
                                          MARGIN = 1, FUN = Entropy ))
  informationGain <- entropyBefore - entropyAfter
  return (informationGain)
}

#Testeamos Information Gain
InformationGain(table(data_filtered[,c('lettr', 'x-bar')]))
InformationGain(table(data_filtered[, c('lettr', 'y-bar')]))
```

Podemos observar los casos test especificados a continuación, los cuales verifican nuestras funciones, tras haberlas sacado a mano:

```
> Entropy(c(10, 0))
[1] 0
> Entropy(c(0, 7))
[1] 0
> Entropy(c(3, 7))
[1] 0.8812909
> #Testeamos Information Gain
> InformationGain(table(data_filtered[,c('lettr', 'x-bar')]))
[1] 0.169979
> InformationGain(table(data_filtered[, c('lettr', 'y-bar')]))
[1] 0.5386514
```

Construcción del árbol:

Usamos las funciones que se nos han entregado en los documentos del repositorio universitario para formarlos:

```
#ID3 Code
TrainID3 <- function(node, data) {

  node$obsCount <- nrow(data)

  #if the data-set is pure (e.g. all toxic), then
  if (IsPure(data)) {
    #construct a leaf having the name of the pure feature (e.g. 'toxic')
    child <- node$AddChild(unique(data[,ncol(data)]))
    node$feature <- tail(names(data), 1)
    child$obsCount <- nrow(data)
    child$feature <- ''
  } else {
    #chose the feature with the highest information gain (e.g. 'color')
    ig <- sapply(colnames(data)[-ncol(data)],
                 function(x) InformationGain(
                   table(data[,x], data[,ncol(data)])
                 )
    )
    feature <- names(ig)[ig == max(ig)][1]

    node$feature <- feature

    #take the subset of the data-set having that feature value
    childObs <- split(data[,!(names(data) %in% feature)], data[,feature],
drop = TRUE)

    for(i in 1:length(childObs)) {
```

```
      #construct a child having the name of that feature value (e.g.
'red')
      child <- node$AddChild(names(childObs)[i])

      #call the algorithm recursively on the child and the subset
      TrainID3(child, childObs[[i]])
    }
  }
}

#.Debemos de tener en cuenta que dicha variable que deseamos clasificar
#Debe de ser la ultima en nuestro dataset
#Tras esto, pasamos a construir el arbol ID3 para predecir la variable
lettr
treeID3 <- Node$new("lettr")
TrainID3(treeID3, train)
print(treeID3, "feature", "obsCount")
# Prediction function
Predict <- function(tree, features) {
  if (tree$children[[1]]$isLeaf) return (tree$children[[1]]$name)
  child <- tree$children[[features[[tree$feature]]]]
  return ( Predict(child, features))
}
```

El resultado de haber montado nuestro árbol usando ID3 se puede visualizar a continuación:

```
                                levelName feature obsCount
1   lettr                                 y-ege     1846
2    |--0                                 lettr      176
3    |   °--Y                                        176
4    |--1                                 lettr       37
5    |   °--Y                                         37
6    |--2                                 x2ybr      174
7    |   |--3                             lettr        2
8    |   |   °--X                                     2
9    |   |--5                             lettr        1
10   |   |   °--X                                     1
11   |   |--6                             lettr        1
12   |   |   °--X                                     1
13   |   |--7                             lettr        1
14   |   |   °--X                                     1
15   |   |--9                             lettr        1
16   |   |   °--Y                                     1
17   |   |--10                            lettr       29
18   |   |   °--Y                                    29
19   |   |--11                            lettr       87
20   |   |   °--Y                                    87
21   |   |--12                            lettr       49
22   |   |   °--Y                                    49
23   |   °--13                            lettr        3
24   |       °--Y                                     3
25   |--3                                 xegvy      191
26   |   |--7                             lettr       11
27   |   |   °--X                                    11
28   |   |--8                             lettr      104
29   |   |   °--X                                    104
30   |   |--9                             y2bar       23
31   |   |   |--2                         lettr        1
32   |   |   |   °--Y                                 1
33   |   |   |--3                         lettr        1
34   |   |   |   °--Y                                 1
35   |   |   |--6                         lettr        1
36   |   |   |   °--Y                                 1
37   |   |   |--7                         lettr        4
38   |   |   |   °--X                                 4
39   |   |   |--8                         lettr       15
40   |   |   |   °--X                                15
41   |   |   °--9                         lettr        1
42   |   |       °--X                                 1
43   |   |--10                            lettr       16
44   |   |   °--Y                                    16
```

```
|   |--11                                x-bar       31
|   |   |--3                             lettr       10
|   |   |   *--Y                                      10
|   |   |--4                             lettr       13
|   |   |   *--Y                                      13
|   |   |--5                             lettr        5
|   |   |   *--Y                                       5
|   |   |--6                             lettr        1
|   |   |   *--Y                                       1
|   |   |--7                             lettr        1
|   |   |   *--Y                                       1
|   |   *--9                             lettr        1
|   |       *--X                                       1
|   *--12                                width        6
|       |--4                             lettr        1
|       |   *--Y                                       1
|       |--6                             lettr        2
|       |   *--X                                       2
|       |--7                             lettr        1
|       |   *--Y                                       1
|       *--8                             lettr        2
|           *--Y                                       2
*--4                                     yegvx      308
    |--3                                 lettr       16
    |   *--Y                                         16
    |--4                                 x-bar       22
    |   |--3                             lettr        1
    |   |   *--X                                       1
    |   |--4                             lettr        1
    |   |   *--Y                                       1
    |   |--5                             lettr        8
    |   |   *--Y                                       8
    |   |--6                             lettr        8
    |   |   *--Y                                       8
    |   |--7                             lettr        3
    |   |   *--Y                                       3
    |   *--8                             lettr        1
    |       *--Y                                       1
    |--5                                 xy2br       16
    |   |--5                             lettr        1
    |   |   *--Y                                       1
    |   |--6                             lettr        2
    |   |   *--Y                                       2
    |   |--8                             lettr        1
    |   |   *--Y                                       1
    |   |--9                             lettr        6
    |   |   *--X                                       6
    |   |--10                            lettr        5
    |   |   *--X                                       5
    |   *--11                            lettr        1
    |       *--X                                       1
    |--6                                 y-bar       22
    |   |--4                             lettr        1
    |   |   *--Y                                       1
    |   |--5                             lettr        1
    |   |   *--... 1 nodes w/ 0 sub                   NA
    |   *--... 4 nodes w/ 5 sub                       NA
    *--... 5 nodes w/ 41 sub                          NA
*--... 10 nodes w/ 504 sub                            NA
```

Tras haber montado este árbol, vamos a pasar a la siguiente técnica de árboles: C5.0 el cual nos dará la opción de podar.

Algoritmo II.- Método C5.0

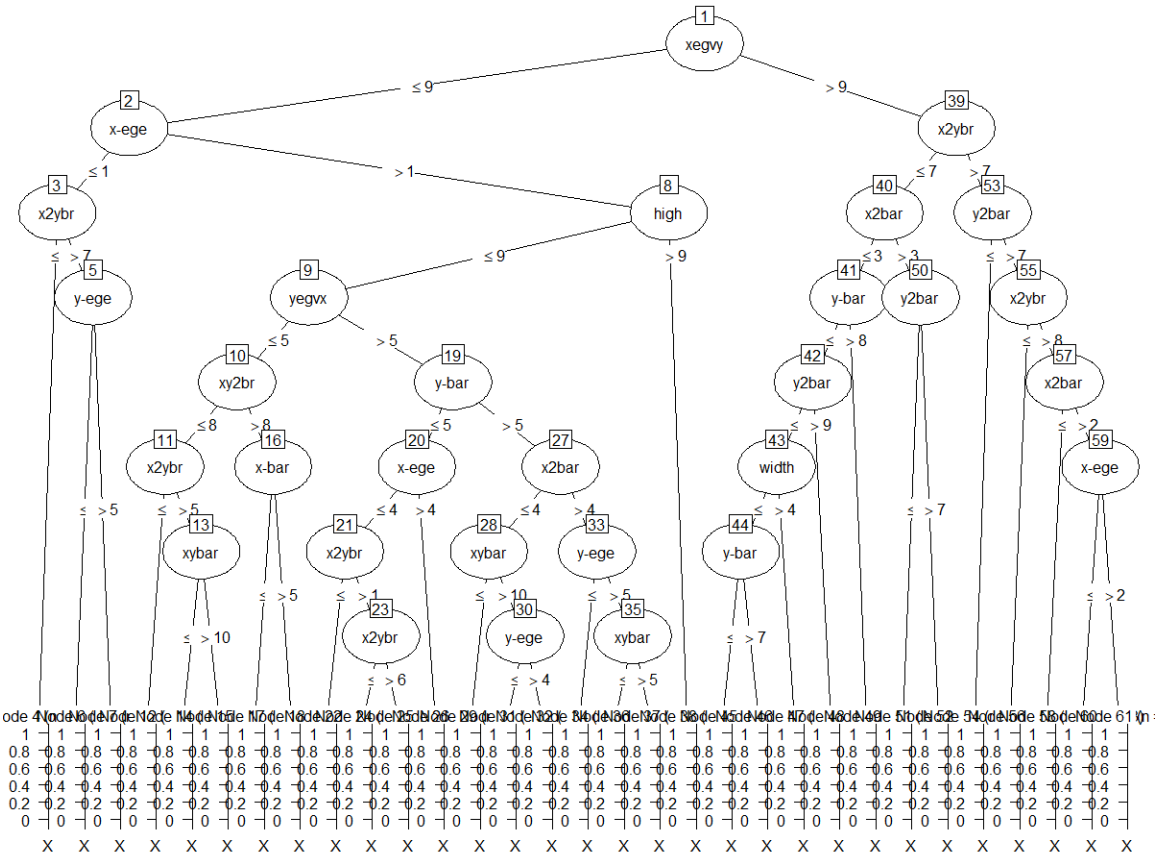
Construimos el árbol usando código en R y sus librerías correspondientes:

```
########## Procedemos CON C50 ####################################2
library(C50)

tree_C50 <- C5.0(lettr ~ ., data = train,
                 control = C5.0Control(noGlobalPruning = FALSE, CF =
0.25))
# Observamos que si que deseamos podar, y nuestro factor es de 0.25
# Cuanto mas alto CF, menos podaremos

summary(tree_C5.0)
```

```
C5.0 [Release 2.07 GPL Edition]         Fri Nov 19 08:26:40 2021
-------------------------------

Class specified by attribute `outcome'

Read 1846 cases (17 attributes) from undefined.data

Decision tree:

xegvy > 9:
:...x2ybr > 7:
:   :...y2bar <= 7: Y (471)
:   :   y2bar > 7:
:   :   :...x2ybr <= 8: Z (6/1)
:   :       x2ybr > 8:
:   :       :...x2bar <= 2: Y (62)
:   :           x2bar > 2:
:   :           :...x-ege <= 2: Y (3)
:   :               x-ege > 2: X (3)
:   x2ybr <= 7:
:   :...x2bar > 3:
:       :...y2bar <= 7: Y (38/1)
:       :   y2bar > 7: X (2/1)
:       x2bar <= 3:
:       :...y-bar > 8: Z (2)
:           y-bar <= 8:
:           :...y2bar > 9: Z (2)
:               y2bar <= 9:
:               :...width > 4: X (44)
:                   width <= 4:
:                   :...y-bar <= 7: X (2)
:                       y-bar > 7: Y (2)
xegvy <= 9:
:...x-ege <= 1:
    :...x2ybr <= 7: Z (388)
    :   x2ybr > 7:
    :   :...y-ege <= 5: Y (2)
    :       y-ege > 5: Z (12)
    x-ege > 1:
    :...high > 9: Z (37/1)
        high <= 9:
        :...yegvx <= 5:
            :...xy2br > 8:
            :   :...x-bar <= 5: X (28/1)
            :   :   x-bar > 5: Z (3)
            :   xy2br <= 8:
            :   :...x2ybr <= 5: Z (25/1)
            :       x2ybr > 5:
            :       :...xybar <= 10: Y (40)
            :           xybar > 10: Z (6)
            yegvx > 5:
            :...y-bar <= 5:
                :...x-ege > 4: Y (4)
                :   x-ege <= 4:
                :   :...x2ybr <= 1: X (5)
                :       x2ybr > 1:
                :       :...x2ybr <= 6: Z (30/1)
                :           x2ybr > 6: X (4/1)
                y-bar > 5:
                :...x2bar <= 4:
                    :...xybar <= 10: X (458/1)
                    :   xybar > 10:
                    :   :...y-ege <= 4: X (47/1)
                    :       y-ege > 4: Z (30)
                    x2bar > 4:
                    :...y-ege <= 5: X (42)
                        y-ege > 5:
                        :...xybar <= 5: Y (2)
                            xybar > 5: Z (46)

                Evaluation on training data (1846 cases):

                        Decision Tree
                      ----------------
                      Size      Errors

                        31   10( 0.5%)   <<


                       (a)   (b)   (c)    <-classified as
                      ----  ----  ----
                       630           2    (a): class X
                         3   623     2    (b): class Y
                         2     1   583    (c): class Z


                Attribute usage:

                100.00% xegvy
                 65.82% x-ege
                 62.24% x2ybr
                 43.72% high
                 42.52% x2bar
                 41.71% yegvx
                 39.00% y-bar
                 34.40% y2bar
                 34.07% xybar
                  9.80% y-ege
                  5.53% xy2br
                  2.60% width
                  1.68% x-bar
```

Por la función de summary, podemos observar la metodología sobre como se ha construido el árbol. Se pueden observar también el nivel de error en la matriz de confusión final en el cual se han clasificado incorrectamente las clases. Un 0.5% diría que es un error aceptable para nuestro modelo.

Para poder visualizar nuestro árbol C5.0 vamos a hacer uso de la función plot, la cual nos mostrará a su vez el grado de complejidad y poda:



Test de Precisión del árbol:

```
predictions <- predict(tree_C50, newdata = test, type = "class")
table(prediction = predictions, real = test$lettr)
error_class <- mean(predictions != test$lettr)
paste("Classification error is: ", 100*error_class, "%")
```

```
> predictions <- predict(tree_C50, newdata = test, type = "class")
> table(prediction = predictions, real = test$lettr)
          real
prediction   X   Y   Z
         X 151   2   0
         Y   3 153   1
         Z   1   3 147
> error_class <- mean(predictions != test$lettr)
> paste("Classification error is: ", 100*error_class, "%")
[1] "Classification error is:  2.16919739696312 %"
```

Podemos observar que, aunque nuestro modelo sea ultra-preciso, el nivel de complejidad es demasiado alto, observamos que las hojas finales que cubre la imagen solo llegan a mostrar las cajas que contienen la letra X.

Reglas:

Antes de proceder a observar las siguientes variables, debemos observar las reglas de las clasificaciones para ver de que criterios depende cada clase, la cual en R se demuestra de la siguiente manera:

```
ruleModel <- C5.0(lettr ~., data = train, rules = TRUE)

summary(ruleModel)
```

```
> ruleModel <- C5.0(lettr ~., data = train, rules = TRUE)
> ruleModel

Call:
C5.0.formula(formula = lettr ~ ., data = train, rules = TRUE)

Rule-Based Model
Number of samples: 1846
Number of predictors: 16

Number of Rules: 19

Non-standard options: attempt to group attributes

> summary(ruleModel)

Call:
C5.0.formula(formula = lettr ~ ., data = train, rules = TRUE)


C5.0 [Release 2.07 GPL Edition]         Fri Nov 19 08:58:16 2021
-------------------------------

Class specified by attribute `outcome'

Read 1846 cases (17 attributes) from undefined.data

Rules:

Rule 1: (44, lift 2.9)
        width > 4
        y-bar <= 8
        x2bar <= 3
        y2bar <= 9
        x2ybr <= 7
        xegvy > 9
        ->  class X  [0.978]

Rule 2: (46/2, lift 2.7)
        x-bar <= 5
        xy2br > 8
        xegvy <= 9
        ->  class X  [0.938]

Rule 3: (14, lift 2.7)
        x2bar > 2
        y2bar > 7
        x2ybr > 8
        x-ege > 2
        ->  class X  [0.938]

Rule 4: (668/114, lift 2.4)
        high <= 9
        x-ege > 1
        xegvy <= 9
        yegvx > 5
        ->  class X  [0.828]

Rule 5: (92/45, lift 1.5)
        x2ybr <= 7
        xegvy > 9
        ->  class X  [0.511]

Rule 6: (471, lift 2.9)
        y2bar <= 7
        x2ybr > 7
        xegvy > 9
        ->  class Y  [0.998]

Rule 7: (362, lift 2.9)
        x2bar <= 2
        x2ybr > 8
        xegvy > 9
        ->  class Y  [0.997]

Rule 8: (191, lift 2.9)
        width <= 4
        y-bar > 7
        xegvy > 9
        ->  class Y  [0.995]

Rule 9: (106/1, lift 2.9)
        x2bar > 3
        y2bar <= 7
        xegvy > 9
        ->  class Y  [0.981]

Rule 10: (16, lift 2.8)
        y-bar <= 5
        x-ege > 4
        ->  class Y  [0.944]

Rule 11: (13, lift 2.7)
        x2bar > 4
        xybar <= 5
        ->  class Y  [0.933]

Rule 12: (308/69, lift 2.3)
        high <= 9
        yegvx <= 5
        ->  class Y  [0.774]

Rule 13: (388, lift 3.1)
        x2ybr <= 7
        x-ege <= 1
        ->  class Z  [0.997]

Rule 14: (220, lift 3.1)
        xybar > 10
        xegvy <= 9
        y-ege > 4
        ->  class Z  [0.995]

Rule 15: (136, lift 3.1)
        x2bar > 4
        xybar > 5
        xegvy <= 9
        y-ege > 5
        yegvx > 5
        ->  class Z  [0.993]

Rule 16: (57/1, lift 3.0)
        y-bar <= 5
        x2ybr > 1
        x2ybr <= 6
        ->  class Z  [0.966]

Rule 17: (26/1, lift 2.9)
        x2ybr <= 5
        yegvx <= 5
        ->  class Z  [0.929]

Rule 18: (10/2, lift 2.4)
        y-bar > 8
        x2bar <= 3
        x2ybr <= 7
        ->  class Z  [0.750]

Rule 19: (760/295, lift 1.9)
        y2bar > 7
        x2ybr <= 8
        ->  class Z  [0.612]

Default class: Z
```

```
Evaluation on training data (1846 cases):

        Rules
  ----------------
    No      Errors

    19    7( 0.4%)   <<


   (a)   (b)   (c)    <-classified as
  ----  ----  ----
   629           3    (a): class X
     2   625     1    (b): class Y
           1   585    (c): class Z


Attribute usage:

 85.16% xegvy
 79.09% x2ybr
 70.96% y2bar
 58.50% x-ege
 57.69% yegvx
 52.87% high
 36.84% x2bar
 19.34% xybar
 18.63% y-ege
 17.23% y-bar
 12.73% width
  2.49% x-bar
  2.49% xy2br
```

Evaluación de nuestro modelo C5.0 y la importancia consecuente de cada clasificador

Algoritmo III .- CART

```
library(rpart)
library(rpart.plot)
library(caret)

tree_CART <- rpart(formula = lettr ~ ., data = train, method = 'class')
print(tree_CART)
summary(tree_CART)
```

```
> print(tree_CART)
n= 1846

node), split, n, loss, yval, (yprob)
      * denotes terminal node

  1) root 1846 1214 X (0.342361863 0.340195016 0.317443120)
    2) xegvy>=9.5 637   62 Y (0.080062794 0.902668760 0.017268446)
      4) x2ybr< 6.5 36    8 X (0.777777778 0.055555556 0.166666667) *
      5) x2ybr>=6.5 601   28 Y (0.038269551 0.953410982 0.008319468) *
    3) xegvy< 9.5 1209  628 X (0.480562448 0.043837883 0.475599669)
      6) x-ege>=1.5 807  226 X (0.719950434 0.063197026 0.216852540)
       12) y-ege< 4.5 389   16 X (0.958868895 0.033419023 0.007712082) *
       13) y-ege>=4.5 418  210 X (0.497607656 0.090909091 0.411483254)
         26) xybar< 8.5 313  107 X (0.658146965 0.105431310 0.236421725)
           52) x2bar< 4.5 215    9 X (0.958139535 0.009302326 0.032558140) *
           53) x2bar>=4.5 98   31 Z (0.000000000 0.316326531 0.683673469)
            106) x2ybr>=7.5 27    0 Y (0.000000000 1.000000000 0.000000000) *
            107) x2ybr< 7.5 71    4 Z (0.000000000 0.056338028 0.943661972) *
         27) xybar>=8.5 105    7 Z (0.019047619 0.047619048 0.933333333) *
      7) x-ege< 1.5 402    2 Z (0.000000000 0.004975124 0.995024876) *
```

Podemos observar el orden de elección de los separadores. A continuación, el summary nos permitirá obtener mas datos como los valores críticos de coste complejidad y la

importancia de cada variable además de un resumen detallado de como se van dividiendo las hojas del árbol:

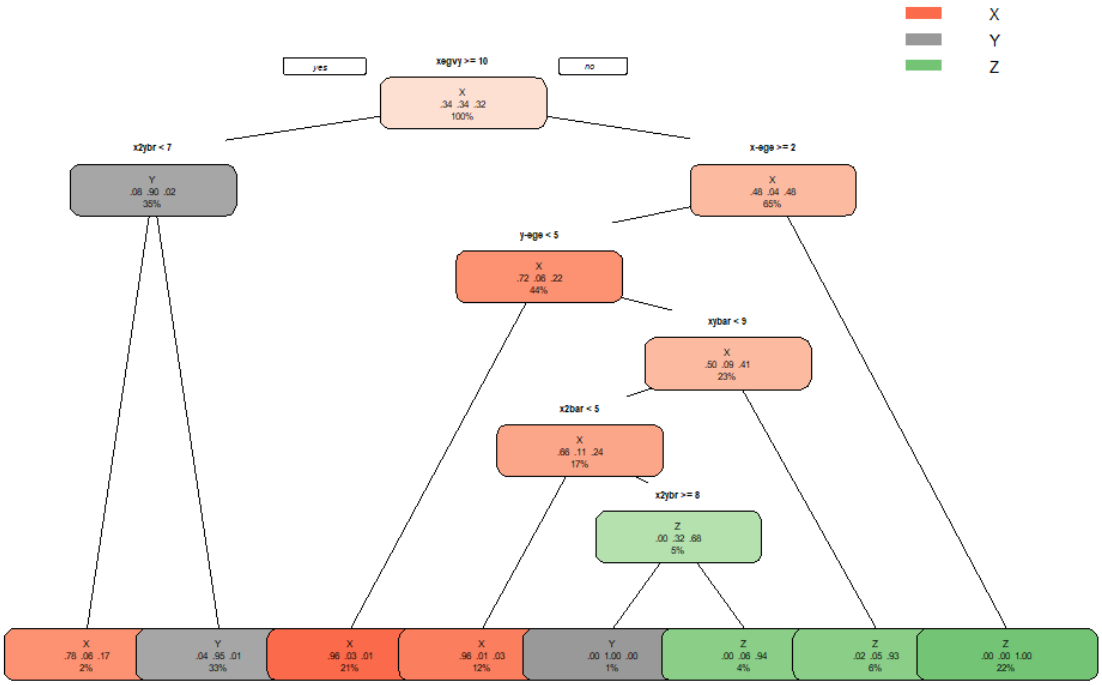
```
#Call:
#rpart(formula = lettr ~ ., data = train, method = "class")
#  n= 1846
#
#          CP nsplit  rel error     xerror        xstd
#1 0.43163097      0 1.00000000 1.01894563 0.016640216
#2 0.32948929      1 0.56836903 0.59637562 0.017279500
#3 0.03953871      2 0.23887974 0.23887974 0.012878618
#4 0.02224053      5 0.10461285 0.11037891 0.009182686
#5 0.02141680      6 0.08237232 0.09390445 0.008519063
#6 0.01000000      7 0.06095552 0.06919275 0.007375782
#
#Variable importance
#xegvy x2ybr y2bar y-ege y-bar x-ege x-bar x2bar xybar yegvx x-box onpix  high y-box
#   16    15    12    12    12    11     5     5     5     2     1     1     1     1
#
#Node number 1: 1846 observations,    complexity param=0.431631
#  predicted class=X  expected loss=0.6576381  P(node) =1
#    class counts:   632   628   586
#   probabilities: 0.342 0.340 0.317
#  left son=2 (637 obs) right son=3 (1209 obs)
#  Primary splits:
#      xegvy < 9.5  to the right, improve=462.2709, (0 missing)
#      x2ybr < 7.5  to the right, improve=409.5985, (0 missing)
#      y-bar < 8.5  to the left,  improve=360.9242, (0 missing)
#      y-ege < 2.5  to the right, improve=307.3850, (0 missing)
#      y2bar < 8.5  to the left,  improve=242.2401, (0 missing)
#  Surrogate splits:
#      y-bar < 8.5  to the right, agree=0.887, adj=0.673, (0 split)
#      x2ybr < 9.5  to the right, agree=0.887, adj=0.672, (0 split)
#      y-ege < 2.5  to the left,  agree=0.856, adj=0.582, (0 split)
#      y2bar < 3.5  to the left,  agree=0.769, adj=0.330, (0 split)
#      x-bar < 5.5  to the left,  agree=0.727, adj=0.209, (0 split)
#
#Node number 2: 637 observations,    complexity param=0.0214168
#  predicted class=Y  expected loss=0.09733124  P(node) =0.3450704
#    class counts:    51   575    11
#   probabilities: 0.080 0.903 0.017
#  left son=4 (36 obs) right son=5 (601 obs)
#  Primary splits:
```

Por hacer la practica más breve, vamos a dejar la salida de la terminal de R así, pero indicamos que el comando de summary viene con mucha más información. Si se desea observar al completo, es tan simple como copiar y pegar el código de R mostrado en una terminal.

Aun así, creo que la manera más eficiente de visualizar los datos, es mediante un diagrama de árboles, el cual se muestra a continuación:

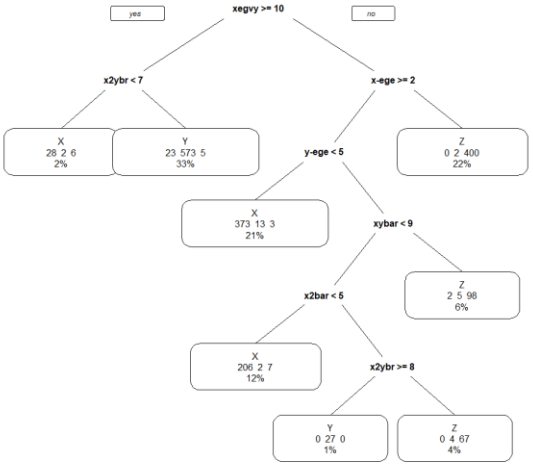
Visualización de Arboles:

```
rpart.plot(tree_CART, type = 1, branch = 0, tweak = 1.55, fallen.leaves =
TRUE,
           varlen = 0, faclen = 0)
```



Hemos de comentar que los datos en este árbol están normalizados por clase, como se puede observar, todos los valores están entre 0 y 1, dependiendo de la abundancia de cada clase por nodo. Por lo cual, presentamos una alternativa en las que aparecen los valores actuales.

```
prp(tree_CART, faclen = 3, clip.facs = TRUE, split.fun = split.fun, tweak
= 1.2, extra = 101)
```



Reglas de CART:

```
rpart.rules(tree_CART, style = "tall", cover = TRUE, nn = TRUE, clip.facs = TRUE)
```

```
> rpart.rules(tree_CART, style = "tall", cover = TRUE, nn = TRUE, clip.facs = TRUE)
                X    Y    Z
[4]   lettr is X [ .78  .06  .17] with cover 2% when
          xegvy >= 10
          x2ybr < 7

[52]  lettr is X [ .96  .01  .03] with cover 12% when
          xegvy < 10
          x-ege >= 2
          y-ege >= 5
          xybar < 9
          x2bar < 5

[12]  lettr is X [ .96  .03  .01] with cover 21% when
          xegvy < 10
          x-ege >= 2
          y-ege < 5

[5]   lettr is Y [ .04  .95  .01] with cover 33% when
          xegvy >= 10
          x2ybr >= 7

[106] lettr is Y [ .00 1.00  .00] with cover 1% when
          xegvy < 10
          x-ege >= 2
          y-ege >= 5
          x2ybr >= 8
          xybar < 9
          x2bar >= 5

[27]  lettr is Z [ .02  .05  .93] with cover 6% when
          xegvy < 10
          x-ege >= 2
          y-ege >= 5
          xybar >= 9

[107] lettr is Z [ .00  .06  .94] with cover 4% when
          xegvy < 10
          x-ege >= 2
          y-ege >= 5
          x2ybr < 8
          xybar < 9
          x2bar >= 5

[7]   lettr is Z [ .00  .00 1.00] with cover 22% when
          xegvy < 10
          x-ege < 2
```

Prueba de predicción:

```
#Prediction of the training cases from its respective dataset
pred_train <- predict(tree_CART, newdata = train, type = "class")
caret::confusionMatrix(pred_train, train$lettr)
statsFromConfusionMatrix(confusionMatrix(pred_train, train$lettr))
```

```
> caret::confusionMatrix(pred_train, train$lettr)
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 607  17  16
         Y  23 600   5
         Z   2  11 565

Overall Statistics

               Accuracy : 0.9599
                 95% CI : (0.9499, 0.9684)
    No Information Rate : 0.3424
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9398

 Mcnemar's Test P-Value : 0.002853

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            0.9604   0.9554   0.9642
Specificity            0.9728   0.9770   0.9897
Pos Pred Value         0.9484   0.9554   0.9775
Neg Pred Value         0.9793   0.9770   0.9834
Prevalence             0.3424   0.3402   0.3174
Detection Rate         0.3288   0.3250   0.3061
Detection Prevalence   0.3467   0.3402   0.3131
Balanced Accuracy      0.9666   0.9662   0.9769
```

Matriz de confusión basada en el set de entrenamiento

Observamos que nuestro set de entrenamiento consigue entrenar el modelo de una manera muy eficiente con una precisión del 96%. ¿Pero cómo de bien se adapta a la predicción con los datos de prueba?

```
pred_test <- predict(tree_CART, newdata = test, type = "class")
caret::confusionMatrix(pred_test, test$lettr)
```

```
> caret::confusionMatrix(pred_test, test$lettr)
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 145   8   3
         Y   9 146   0
         Z   1   4 145

Overall Statistics

               Accuracy : 0.9458
                 95% CI : (0.921, 0.9646)
    No Information Rate : 0.3427
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9186

 Mcnemar's Test P-Value : 0.1675

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            0.9355   0.9241   0.9797
Specificity            0.9641   0.9703   0.9840
Pos Pred Value         0.9295   0.9419   0.9667
Neg Pred Value         0.9672   0.9608   0.9904
Prevalence             0.3362   0.3427   0.3210
Detection Rate         0.3145   0.3167   0.3145
Detection Prevalence   0.3384   0.3362   0.3254
Balanced Accuracy      0.9498   0.9472   0.9819
```

Vemos que para el modelo y las predicciones orientadas en el dataset de prueba tienen una precisión menor del 94.58% pero aun así declararíamos que tenemos un modelo de clasificación muy efectivo.

Podar con CART:

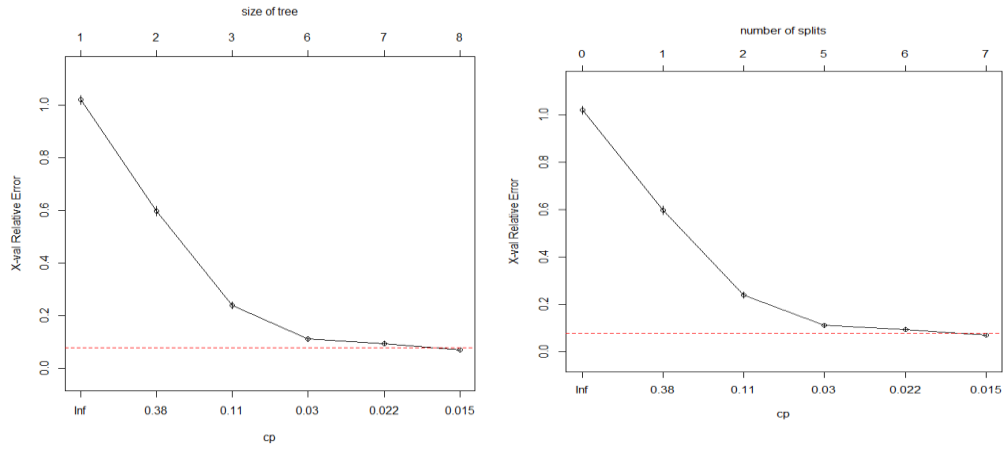
La primera tarea para poder podar este árbol es observar cual será el valor optimo para el coste de complejidad. El cual se determinará de la siguiente manera:

```
> printcp(tree_CART, digits = 4)

Classification tree:
rpart(formula = lettr ~ ., data = train, method = "class")

Variables actually used in tree construction:
[1] x-ege x2bar x2ybr xegvy xybar y-ege

Root node error: 1214/1846 = 0.6576

n= 1846

        CP nsplit rel error  xerror     xstd
1 0.43163      0   1.00000 1.01895 0.016640
2 0.32949      1   0.56837 0.59638 0.017280
3 0.03954      2   0.23888 0.23888 0.012879
4 0.02224      5   0.10461 0.11038 0.009183
5 0.02142      6   0.08237 0.09390 0.008519
6 0.01000      7   0.06096 0.06919 0.007376
```

Observamos que el coste de complejidad optimo va a ser cp = 0.2048

El valor optimo se escoge observando la evolución del error relativo basado en el numero de splits y el tamaño del arbol antes del límite de podar:

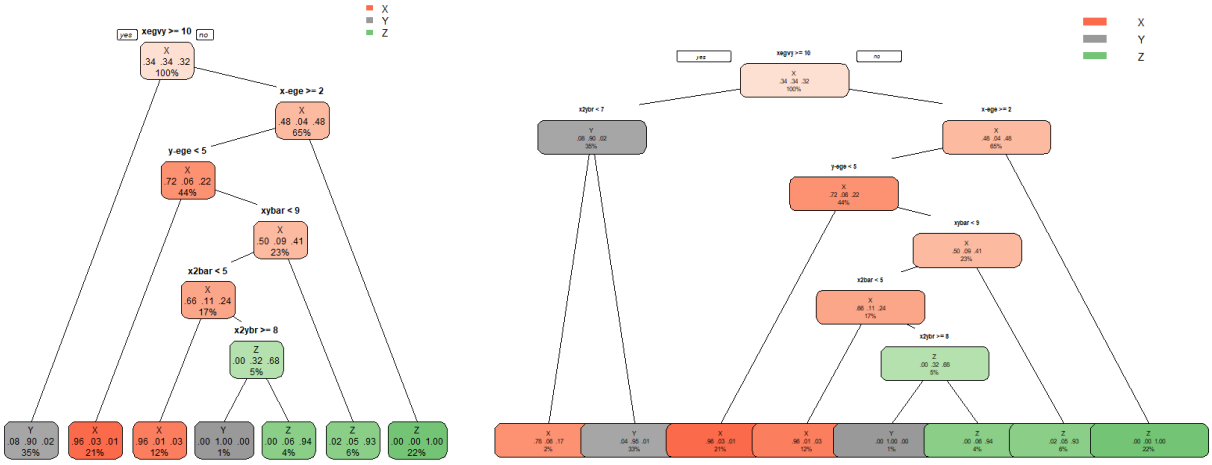
```
printcp(tree_CART, digits = 4)
plotcp(tree_CART, lty = 2, col = "red", upper = "size")
plotcp(tree_CART, lty = 2, col = "red", upper = "splits")
```



Observamos que el límite mínimo del valor de CP antes de podar es justo 0.224

Análisis y visualización al podar:

```
#Pruning analysis con 0.02224
tree_pruned <- prune(tree_CART, cp = 0.02224)
rpart.plot(tree_pruned, type = 1, branch = 0, tweak = 1,
           fallen.leaves = TRUE, varlen = 0, faclen = 0)
```



Comparación del árbol podado (izquierda) y el árbol sin podar (derecha). La diferencia principal entre estos dos es el hecho de que el árbol *x2ybr < 7* se ha podado completamente y se ha dejado como una hoja única.

Predicción con los datasets de entrenamiento y test:

```
pred_train <- predict(tree_pruned, newdata =  train, type = "class")
caret::confusionMatrix(pred_train, train$lettr)
error_class <- mean(pred_train != train != lettr)
error_class
predictions <- predict(tree_pruned, newdata = test, type =
"class")caret::confusionMatrix(predictions, test$lettr)
```

Observamos a continuación las matrices de confusión del set de entrenamiento y del test respectivamente (con sus errores de clasificación):

```
> pred_train <- predict(tree_pruned, newdata =  train, type = "class")
> caret::confusionMatrix(pred_train, train$lettr)
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 579  15  10
         Y  51 602  11
         Z   2  11 565

Overall Statistics

               Accuracy : 0.9458
                 95% CI : (0.9345, 0.9557)
    No Information Rate : 0.3424
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9187

 Mcnemar's Test P-Value : 1.567e-05

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            0.9161   0.9586   0.9642
Specificity            0.9794   0.9491   0.9897
Pos Pred Value         0.9586   0.9066   0.9775
Neg Pred Value         0.9573   0.9780   0.9834
Prevalence             0.3424   0.3402   0.3174
Detection Rate         0.3137   0.3261   0.3061
Detection Prevalence   0.3272   0.3597   0.3131
Balanced Accuracy      0.9478   0.9538   0.9769
```

```
> predictions <- predict(tree_pruned, newdata = test, type = "class")
> caret::confusionMatrix(predictions, test$lettr)
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 144   6   2
         Y  10 148   1
         Z   1   4 145

Overall Statistics

               Accuracy : 0.9479
                 95% CI : (0.9235, 0.9664)
    No Information Rate : 0.3427
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9219

 Mcnemar's Test P-Value : 0.3715

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            0.9290   0.9367   0.9797
Specificity            0.9739   0.9637   0.9840
Pos Pred Value         0.9474   0.9308   0.9667
Neg Pred Value         0.9644   0.9669   0.9904
Prevalence             0.3362   0.3427   0.3210
Detection Rate         0.3124   0.3210   0.3145
Detection Prevalence   0.3297   0.3449   0.3254
Balanced Accuracy      0.9514   0.9502   0.9819
```

Se puede ver que curiosamente en este caso, el dataset de prueba acaba teniendo una precisión ligeramente mas alta que la de entrenamiento por un 0.2%, aún así, este modelo promete mucho para clasificar las letras correctamente.

PARTE II DE LA PRACTICA – ENSEMBLES

Para esta siguiente sección visitaremos las distintas técnicas de clasificación que involucra ensembles:

- Bagging
- Random Forests
- Boosting

Preparación de los datos

```
bagging_model<- randomForest(formula=lettr~ ., data= train,
mtry=16)  #16 are all the predictors
#Result of bagging model
print(bagging_model)
```

Observamos que nuestro modelo de Bagging tiene un error de un 1.36%

```
        OOB estimate of  error rate: 1.36%
Confusion matrix:
    X   Y   Z class.error
X 617   8   3  0.01751592
Y   6 618   2  0.01277955
Z   3   3 585  0.01015228
```

Predicciones con el set de entrenamiento y el de prueba:

```
pred_train <- predict(bagging_model, newdata = train)
confusion_matrix<-table(train$lettr,pred_train, dnn=c("observations",
"predictions"))
confusion_matrix

pred_test <- predict(bagging_model, newdata = test)
confusion_matrix<-table(test$lettr,pred_test, dnn=c("observations", "predictions"))

confusion_matrix
```

```
> confusion_matrix
             predictions
observations   X   Y   Z
           X 628   0   0
           Y   0 626   0
           Z   0   0 591
```

```
> confusion_matrix
             predictions
observations   X   Y   Z
           X 159   0   0
           Y   1 158   1
           Z   0   0 143
```

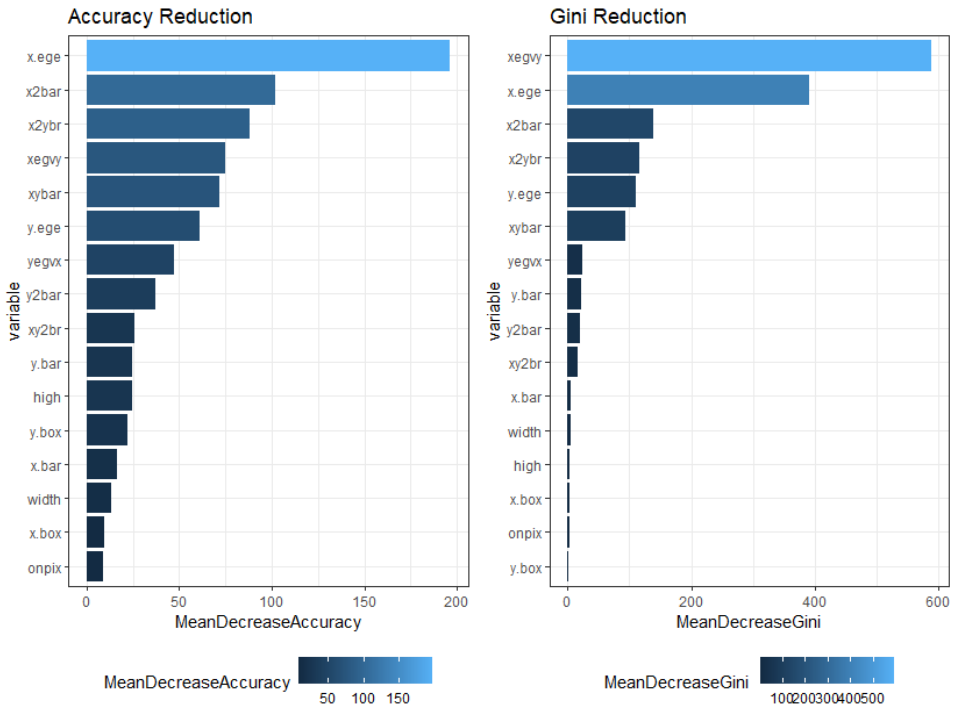
Observamos las matrices de confusión de entrenamiento y testing respectivamente. Mientras el conjunto de entrenamiento parece ser que lo clasifica todo bien, el modelo basado en el set de verificación tiene una tasa de error muy baja. Para observarlo totalmente, vamos a generar el modelo bagging global (con todo el dataset):

```
Call:
 randomForest(formula = lettr ~ ., data = letterData, mtry = 16,      importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 16

        OOB estimate of  error rate: 0.95%
Confusion matrix:
    X   Y   Z class.error
X 778   5   4 0.011435832
Y   7 777   2 0.011450382
Z   2   2 730 0.005449591
```

Tenemos menos de un 1% de error. Ahora que sabemos esto, vamos a analizar mediante los métodos de reducción de Gini y Exactitud para observar los clasificadores más importantes del modelo:

```
importance_pred <- as.data.frame(importance(bagging_model_fulldataset, scale = TRUE))
importance_pred <- rownames_to_column(importance_pred, var = "variable")
```

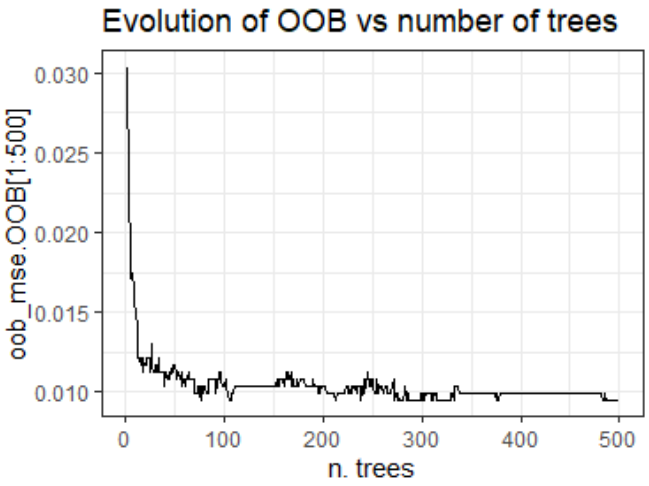
Para el resto de código, debemos de hacer un ggplot que observaremos a continuación:



Las gráficas nos definen las variables xegvy y x.ege como los clasificadores principales mas importantes. Al compararlo con los métodos de ID3 que analizamos previamente, podemos ver que ID3 prioriza más a x.egvy. Por lo tanto, creo que sería mejor orientarse por el método de Gini de reducción.

Subsecuentemente, vamos a tener que hallar el numero optimo de arboles basado en como evoluciona el ***out of bag (OOB)*** error:

```
oob_mse<-data.frame(oob_mse=data_filtered$err.rate,
trees=seq_along(data_filtered$err.rate))
ggplot(data=oob_mse[1:500,], aes(x=trees[1:500], y=oob_mse.OOB[1:500]))+
  geom_line()+ labs(title = "Evolution of OOB vs number of trees", x="n. trees")+ theme_bw()
```



Vamos a escoger el valor 75, ya que podemos ver que el error OOB no se reduce a partir de ese punto (aunque se observen fluctuaciones insignificativas)

Resultados del modelo de bagging con los 75 árboles:

El parámetro ntree de randomForest tiene como valor default 500, pero es lo que estamos modificando en comparación con nuestro modelo original hallado.

```
> print(bagging_model_75)

Call:
 randomForest(formula = lettr ~ ., data = letterData, mtry = 16,      ntree = 75, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 75
No. of variables tried at each split: 16

        OOB estimate of  error rate: 1.04%
Confusion matrix:
    X   Y   Z class.error
X 779   6   2 0.010165184
Y   9 775   2 0.013994911
Z   3   2 729 0.006811989
```

Predicciones del set de entrenamiento y validación respectivamente (observamos que no hay errores de clasificación) :

```
> confusionMatrix(pr
          predictions
targets   1   2   3
      1 628   0   0
      2   0 626   0
      3   0   0 591
```

```
> table(prediction=pred
          real
prediction   X   Y   Z
         X 159   0   0
         Y   0 160   0
         Z   0   0 143
```

Random Forest

De lo que se encarga Random Forest es de seleccionar una muestra aleatoria de las variables para poder evitar que se cause un gran sesgo para los clasificadores importantes como xegvy.

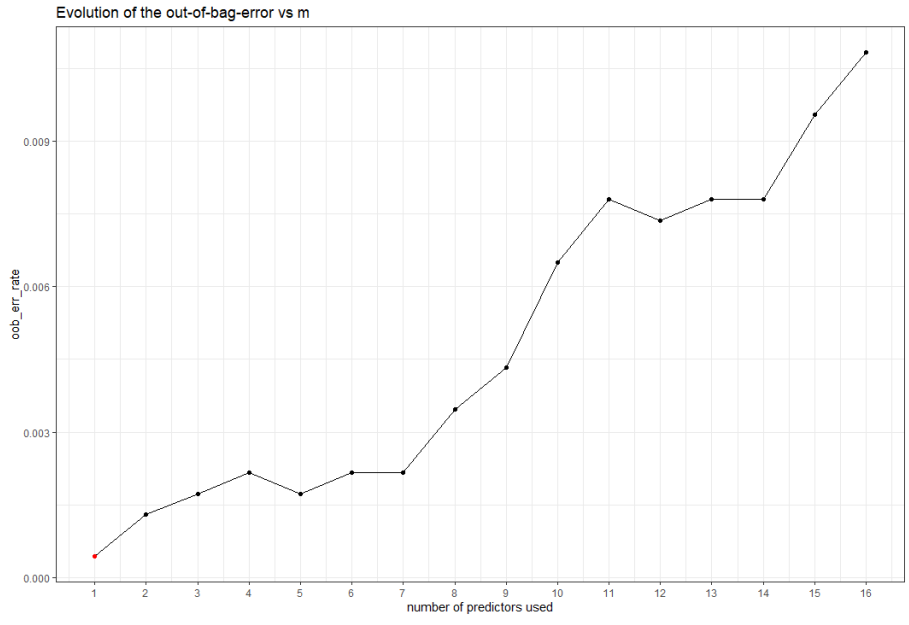
Vamos a seleccionar 4 de los 16 predictores (Ya que la regla del dedo gordo es que debemos seleccionar el numero que es la raíz del de los predictores)

```
bagging_model<- randomForest(formula=lettr  ~ .,
data=letterData_train,  mtry=4)
print(bagging_model)
```

```
> print(bagging_model)

Call:
 randomForest(formula = lettr ~ ., data = letterData_train, mtry = 4)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 0.33%
Confusion matrix:
    X   Y   Z class.error
X 625   3   0 0.004777070
Y   2 624   0 0.003194888
Z   0   1 590 0.001692047
```

Observamos uno de los errores mas pequeños hasta ahora, con un 0.33% de error. Sin embargo, vamos a intentar optimizar las variables para obtener mejores resultados.

Empezaremos dicha optimización hallando el numero optimo de clasificadores a usar con la función Tuning:

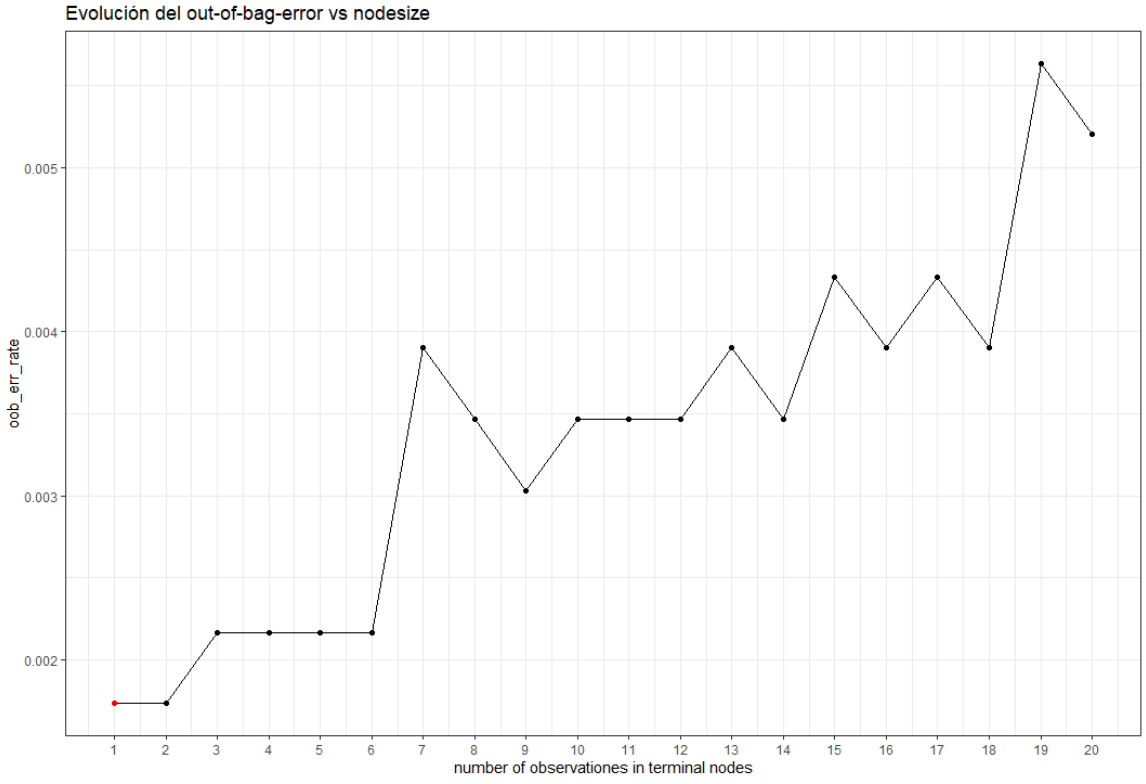


Observamos de la grafica que el numero optimo de clasificadores es 1, lo cual me parece bastante raro. Creo que la causa de esto se puede deber a que adentro de las primero 4 columnas se encuentren las variables con la influencia de clasificación más grande, pero esto quitaría todo el propósito del método de random forest.

```
> print(bagging_model)

Call:
 randomForest(formula = lettr ~ ., data = letterData_train, mtry = 1)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 0.11%
Confusion matrix:
    X   Y   Z class.error
X 627   1   0 0.001592357
Y   1 625   0 0.001597444
Z   0   0 591 0.000000000
```

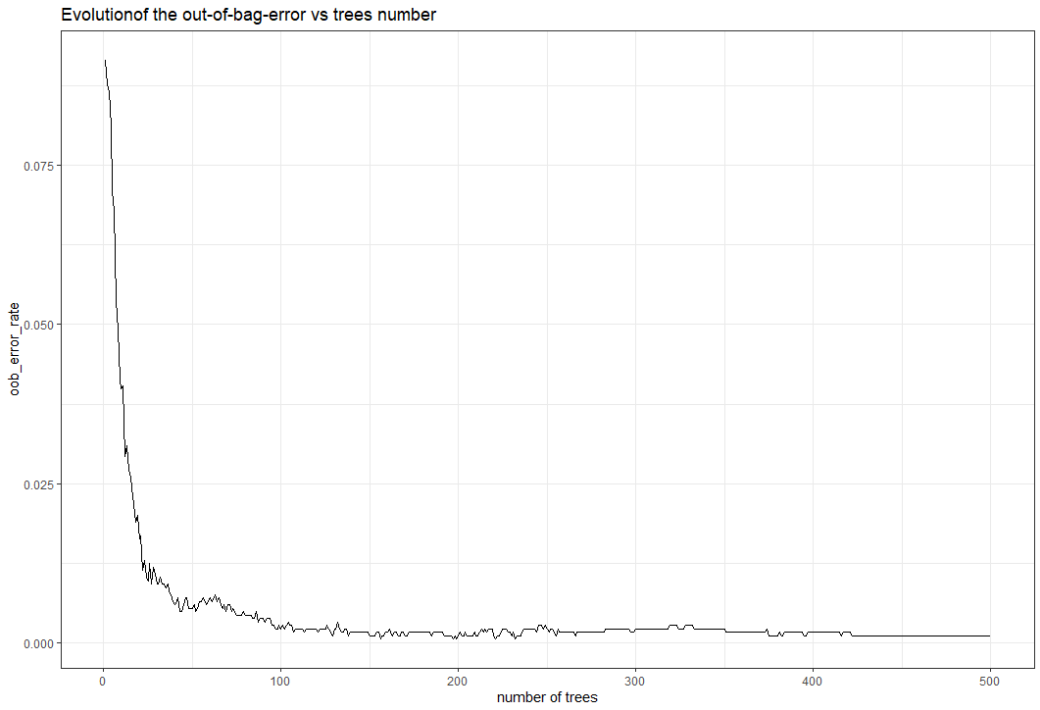
Lo importante de esto al final, es que el total de precisión se nos ha partido entre 3, dejando el margen de error en un 0.11%

Para continuar nuestra optimización, pasamos a calcular el tamaño óptimo de nuestros nodos:



El tamaño con menor tasa de error que nos genera es 1. Por lo tanto, para acabar debemos de decidir el numero de arboles para nuestro random forest.

El cual se observa generando los resultados de comparación entre el error y el numero de arboles que tiene el modelo. Si observamos bien la gráfica, podemos observar que a partir de 425 árboles, ya no hay decrecimiento en el error Out-Of-Bag.

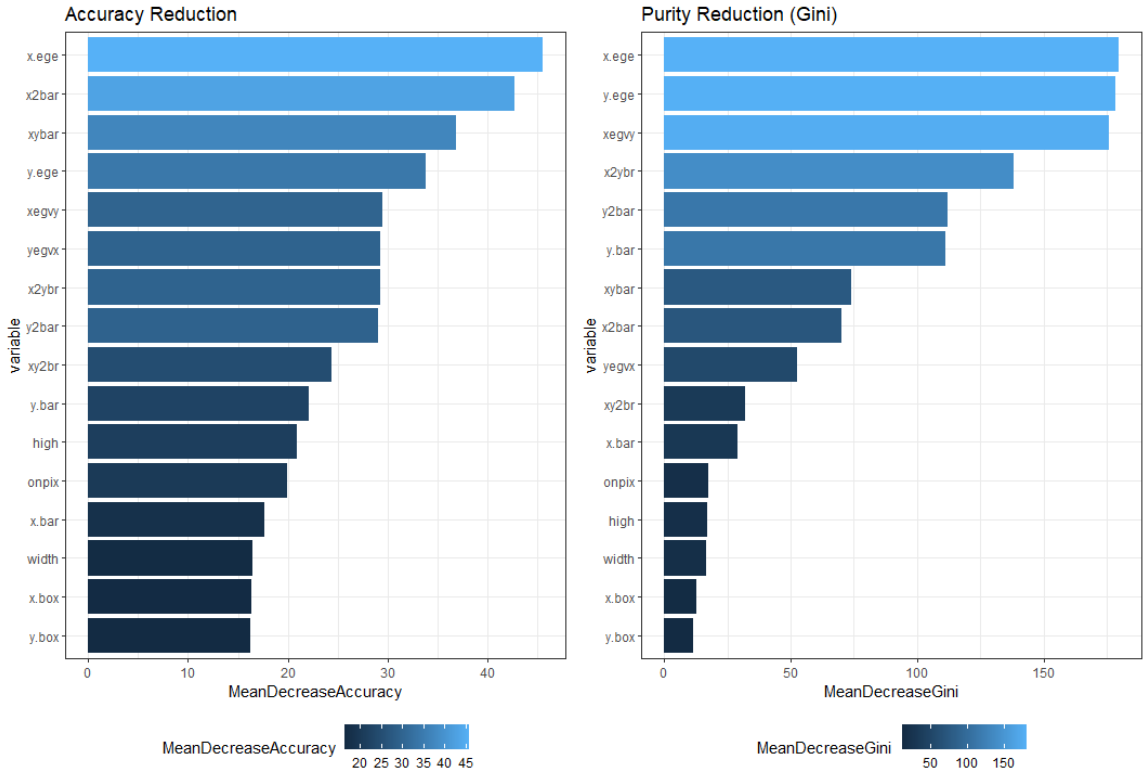


Por lo tanto, ya tenemos todos los valores óptimos que debemos de tener para montar nuestro modelo optimo:

- Numero de clasificadores = 1
- Tamaño del nodo optimo = 1
- Numero de árboles = 425

```
Call:
 randomForest(formula = lettr ~ ., data = letterData_train, mtry = 3,      ntree = 425, nod
esize = 1, importance = TRUE, norm.votes = TRUE)
               Type of random forest: classification
                     Number of trees: 425
No. of variables tried at each split: 3

        OOB estimate of  error rate: 0.16%
Confusion matrix:
    X   Y   Z class.error
X 626   2   0 0.003184713
Y   1 625   0 0.001597444
Z   0   0 591 0.000000000
```

Curiosamente, observamos que tenemos una imprecisión ligeramente mas alta, comparado con cuando hallamos el numero optimo de clasificadores, no se a que se puede deber. A lo mejor el seed (?).



En la grafica de arriba, observamos que la variable más importante es x.ege, la cual es la que mas reduce la exactitud y el coeficiente Gini.

Personalmente, creo que el modelo ha quedado increíblemente bien, y podemos formar las predicciones del training y validación. Esto se debe a que, aunque hayamos creado modelos con una precisión muy alta, siempre se debe comprobar que no haya overfitting y que nuestro modelo pueda generalizar apropiadamente:

```
> caret::confusionMatrix(pred_train, letterData_t
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 628   0   0
         Y   0 626   0
         Z   0   0 591

Overall Statistics

               Accuracy : 1
                 95% CI : (0.998, 1)
    No Information Rate : 0.3404
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 1

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            1.0000   1.0000   1.0000
Specificity            1.0000   1.0000   1.0000
Pos Pred Value         1.0000   1.0000   1.0000
Neg Pred Value         1.0000   1.0000   1.0000
Prevalence             0.3404   0.3393   0.3203
Detection Rate         0.3404   0.3393   0.3203
Detection Prevalence   0.3404   0.3393   0.3203
Balanced Accuracy      1.0000   1.0000   1.0000
```

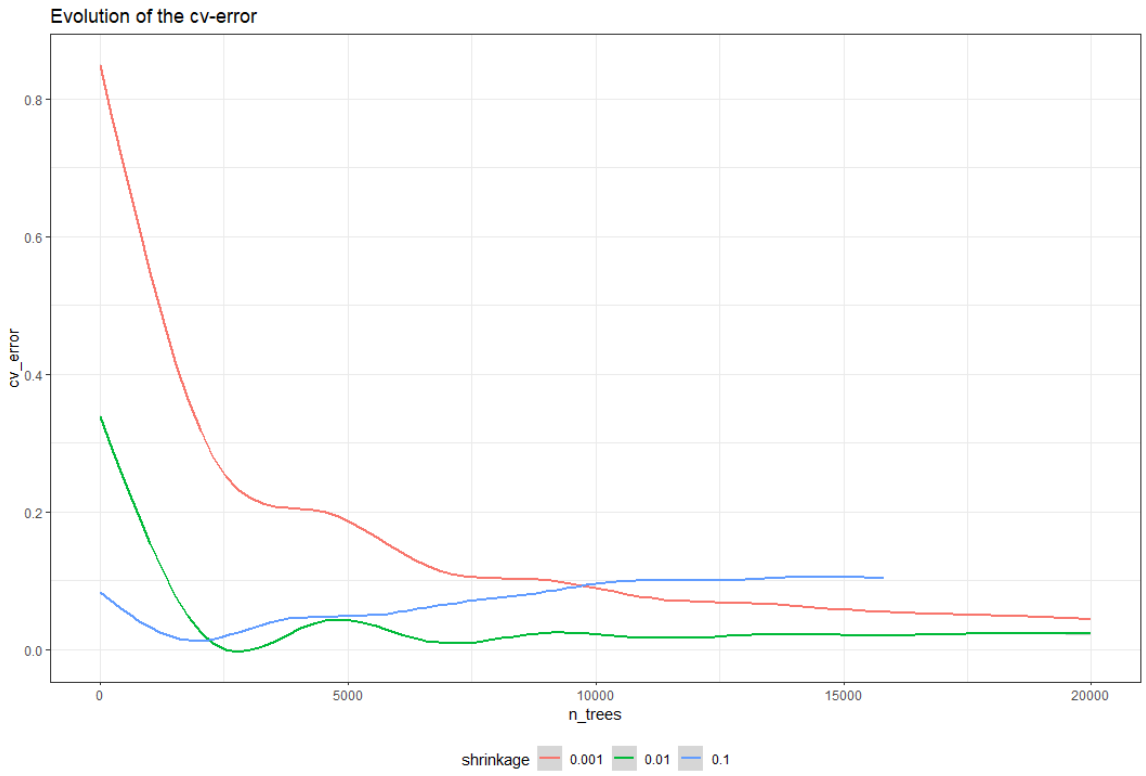
```
> caret::confusionMatrix(predictions, letterData_test$lettr)
Confusion Matrix and Statistics

          Reference
Prediction   X   Y   Z
         X 159   0   0
         Y   0 160   0
         Z   0   0 143

Overall Statistics

               Accuracy : 1
                 95% CI : (0.992, 1)
    No Information Rate : 0.3463
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 1

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: X Class: Y Class: Z
Sensitivity            1.0000   1.0000   1.0000
Specificity            1.0000   1.0000   1.0000
Pos Pred Value         1.0000   1.0000   1.0000
Neg Pred Value         1.0000   1.0000   1.0000
Prevalence             0.3442   0.3463   0.3095
Detection Rate         0.3442   0.3463   0.3095
Detection Prevalence   0.3442   0.3463   0.3095
Balanced Accuracy      1.0000   1.0000   1.0000
```

Observamos ambos modelos que no tienen ningún error, tanto en entrenamiento, como en verificación. Maravilloso

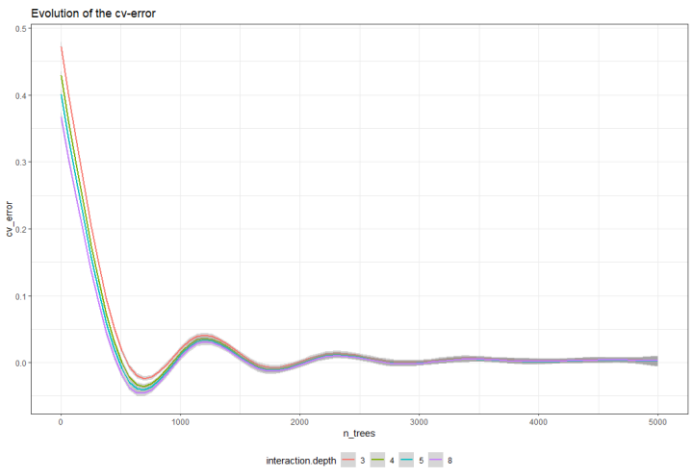
Boosting

Este algoritmo no se basa en las muestras aleatorias, sino que debemos encontrar el shrinkage optimo, observando el numero de arbole que nos de el menor error cv al igual que el Mean Square Error.

Para poder operar con R, debemos importar la librería gbm, la cual se concentra en este tipo de modelos.

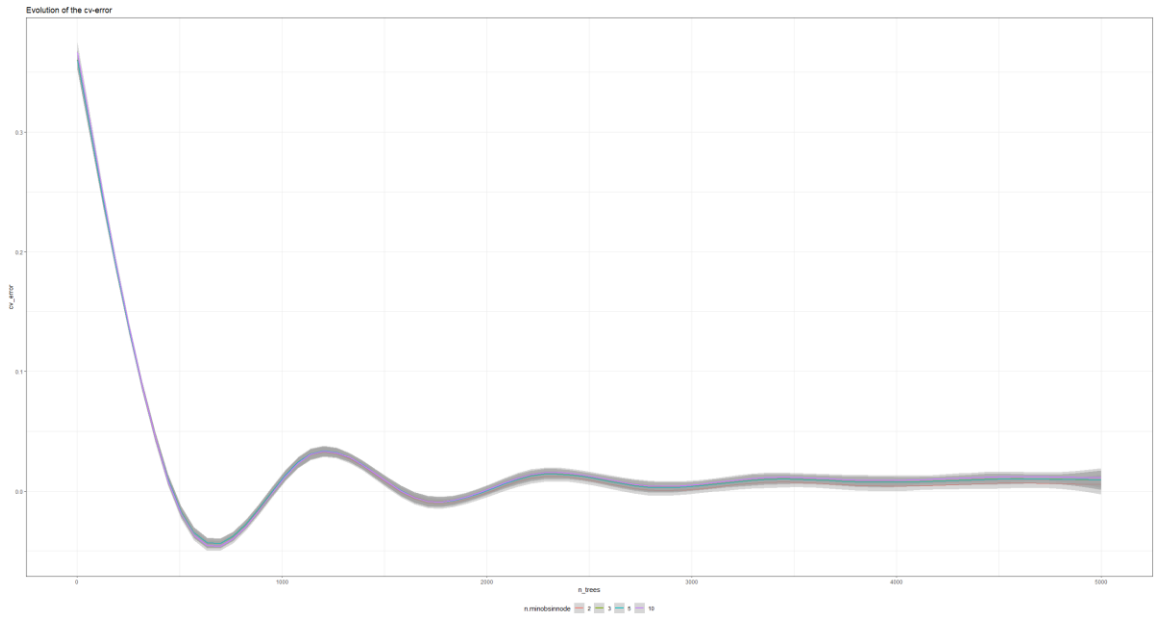


Observamos que el shrinkage óptimo de nuestro modelo mediante cross-validation será 0.01. A continuación debemos de observar la complejidad respectiva, al igual de seleccionar el número de sus observaciones:



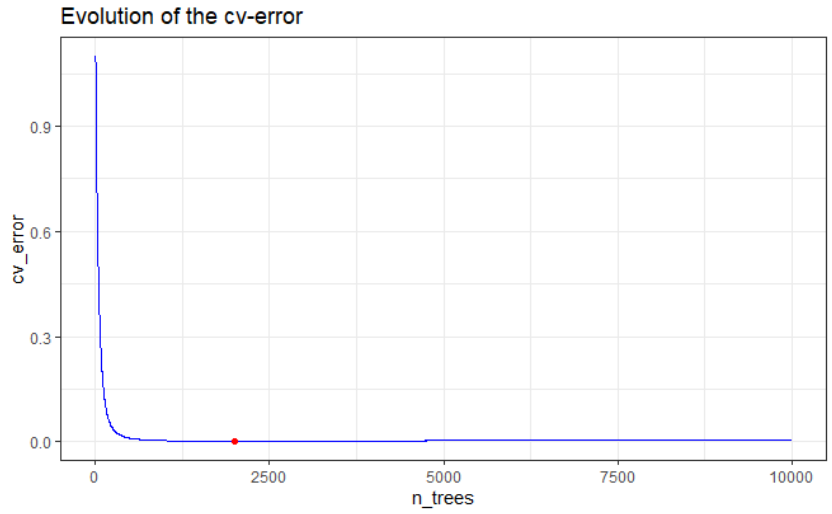
Para la complejidad del sistema hemos decidido por optar por el que tiene menor error (8) aunque todas tengan valores parecidas.

Por último, vamos a observar el número de observaciones optimas:



Una vez más, todos los valores son muy parecidos. Acabamos seleccionando el numero 10 ya que es el que menor error tiene.

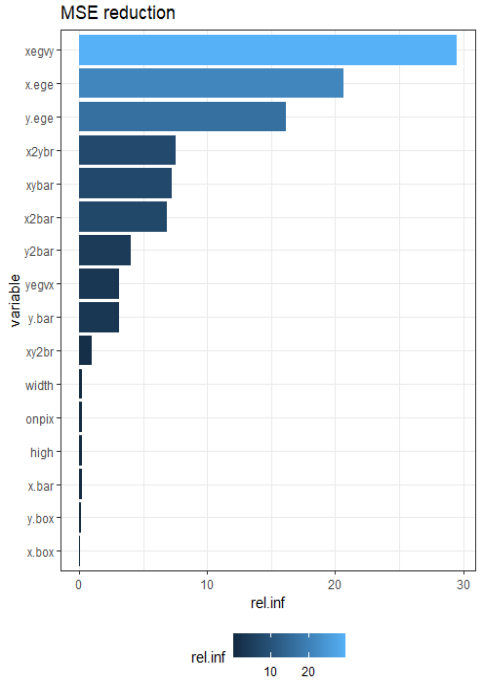
Por último, observamos el número de árboles optimo (Esta propiedad se observa de la misma manera en la cual mencionamos antes con los Random Forests):




```
> error[which.min(error$cv_error),]
      cv_error n_trees
1993 0.00206587    1993
```

Observamos que numero de árboles, del cual nos proporciona el error mas pequeño son 1993 arboles

Graficas de Reducción e Influencia



Tenía planeado sacar la grafica que indica los clasificadores con mayor influencia, pero mi ordenador llevaba mas de 50 minutos haciendo el algoritmo. Como alternativa, pongo el código R que ejecuta dicha tarea:

```
set.seed(123)
tree_boosting <- gbm(lettr ~ ., data
=letterData[train, ],
                     distribution = "multinomial",
                     n.trees = 1000,
                     interaction.depth = 8,
                     shrinkage = 0.01,
                     n.minobsinnode = 10,
                     bag.fraction = 0.5)

summary(tree_boosting)
```

Finalmente, procedemos a hallar la precisión del modelo:

```
> table(prediction=predictions, real= letterData[-train, "lettr"])
          real
prediction   X   Y   Z
         X 162   0   0
         Y   0 145   0
         Z   0   0 155
>
> error_classification <- mean(predictions != letterData[-train, "lettr"])
>
> paste("The classification error is:", 100*error_classification, "%",
+       sum(predictions==letterData[-train, "lettr"]),
+       "correct classified cases from", length(predictions))
[1] "The classification error is: 0 % 462 correct classified cases from 462"
```

Conclusión

Hemos observado todos los diferentes métodos principales de clasificación con Decision Trees y Ensembles con los cuales podemos predecir variables categóricas en nuestro modelo.

Sorprendentemente, hemos obtenido resultados increíbles con un margen super pequeño de error en todos los casos.

Igualmente, hemos de comentar, que los resultados han ido oscilando con cada ejecución que hemos plantado con R. No sé muy bien la causa que puede tener esto. Para intentar solucionarlo procuré imponer el seed value a 123, pero no fue a mucho remedio.

Por último, debo de comentar que el código usado en esta práctica es muy extenso. Si tiene interés por verlo, lo puede obtener en la siguiente página web: