```
library(doParallel)
```

```
## Loading required package: foreach
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
library(DataExplorer)
library(dplyr)
```

```
##
## 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(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

Primero de todo obtenemos los datos necesarios

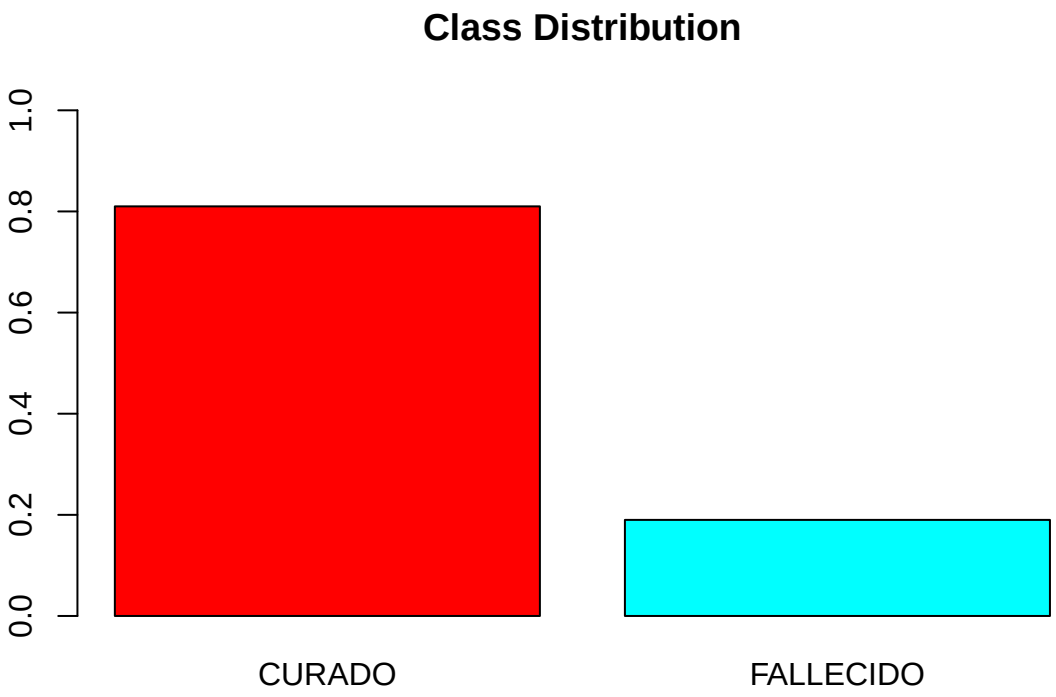
```
datos <- read.csv("../data_tfg.csv", nrows =1800)
str(datos)
```

```
## 'data.frame':    1800 obs. of  20 variables:
##  $ SITUACION: chr  "FALLECIDO" "FALLECIDO" "FALLECIDO" "FALLECIDO" ...
##  $ EDAD     : int  90 95 62 85 72 59 87 92 80 87 ...
##  $ SEXO     : chr  "MUJER" "MUJER" "HOMBRE" "HOMBRE" ...
##  $ NPC      : int  16 7 4 9 12 7 3 8 14 9 ...
##  $ NSIST    : int  10 6 2 6 6 4 3 4 7 5 ...
##  $ DM       : chr  "SI" "SI" "NO" "SI" ...
##  $ IC       : chr  "NO" "NO" "NO" "NO" ...
##  $ EPOC     : chr  "NO" "NO" "NO" "NO" ...
##  $ HTA      : chr  "SI" "NO" "NO" "NO" ...
##  $ DEP      : chr  "SI" "NO" "NO" "SI" ...
##  $ VIH      : chr  "NO" "NO" "NO" "NO" ...
##  $ CI       : chr  "NO" "SI" "NO" "SI" ...
##  $ ACV      : chr  "SI" "SI" "NO" "NO" ...
##  $ IRC      : chr  "NO" "NO" "NO" "NO" ...
##  $ CIR      : chr  "NO" "NO" "NO" "NO" ...
##  $ OST      : chr  "NO" "NO" "NO" "NO" ...
```

```
##  $ ARTROSIS : chr  "SI" "SI" "SI" "NO" ...
##  $ ARTRITIS : chr  "NO" "NO" "NO" "NO" ...
##  $ DEM      : chr  "SI" "NO" "NO" "NO" ...
##  $ DC       : chr  "NO" "NO" "NO" "NO" ...
```

Observamos el desbalanceo en forma de tabla

```
barplot(prop.table(table(datos$SITUACION)),
        col = rainbow(2),
        ylim = c(0, 1.01),
        main = "Class Distribution")
```



Y cuantos hay de cada clase

```
print(paste("CURADO",nrow(datos[datos$SITUACION == 'CURADO',])))
```

```
## [1] "CURADO 1458"
```

```
print(paste("FALLECIDO", nrow(datos[datos$SITUACION == 'FALLECIDO',])))
```

```
## [1] "FALLECIDO 342"
```

Ahora establecemos el preprocesamiento estableciendo a factores los valores necesarios

```
ind.cualit <- c(which(names(datos) == "SITUACION"),which(names(datos)=="SEXO"),which(names(datos)=="DM"

for(i in ind.cualit){

  datos[,i] <- as.factor(datos[, i])

}

str(datos)
```

```
## 'data.frame':    1800 obs. of  20 variables:
##  $ SITUACION: Factor w/ 2 levels "CURADO","FALLECIDO": 2 2 2 2 2 2 2 2 2 2 ...
##  $ EDAD     : int  90 95 62 85 72 59 87 92 80 87 ...
##  $ SEXO     : Factor w/ 2 levels "HOMBRE","MUJER": 2 2 1 1 1 1 2 1 1 2 ...
##  $ NPC      : int  16 7 4 9 12 7 3 8 14 9 ...
##  $ NSIST    : int  10 6 2 6 6 4 3 4 7 5 ...
##  $ DM       : Factor w/ 2 levels "NO","SI": 2 2 1 2 2 2 2 2 2 2 ...
##  $ IC       : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
##  $ EPOC     : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
##  $ HTA      : Factor w/ 2 levels "NO","SI": 2 1 1 1 2 2 1 2 2 1 ...
##  $ DEP      : Factor w/ 2 levels "NO","SI": 2 1 1 2 1 1 1 1 2 1 ...
##  $ VIH      : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
##  $ CI       : Factor w/ 2 levels "NO","SI": 1 2 1 2 2 1 1 1 1 1 ...
##  $ ACV      : Factor w/ 2 levels "NO","SI": 2 2 1 1 2 1 1 1 2 2 ...
##  $ IRC      : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 2 1 1 2 1 ...
##  $ CIR      : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 2 1 ...
##  $ OST      : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
##  $ ARTROSIS : Factor w/ 2 levels "NO","SI": 2 2 2 1 1 1 1 1 1 1 ...
##  $ ARTRITIS : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
##  $ DEM      : Factor w/ 2 levels "NO","SI": 2 1 1 1 2 1 1 2 1 2 ...
##  $ DC       : Factor w/ 2 levels "NO","SI": 1 1 1 1 1 1 1 1 1 1 ...
```

Definimos,las metricas a medir, entre las que destacará el factor kappa que funciona mejor con los datos desbalanceados y será la principal métrica diferencial en el entrenamiento

```
metricas <- function(data, lev = levels(as.factor(data$obs)), model = NULL){
  c(
    ACCURACY = MLmetrics::Accuracy(data[, "pred"], data[, "obs"]),
    SENS = sensitivity(data[, "pred"],data[, "obs"],positive="FALLECIDO",negative="CURADO"),
    SPEC = specificity(data[, "pred"], data[, "obs"],positive="FALLECIDO",negative="CURADO"),
    PPV = posPredValue(data[, "pred"], data[, "obs"],positive="FALLECIDO",negative="CURADO"),
    NPV = negPredValue(data[, "pred"], data[, "obs"],positive="FALLECIDO",negative="CURADO"),
    KAPPA = psych::cohen.kappa(cbind(data[, "obs"],data[, "pred"]))$kappa,
    BAL_ACC = (sensitivity(data[, "pred"],data[, "obs"],positive="FALLECIDO",negative="CURADO") + speci
  )
}
```

Recordemos que para el problema tenemos varios parametros previos:

np -> Numero de elementos de la clase minoritaria en cada particion p -> Numero de particiones realizadas b -> El número máximo de modelos utilizados para entrenar sobre cada particion mt <- Funcion que determina cuando debemos parar de intentar ampliar el modelo actual

Vamos a crear una funcion para cada parametro

Primero de todo creamos la que hace referencia al parametro np, que representa el numero de elementos de la clase minoritaria en cada particion. En nuestro caso simplemente obtenemos un 75% de dichos elementos.

```
calculate_np <- function( train.set,min_str, may_str, OUTPUT,  prob_codo=0.75)
{
  nmin = sum(train.set[[OUTPUT]] == min_str)
  nmay = sum(train.set[[OUTPUT]] == may_str)

  np <- round(nmin*prob_codo)
  return(np)
}
```

Ahora pasamos al numero de particiones que en nuestro caso se consigue con la formula estudiada por el

trabajo original, basada en aplicar la division de logaritmos de valores menores que 1 (ambos negativos) para obtener un ratio entre el alfa establecido y el numero de elementos escogidos por particion.

```
calculate_p <- function( train.set,np, min_str, may_str,  OUTPUT, prob_codo=0.75, alpha_p= .01)
{
  nmin = sum(train.set[[OUTPUT]] == min_str)
  nmay = sum(train.set[[OUTPUT]] == may_str)

  p <- ceiling(log(alpha_p)/(log(1-1/np)*np))
  return(p)
}
```

Posteriormente hacemos un enfoque para el parametro b, que se encarga de establecer cuantos modelos como mucho habrá en cada ensemble de cada particion realizada. Para ello haremos un enfoque similar al del numero de particiones, salvo que usaremos una relacion entre nmin y np, y no solo el numero de elementos minoritarios por particion (p). De esta forma b siempre esta por encima del numero de particiones realizadas.

```
calculate_b <- function( train.set,np, min_str, may_str, OUTPUT, prob_codo=0.75, alpha_b= .01)
{
  nmin = sum(train.set[[OUTPUT]] == min_str)
  nmay = sum(train.set[[OUTPUT]] == may_str)

  b <- ceiling(log(alpha_b)/(log(1-1/nmin)*np))
  return(b)
}
```

Por ultimo tenemos la funcion mt que calcula cuantos intentos como mucho deberian hacerse para ampliar un ensemble. Si fueran mas que esos consideramos que ya no se puede mejorar mas y lo dejariamos como queda. Se observa que siempre se haran menos de b por ensemble, pues b es el limite de modelos por cada ensemble establecido

```
mt <- function(b, n) { ceiling((b-n) / 3) }
```

Finalmente tenemos una funcion auxiliar que devuelve b veces la misma funcion de entrenamiento, pues nuestra funcion admite un vector de modelos a entrenar para cada intento del ensemble.

```
get_function_vector <- function(b,function_training){
  function_vector <- c()
  for(i in 1:b){
    function_vector <- append(function_vector, function_training)
  }


  return(function_vector)
}
```

Creamos ahora un vector de configuraciones dado por varias funciones. La idea base es crear una configuracion para Random forest, regresion logistica, Support Vector Machines y Gradient Boost.

```
configuraciones <- c(

#RFOREST
 function(df.train, metricas) {


    tC <- trainControl(
      summaryFunction = metricas,
      allowParallel = TRUE,
```

```
      classProbs = TRUE
    )


    method <- "ranger"
    metric <- "KAPPA"
    maximize <- T


    #Entrenamos el randomforest
    rf <- train(
      SITUACION ~ .,
      data = df.train,
      method = method,
      metric = metric,
      maximize = maximize,
      num.trees = 200,
      importance = "impurity",
      trControl = tC
    )

    return(rf)
  },
#RLOG
 function(df.train, metricas) {

    tC <- trainControl(
      summaryFunction = metricas,
      allowParallel = TRUE,
      classProbs = TRUE
    )

    method <- "glmnet"
    metric <- "KAPPA"
    maximize <- T

    # Entrenamos la Rlog
    rlog <- train(SITUACION ~ .,
      data = df.train,
      method = "glmnet",
      family = 'binomial',
      metric = "KAPPA",
      maximize = T,
      trControl = tC
    )

    return(rlog)
  },
#SVM
function(df.train, metricas) {

    tC <- trainControl(
      summaryFunction = metricas,
```

```
      allowParallel = TRUE,
      classProbs = TRUE
    )

    method <- "svmLinear"
    metric <- "KAPPA"
    maximize <- T

    # Entrenamos la Rlog
    svm <- train(
      SITUACION ~ .,
      data = df.train,
      method = method,
      metric = metric,
      maximize = maximize,
      trControl = tC
    )

    return(svm)
  },
#GBM
 function(df.train, metricas) {

    tC <- trainControl(
      summaryFunction = metricas,
      allowParallel = TRUE,
      classProbs = TRUE,
      verboseIter = FALSE

    )

    method <- "gbm"
    metric <- "KAPPA"
    maximize <- T

    # Entrenamos el gradient boosting
    gbm <- train(SITUACION ~ .,
                 data = df.train,
                 method = method,
                 metric = metric,
                 maximize = maximize,
                 verbose = FALSE,
                 trControl = tC
    )

    return(gbm)
  })

print(length(configuraciones))
```

```
## [1] 4
```

Ahora que ya tenemos las configuraciones pasamos a obtener una separacion en subconjuntos balanceados. Para ello obtengo con la funcion creada imbalancedFold una serie de pliegues que permitan aplicar esa división

que permita que cada conjunto del k-fold tenga repartidos los subconjuntos desbalanceados.

```
imbalancedFold <- function(data, n_folds, target, minority_class) {
  n_samples <- nrow(data)
  n_majority_total <- n_samples - sum(data[[target]] == minority_class)
  n_minority_total <- sum(data[[target]] == minority_class)

  n_minority_per_fold <- ceiling(n_minority_total / n_folds)
  n_majority_per_fold <- ceiling(n_samples / n_folds - n_minority_per_fold)

  fold_indices <- list()

  for (i in 1:n_folds) {
    #elegimos muestras de la clase mayoritaria
    majority_indices <- which(data[[target]] != minority_class)
    used_majority_indices <- unlist(fold_indices)
    available_majority_indices <- setdiff(majority_indices, used_majority_indices)

    n_available_majority <- length(available_majority_indices)
    n_majority_this_fold <- min(n_majority_per_fold, n_available_majority)
    selected_majority_indices <- sample(available_majority_indices, size = n_majority_this_fold)

    #Cogemos las muestras de la clase minoritaria
    minority_indices <- which(data[[target]] == minority_class)
    used_minority_indices <- setdiff(used_majority_indices, available_majority_indices)
    available_minority_indices <- setdiff(minority_indices, used_minority_indices)
    n_available_minority <- length(available_minority_indices)
    n_minority_this_fold <- min(n_minority_per_fold, n_available_minority)
    selected_minority_indices <- sample(available_minority_indices, size = n_minority_this_fold)

    #Combinamos ambos indices
    selected_indices <- c(selected_majority_indices, selected_minority_indices)

    fold_indices[[i]] <- selected_indices
  }

  return(fold_indices)
}
```

En nuestro caso de prueba haremos k=5

```
folds <-imbalancedFold(datos, 5,"SITUACION", "FALLECIDO")
```

Ahora definimos la funcion que realiza el entrenamiento. Sigue las ideas explicadas en el trabajo. Consiste en realizar p particiones donde aseguremos en cada una una cantidad np de elementos de la clase minoritaria y sobre ellos entrenar multiples modelos.

Para eso tenemos una lista llamada dfs que contiene los indices de cada particion bien balanceada. Sobre cada una de estas particiones realizaremos un entrenamiento iterativo:

- Primero se entrena el modelo sobre la primera funcion del vector de funciones y se almacenan sus resultados de las metricas pasadas
- Despues se intenta ampliar el modelo hasta que la funcion de intentos “mt” lo limite. Para ampliar el modelo se entrena con la siguiente funcion de entrenamiento dada por el parametro de funciones y se compara el valor obtenido con esa funcion nueva y sin ella. Si se mejora se incorpora al ensemble, sino se pasa a un nuevo intento.
- Una vez se ha terminado el numero de intentos para el ensemble actual se considera completado y se

incorpora como modelo final dado para la particion.

Asi seguiremos hasta realizar este proceso en las p particiones y tendremos el modelo como respuesta.

```
train_IPIP <- function( prop.mayoritaria, OUTPUT, min_str, may_str, train.set, test.set,
                        funciones, prediccion, metricas, b, np,  p, mt, seed=42){


  set.seed(seed)
  nmin = sum(train.set[[OUTPUT]] == min_str)
  nmay = sum(train.set[[OUTPUT]] == may_str)



  maySubSize <-  round(np*prop.mayoritaria/(1-prop.mayoritaria))
  minSubSize <-  np

  #Incluye en cada posicion los valores de los elementos de dicha particion, de 1 a p
  dfs <- list()

  minoritario = train.set %>% dplyr::filter(.data[[OUTPUT]] == min_str)
  mayoritario = train.set %>% dplyr::filter(.data[[OUTPUT]] == may_str)

  for(k in 1:p){
    id.minoritaria <- sample(x = 1:nmin, size = minSubSize) #Índices de clase minoritaria para cada sub
    id.mayoritaria <- sample(x= 1:nmay, size = maySubSize) #Índices de la clase mayoritaria para cada s

    dfs[[k]] <- rbind(minoritario[id.minoritaria,],mayoritario[id.mayoritaria,])
  }

  E <- list() # Modelo final (ensemble de ensembles)


  for(k in 1:p){
    Ek <- list() # Ensemble de modelos k-ésimo
    i <- 0 # Contador para el número de intentos de ampliar el ensemble

    # Conjunto de datos balanceado:
    df <- dfs[[k]]
    modelo_i = 1
    while(length(Ek)<=b && i<mt(b,length(Ek))){
      # Seleccionamos muestras para entrenar el modelo
      mayoritaria <- which(df[[OUTPUT]] == may_str)
      minoritaria <- which(df[[OUTPUT]] == min_str)
      ind.train <- c(
        sample(mayoritaria, size = maySubSize, replace = TRUE),
        sample(minoritaria, size = minSubSize, replace = TRUE)
      )

      #Entrenamos con el modelo siguiente en la lista de funciones pasada
      #La idea es establecer un orden de prioridad entre modelos

      modelo <- funciones[[length(Ek)+1]](df[ind.train,], metricas)
      metricas.ensemble <-
        if (length(Ek)==0){
```

```
          u <- -Inf;
          names(u) <- "KAPPA";
          u;
        } else{
          metricas(data.frame(
          obs = test.set[[OUTPUT]],
          pred = as.factor(prediccion(Ek, test.set[colnames(test.set)!=OUTPUT]))
          ))
        }

      Ek[[length(Ek)+1]] <- modelo
      metricas.ensemble.new <- metricas(data.frame(
      obs = test.set[[OUTPUT]],
      pred= as.factor(prediccion(Ek, test.set[colnames(test.set)!=OUTPUT]))
      ))

      #Comprobamos si el nuevo ensemble mejora el modelo. Si no lo hace borramos el elemento incorporad
      if(metricas.ensemble.new["KAPPA"] <= metricas.ensemble["KAPPA"]){
        i <- i+1
        Ek[[length(Ek)]] <- NULL
      } else{
        # En caso de ampliar el ensemble, reseteamos las oportunidades de cara a una nueva ampliación
        i <- 0
      }
    } # Fin del WHILE (hemos terminado de construir el ensemble k-ésimo)

    # Guardamos la información del ensemble k-ésimo
    E[[length(E)+1]] <- Ek

  }
return(E);
}
```

Ahora falta la funcion de prediccion. En nuestro caso consiste en dos funciones.

La primera se aplica sobre el ensemble de modelos de una particion concreta. Sobre cada modelo establecido se realiza la prediccion especifica (metodo predict del mismo). Luego se hace una tabla que asigna a cada ejemplo la proporcion de modelos que han predicho de esa forma. Si han sido mas de q (75% por defecto, es decir, tres cuartos de los modelos al menos predicen que es curado) se establece que se predice como "curado" y si no será fallecido. Finalmente se hace lo mismo pero en vez de usar la funcion predict de cada modelo se pasa a usar la funcion de prediccion dada sobre cada ensemble, se realiza la proporción sobre los conjuntos de ensembles y con si el valor de proporcion de prediccion es >50% se asume como "CURADO" siendo si no "FALLECIDO".

```
prediccion <- function(conj.model, x, q = 0.75){ #q=0.75, pero se deberían probar valores como 0.5, 0.2
  pred <- data.frame(matrix(nrow=nrow(x),ncol=0))
  for(modelo in conj.model) pred <- cbind(pred, predict(modelo,x))
  pred <- apply(pred, 1, function(x) prop.table(table(x))["CURADO"])
  ifelse(is.na(pred) | pred<q, "FALLECIDO", "CURADO")
}

prediccion.final <- function(ensemble, x, q = 0.5){
  # Colocamos en cada fila de un conjunto de datos todas las predicciones para una muestra
  pred <- as.data.frame(lapply(ensemble, function(e) prediccion(e,x)))
  pred <- apply(pred, 1, function(x) prop.table(table(x))["CURADO"])
```

```
  ifelse(is.na(pred) | pred<q, "FALLECIDO", "CURADO")
}
```

Ahora pasamos a realizar el entrenamiento sobre cada configuracion establecida en la lista de configuraciones. Para ello recorremos en cada configuracion el k-fold y damos para cada fold una funcion de vectores asociada al numero b de modelos maximos para un ensemble (de momento todas las funciones son iguales en cada iteracion, pero este approach permite crear vectores de funciones variables).

```
mean_metricas <- list()



for(conf in 1:length(configuraciones)){

print(sprintf("Configuracion %d", conf))

  metricas.final <- list()
  for (i in 1:length(folds)) {

    train.set <- datos[unlist(folds[i]),]
    test.set <- datos[-unlist(folds[i]),]

    np <- calculate_np( train.set, "FALLECIDO", "CURADO", "SITUACION")
    p <- calculate_p( train.set,np, "FALLECIDO", "CURADO", "SITUACION")
    b <- calculate_b( train.set,np,  "FALLECIDO", "CURADO", "SITUACION")
    function_vector<- get_function_vector(b, configuraciones[conf])
    ensemble.fold <- train_IPIP(0.55, "SITUACION", "FALLECIDO", "CURADO", train.set, test.set,
                    function_vector, prediccion,  metricas, b, np, p, mt)

    print(sprintf("Valor de los ensembles en el fold %d:", i))
    print(unlist(lapply(ensemble.fold,length)))

    metricas.final <- append( metricas.final, metricas(data.frame(
      obs = test.set$SITUACION,
      pred= as.factor(prediccion.final(ensemble.fold, test.set[-1]))
    )))


  }

  mean_metricas <- append(mean_metricas, apply(matrix(unlist(metricas.final), ncol= 7, byrow=T), 2, mea

}
```

```
## [1] "Configuracion 1"
## [1] "Valor de los ensembles en el fold 1:"
## [1] 1 1 2 2 1
## [1] "Valor de los ensembles en el fold 2:"
## [1] 2 4 1 1 2
## [1] "Valor de los ensembles en el fold 3:"
## [1] 1 2 1 1 2
## [1] "Valor de los ensembles en el fold 4:"
## [1] 1 1 1 1 1
## [1] "Valor de los ensembles en el fold 5:"
## [1] 1 2 4 1 1
```

```
## [1] "Configuracion 2"
## [1] "Valor de los ensembles en el fold 1:"
## [1] 4 1 2 1 2
## [1] "Valor de los ensembles en el fold 2:"
## [1] 4 1 1 2 2
## [1] "Valor de los ensembles en el fold 3:"
## [1] 2 1 1 1 1
## [1] "Valor de los ensembles en el fold 4:"
## [1] 1 1 4 2 1
## [1] "Valor de los ensembles en el fold 5:"
## [1] 2 2 1 1 1
## [1] "Configuracion 3"
## [1] "Valor de los ensembles en el fold 1:"
## [1] 2 2 1 1 2
## [1] "Valor de los ensembles en el fold 2:"
## [1] 4 1 1 1 1
## [1] "Valor de los ensembles en el fold 3:"
## [1] 1 1 1 1 1
## [1] "Valor de los ensembles en el fold 4:"
## [1] 2 1 4 1 1
## [1] "Valor de los ensembles en el fold 5:"
## [1] 1 1 4 1 1
## [1] "Configuracion 4"
## [1] "Valor de los ensembles en el fold 1:"
## [1] 5 2 2 1 5
## [1] "Valor de los ensembles en el fold 2:"
## [1] 1 1 1 1 1
## [1] "Valor de los ensembles en el fold 3:"
## [1] 1 1 1 1 2
## [1] "Valor de los ensembles en el fold 4:"
## [1] 1 1 2 2 2
## [1] "Valor de los ensembles en el fold 5:"
## [1] 1 1 1 1 1
```

Finalmente se muestra el resultado de las diferentes estadisticas para cada configuracion dada.

```
matrix_mean <- matrix(mean_metricas, nrow = 4, ncol = 7, byrow = TRUE)

col_names <- c("ACCURACY", "SENS", "SPEC", "PPV", "NPV", "KAPPA", "BAL_ACC")
row_names <- c("RANGER", "RLOG", "SVM", "GBM")

colnames(matrix_mean) <- col_names
rownames(matrix_mean) <- row_names

matrix_mean
```

```
##        ACCURACY  SENS      SPEC      PPV       NPV       KAPPA     BAL_ACC
## RANGER 0.8893506 0.920266  0.8820908 0.6479039 0.9794152 0.6907528 0.9011784
## RLOG   0.8925476 0.9166348 0.8868895 0.6556424 0.9784649 0.6972194 0.9017621
## SVM    0.8975384 0.9107979 0.8944302 0.6704659 0.9772458 0.7077459 0.902614
## GBM    0.8986556 0.9107899 0.8958012 0.6724236 0.9772357 0.7101008 0.9032956
```