uc3m | Universidad Carlos III de Madrid

OPENCOURSEWARE

APRENDIZAJE AUTOMÁTICO PARA EL ANÁLISIS DE DATOS

GRADO EN ESTADÍSTICA Y EMPRESA

Ricardo Aler

Tutorial

Ricardo Aler

1 Agosto 2019

Tutorial sobre datos desbalanceados en mlR

Introducción

Vamos a usar el conjunto de datos *ubIonosphere*, que tiene un ligero desbalanceo (proporción de positivos frente a negativos = 0.56). Podemos ver que dado que la clase mayoritaria es un 64% de los datos, un clasificador trivial que predijera siempre "0" tendría un 64% de aciertos.

```
data(ubIonosphere, package = "unbalanced")

# Comprobamos que la clase es un factor. Lo es, pero si no, habría que convertirla a factor.
class(ubIonosphere$Class)
```

```
## [1] "factor"
```

```
(conteo <- table(ubIonosphere$Class))
```

```
##
##   0   1
## 225 126
```

```
conteo/sum(conteo)
```

```
##
##         0         1
## 0.6410256 0.3589744
```

```
#GRADO DE DESBALANCEO
conteo[2]/conteo[1]
```

```
##    1
## 0.56
```

El problema que tienen los datos desbalanceados es que las clases minoritarias se clasifican mal. Veamos si eso es así con k-vecinos. Tenemos que definir una tarea de clasificación. Es importante que digamos cual de las dos es la clase positiva (la minoritaria), que en este caso es la "1". También, cuando definamos el método para evaluar el modelo (con "Holdout" en este caso), tenemos que especificar que las particiones tienen que estar estratificadas. Por último, entrenamos / evaluamos el modelo con *resample*. Vamos a computar las medidas de error *tpr* y *tnr* (true positive rate y true negative rate), que son la tasa de aciertos de la clase positiva y negativa, respectivamente. También vamos a computar la tasa de aciertos (*acc*) y el "balanced accuracy", que es la media de *tpr* y *tnr*.

```
library(mlr)
```

```
## Warning: package 'mlr' was built under R version 3.5.3
```

```
## Loading required package: ParamHelpers
```

```
## Warning: package 'ParamHelpers' was built under R version 3.5.3
```

```r
# Primero definimos la tarea. Nótese que definimos cual es la clase positiva
task_io <- makeClassifTask(data=ubIonosphere, target="Class", positive = "1")
learner_knn <- makeLearner("classif.kknn")
```

```
## Loading required package: kknn
```

```r
# Al definir el método de evaluación, tenemos que especificar que las particiones sean estratificadas
metodo_evaluacion <- makeResampleDesc("Holdout", stratify=TRUE)
set.seed(0)
particion_datos <- makeResampleInstance(metodo_evaluacion, task_io)

set.seed(0)
errores_kknn <- resample(learner_kknn, task_io, particion_datos, measures = list(acc, bac, tpr, tnr))
```

```
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## [Resample] iter 1:    0.8290598 0.7776190 0.5952381 0.9600000
##
## Aggregated Result: acc.test.mean=0.8290598,bac.test.mean=0.7776190,tpr.test.mean=0.5952381,tnr.test.m
##
```

Vemos que la clase negativa (mayoritaria) se aprende mucho mejor que la positiva: tnr = 0.96 y tpr = 0.5952381. Aún así, dado que *acc* (83%) supera el 65% de aciertos, el clasificador construido con *knn* es mejor que un clasificador trivial. Vemos también que el error *bac* representa mejor que *acc* la precisión del clasificador, sobre todo con respecto a la clase positiva (minoritaria).

```r
errores_kknn$aggr
```

```
## acc.test.mean bac.test.mean tpr.test.mean tnr.test.mean
##     0.8290598     0.7776190     0.5952381     0.9600000
```

La matriz de confusión muestra similares resultados

```r
calculateConfusionMatrix(errores_kknn$pred, relative = TRUE)
```

```
## Relative confusion matrix (normalized by row/column):
##         predicted
## true     0         1         -err.-
##   0      0.96/0.81 0.04/0.11 0.04
##   1      0.40/0.19 0.60/0.89 0.40
##   -err.-      0.19      0.11 0.17
##
##
## Absolute confusion matrix:
##         predicted
## true      0  1 -err.-
##   0      72  3      3
##   1      17 25     17
##   -err.- 17  3     20
```

Mostrando curvas ROC

Vamos a hacer una curva ROC para *kknn*. Para ello es necesario hacer predicciones probabilísticas. Aquí las podemos ver:

```
learner_kknn_prob <- makeLearner("classif.kknn", predict.type = "prob")
model_kknn_prob <- train(learner_kknn_prob, task_io, subset = particion_datos$train.inds[[1]])
predicciones <- predict(model_kknn_prob,  task_io, subset = particion_datos$test.inds[[1]])
head(predicciones$data)
```

```
##      id truth    prob.0    prob.1 response
## 321 321     0 1.0000000 0.0000000        0
## 264 264     0 1.0000000 0.0000000        0
## 144 144     0 0.1029562 0.8970438        1
## 120 120     0 1.0000000 0.0000000        0
## 138 138     0 1.0000000 0.0000000        0
## 333 333     0 1.0000000 0.0000000        0
```

Pero en este momento, no nos interesan las predicciones, vamos a calcular los errores directamente con *resample*. Ahora que hacemos predicciones probabilísticas, podemos aprovechar para calcular también el área bajo la curva (auc), como se puede ver abajo.

```
set.seed(0)
errores_kknn <- resample(learner_kknn_prob, task_io, particion_datos, measures = list(acc, bac, tpr, tn
```

```
## Resampling: holdout
```

```
## Measures:             acc       bac       tpr       tnr       auc
```

```
## [Resample] iter 1:    0.8290598 0.7776190 0.5952381 0.9600000 0.9079365
```

```
##
```

```
## Aggregated Result: acc.test.mean=0.8290598,bac.test.mean=0.7776190,tpr.test.mean=0.5952381,tnr.test.m
```

```
##
```

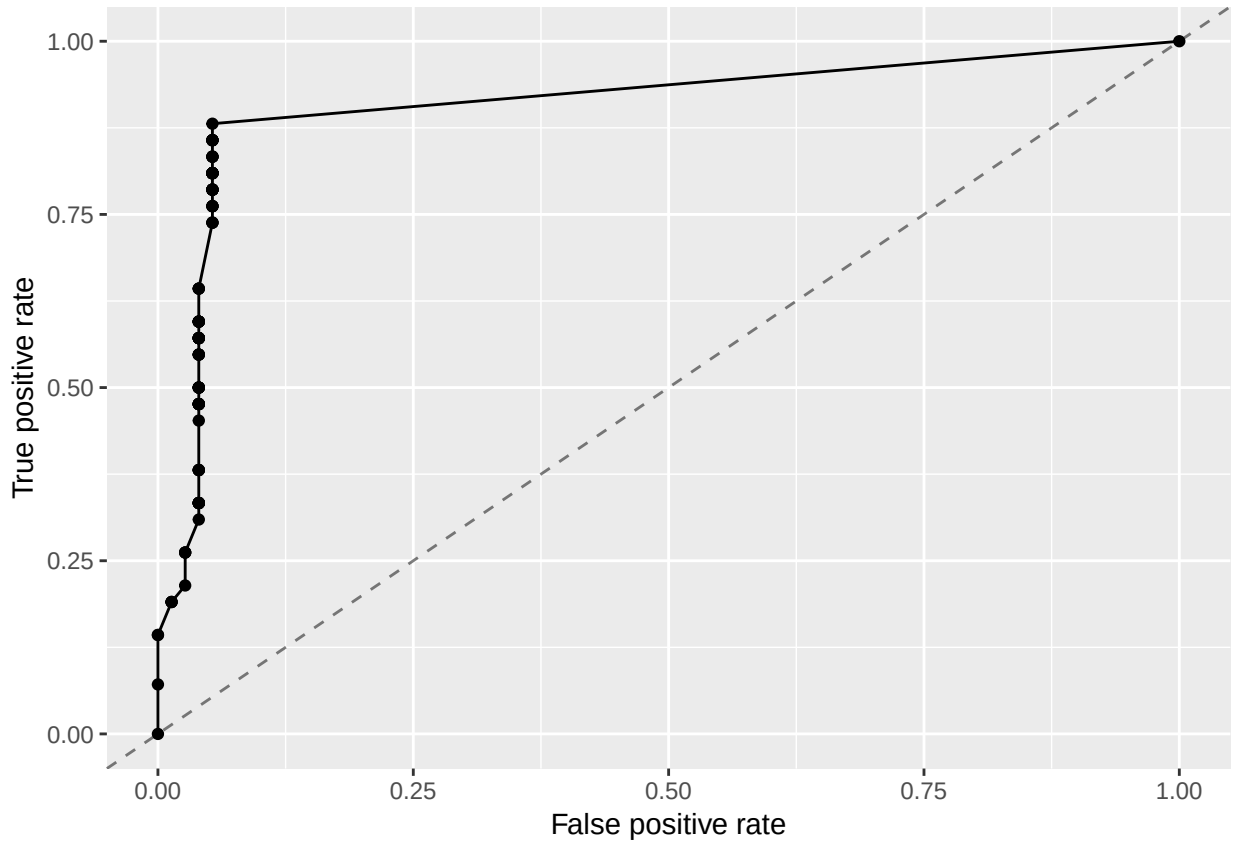
Para generar la curva ROC, primero tenemos que generar cierta información con la función *generateThreshVsPerfData*, a partir de los errores computados anteriormente. Necesitaremos *fpr* y *tpr* para construir la curva ROC, pero podemos generar información también para otras medidas de error, como *tnr* y *bac*. Vemos que la información generada no es mas que un data.frame que contiene una línea para cada valor del threshold.

```
df <- generateThreshVsPerfData(errores_kknn, measures = list(fpr, tpr, tnr, bac))
head(df$data)
```

```
##          fpr       tpr       tnr       bac  threshold
## 1 1.00000000 1.0000000 0.0000000 0.5000000 0.00000000
## 2 0.05333333 0.8809524 0.9466667 0.9138095 0.01010101
## 3 0.05333333 0.8571429 0.9466667 0.9019048 0.02020202
## 4 0.05333333 0.8571429 0.9466667 0.9019048 0.03030303
## 5 0.05333333 0.8571429 0.9466667 0.9019048 0.04040404
## 6 0.05333333 0.8571429 0.9466667 0.9019048 0.05050505
```

Ahora imprimimos la curva ROC. Normalmente, imprime sólo la línea, pero si queremos que dibuje también los puntos, usamos *geom_point()*. mlR genera los plots mediante *ggplot2*.

```
library(ggplot2)
plotROCCurves(df, measures=list(fpr, tpr)) + geom_point()
```



Se puede usar *plotly* para hacer plots ligeramente interactivos. Podemos ver que podríamos mejorar bastante el *tpr*, sin empeorar demasiado el *tnr*. Originalmente obteníamos un *tpr* de 0.59, con un *tnr* de 0.96 (o lo que es lo mismo, un fpr = 1-tnr = 0.04). Vemos que cambiando el threshold, podríamos subir el *tpr* a 0.88 y el *tnr=1-fpr* sólo bajaría a 1-0.05 = 0.95.

```
library(ggplot2)
library(plotly)
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

```
gg <- plotROCCurves(df, measures=list(fpr, tpr)) + geom_point()

ggplotly(gg)
```

Ajustando el threshold a mano

Desgraciadamente, en la curva ROC, no se puede ver el threshold que tendríamos que utilizar para obtener dicho resultado. Vamos a generar otra curva, en cuyo eje x viene el threshold, y en el eje y vienen los distintos errores. Haremos el plot con *ggplot2*. Para ello, primero tenemos que poner los datos en modo *largo* (long)

```
# Primero p
df_long <- tidyr::gather(df$data, variable, value, -threshold)
head(df_long)
```

```
##    threshold variable      value
## 1 0.00000000      fpr 1.00000000
## 2 0.01010101      fpr 0.05333333
## 3 0.02020202      fpr 0.05333333
## 4 0.03030303      fpr 0.05333333
## 5 0.04040404      fpr 0.05333333
## 6 0.05050505      fpr 0.05333333
```

Ahora hacemos el plot. Vamos a hacerlo interactivo. Podemos ver que para un threshold de 0.01 se alcanza un buen equilibrio entre *tpr* y *tnr*. Podemos usar para medirlo la media de los dos *bac*.

```
library(ggplot2)
df_long_tpr_tnr_bac <- df_long[df_long$variable %in% c("tpr", "tnr", "bac"), ]
gg <- ggplot(df_long_tpr_tnr_bac, aes(x=threshold, y=value, col=variable )) + geom_point() + geom_line()
ggplotly(gg)
```

Podemos calcular directamente el threshold, como aquel lugar donde se alcanza el máximo *bac*, así:

```
th <- df$data[which.max(df$data$bac), "threshold"]
th
```

```
## [1] 0.01010101
```

Vamos a fijar dicho threshold con *setThreshold* al hacer las predicciones.

```
preds_kknn_th <- setThreshold(errores_kknn$pred, th)
performance(preds_kknn_th, measures=list(tpr, tnr, bac))
```

```
##       tpr       tnr       bac
## 0.8809524 0.9466667 0.9138095
```

Ajustando el threshold automáticamente

También podemos ajustarlo automáticamente con *tuneThreshold* Hay que fijarse que estamos encontrando el threshold que optimiza *bac*

```
# El threshold óptimo es
(th <- tuneThreshold(errores_kknn$pred, measure=list(bac), nsub=100))
```

```
## $th
## [1] 0.009918694
##
## $perf
##       bac
## 0.9138095
```

```
preds_kknn_th <- setThreshold(errores_kknn$pred, th$th)
errores_kknn_tuned <- performance(preds_kknn_th, measures=list(tpr, tnr, bac))
```

Vemos que obtenemos el mismo *bac* que cuando ajustamos el threshold a mano.

```
errores_kknn_tuned
```

```
##       tpr       tnr       bac
## 0.8809524 0.9466667 0.9138095
```

Ajustando el threshold como cualquier otro hiper-parámetro

En los ajustes de threshold que hemos hecho anteriormente había un aspecto que no era del todo correcto. Tanto las curvas ROC que visualizamos, como el ajuste visual o automático del threshold, lo hicimos con las predicciones en la partición de test, lo cual no es correcto, dado que estamos usando el conjunto de test para ajustar un parámetro del modelo (en este caso, el threshold). Viene bien para visualizar como varían los distintos errores (tpr, tnr, bac, etc.) a medida que cambia el threshold. Pero el ajuste del threshold habría que hacerlo como si fuera cualquier otro hiper-parámetro (la manera más sencilla es con un conjunto de validación, distinto al de test). Eso es lo que vamos a hacer a continuación. En mlR, esto se puede hacer usando el ajuste de hiper-parámetros que ya conocemos, excepto que en *makeTuneControlGrid*, tenemos que aÃ±adir los argumentos *tune.threshold* y *tune.threshold.args*, como vemos en el siguiente código. NÃ¶tese que estamos pidiendo que ajuste el threshold optimizando *bac* (podríamos optimizar cualquier otra medida).

Si quisieramos, podríamos ajustar a la vez el hiper-parámetro k, pero en este caso, sólo ajustaremos el threshold. Esa es al razón por la que hemos fijado la lista de valores de k a un único valor, el 7 (pero podríamos poner una lista de valores, por ejemplo, de 1 a 10).

```
getParamSet(learner_kknn_prob)
```

```
##              Type len     Def                                    Constr Req
## k         integer   -       7                                  1 to Inf   -
## distance  numeric   -       2                                  0 to Inf   -
## kernel   discrete   - optimal rectangular,triangular,epanechnikov,b...   -
## scale     logical   -    TRUE                                          -   -
##          Tunable Trafo
## k           TRUE     -
## distance    TRUE     -
## kernel      TRUE     -
## scale       TRUE     -
```

```
ps = makeParamSet(
  makeDiscreteParam("k", values = 7)
)

control_grid <- makeTuneControlGrid(tune.threshold = TRUE, tune.threshold.args = list(measure=list(bac)
evaluacion_grid <- makeResampleDesc("Holdout", stratify = TRUE)

learner_ajuste_kknn <- makeTuneWrapper(learner_kknn_prob, resampling = evaluacion_grid, par.set = ps, c

set.seed(0)
errores_ajuste_kknn <- resample(learner_ajuste_kknn, task_io, particion_datos, measures = list(acc, bac
```

```
## Resampling: holdout
```

```
## Measures:             acc       bac       tpr       tnr
```

```
## [Tune] Started tuning learner classif.kknn for parameter set:
```

```
##       Type len Def Constr Req Tunable Trafo
## k discrete   -   -       7   -    TRUE     -
```

```
## With control class: TuneControlGrid
## Imputation value: -0
## [Tune-x] 1: k=7
## [Tune-y] 1: bac.test.mean=0.9107143; time: 0.0 min
## [Tune] Result: k=7 : bac.test.mean=0.9107143
## [Resample] iter 1:    0.8974359 0.8780952 0.8095238 0.9466667
##
## Aggregated Result: acc.test.mean=0.8974359,bac.test.mean=0.8780952,tpr.test.mean=0.8095238,tnr.test.m
##
```

Vemos que el threshold es ligeramente distinto al de antes, y también los errores. Esta diferencia es debida a que antes optimizabamos el threshold directamente con el conjunto de test (que ya sabemos que no es correcto), y ahora lo estamos optimizando con un conjunto de validación. Una vez optimizado el threshold, evaluamos el modelo con este threshold sobre el conjunto de test.

```
errores_ajuste_kknn$models[[1]]$learner.model$opt.result$threshold
```

```
## threshold
## 0.1345801
```

```
errores_ajuste_kknn
```

```
## Resample Result
## Task: ubIonosphere
## Learner: classif.kknn.tuned
## Aggr perf: acc.test.mean=0.8974359,bac.test.mean=0.8780952,tpr.test.mean=0.8095238,tnr.test.mean=0.94
## Runtime: 0.223833
```

Podemos aprovechar para ajustar también otros hiper-parámetros (como k), además del threshold:

```
ps = makeParamSet(
  makeDiscreteParam("k", values = 2:10)
)

control_grid <- makeTuneControlGrid(tune.threshold = TRUE, tune.threshold.args = list(measure=list(bac))
evaluacion_grid <- makeResampleDesc("Holdout", stratify = TRUE)

learner_ajuste_kknn <- makeTuneWrapper(learner_kknn_prob, resampling = evaluacion_grid, par.set = ps, co

set.seed(0)
errores_ajuste_kknn <- resample(learner_ajuste_kknn, task_io, particion_datos, measures = list(acc, bac
```

```
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## [Tune] Started tuning learner classif.kknn for parameter set:
##       Type len Def             Constr Req Tunable Trafo
## k discrete   -   - 2,3,4,5,6,7,8,9,10   -    TRUE     -
## With control class: TuneControlGrid
## Imputation value: -0
## [Tune-x] 1: k=2
```

```
## [Tune-y] 1: bac.test.mean=0.8750000; time: 0.0 min
## [Tune-x] 2: k=3
## [Tune-y] 2: bac.test.mean=0.8928571; time: 0.0 min
## [Tune-x] 3: k=4
## [Tune-y] 3: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 4: k=5
## [Tune-y] 4: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 5: k=6
## [Tune-y] 5: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 6: k=7
## [Tune-y] 6: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 7: k=8
## [Tune-y] 7: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 8: k=9
## [Tune-y] 8: bac.test.mean=0.9107143; time: 0.0 min
## [Tune-x] 9: k=10
## [Tune-y] 9: bac.test.mean=0.9107143; time: 0.0 min
## [Tune] Result: k=10 : bac.test.mean=0.9107143
## [Resample] iter 1:    0.9059829 0.8900000 0.8333333 0.9466667
##
## Aggregated Result: acc.test.mean=0.9059829,bac.test.mean=0.8900000,tpr.test.mean=0.8333333,tnr.test.
##
```

```
errores_ajuste_kknn$models[[1]]$learner.model$opt.result$threshold
```

```
## threshold
## 0.1459651
```

```
errores_ajuste_kknn$models[[1]]$learner.model$opt.result$x
```

```
## $k
## [1] 10
```

```
errores_ajuste_kknn
```

```
## Resample Result
## Task: ubIonosphere
## Learner: classif.kknn.tuned
## Aggr perf: acc.test.mean=0.9059829,bac.test.mean=0.8900000,tpr.test.mean=0.8333333,tnr.test.mean=0.9
## Runtime: 1.83565
```

Usando SMOTE para abordar problemas de muestras desbalanceadas

En primer lugar, vamos a rebalancear la muestra "a mano", haciendo que las dos clases tengan el mismo número de instancias. Si la clase mayoritaria tuviera el doble de instancias que la minoritaria, habría que

duplicar el número de instancias positivas (o sea, un *rate* de 2). En este caso, usaremos un *rate* de instancias mayoritaria / instancias minoritaria. Para usar SMOTE, montaremos un método híbrido, cuya primera fase hace el rebalanceo con SMOTE, y cuya segunda fase aplica *knn* sobre la muestra desbalanceada.

```
tabla <- table(getTaskTargets(task_io))
tabla
```

```
##
##   0   1
## 225 126
```

```
rate <- tabla["0"]/tabla["1"]
rate
```

```
##        0
## 1.785714
```

```
#task_m_smote <- smote(task_m, rate = rate)
#table(getTaskTargets(task_m_smote))

learner_smote_kknn = makeSMOTEWrapper(learner_kknn_prob, sw.rate = rate)

set.seed(0)
errores_smote_knn <- resample(learner_smote_kknn, task_io, particion_datos, measures = list(acc, bac, tp
```

```
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## [Resample] iter 1:    0.8888889 0.8609524 0.7619048 0.9600000
##
## Aggregated Result: acc.test.mean=0.8888889,bac.test.mean=0.8609524,tpr.test.mean=0.7619048,tnr.test.m
##
```

Vemos que con este *rate*, conseguimos un error parecido al que obtuvimos antes ajustando el threshold.

```
errores_smote_knn$aggr
```

```
## acc.test.mean bac.test.mean tpr.test.mean tnr.test.mean
##     0.8888889     0.8609524     0.7619048     0.9600000
```

En lugar de poner el *rate* a mano, podemos ajustarlo automáticamente, dado que es un hiper-parámetro del método híbrido *learner_smote_kknn*. Usaremos una validación cruzada de 3 folds para ajustar *rate*.

```
ps = makeParamSet(
  makeDiscreteParam("sw.rate", values = seq(1, 10, 1))
)

evaluacion_grid <- makeResampleDesc("CV", iters=3, stratify = TRUE)
control_grid <- makeTuneControlGrid()

learner_smote_kknn = makeSMOTEWrapper(learner_kknn_prob)

learner_ajuste_smote_kknn <- makeTuneWrapper(learner_smote_kknn, resampling = evaluacion_grid, par.set =
                                             measures = list(bac, tpr, acc, tnr), show.info = FALSE)

set.seed(0)
errores_ajuste_smote_kknn <- resample(learner_ajuste_smote_kknn, task_io, particion_datos, measures = li
```

```
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## [Resample] iter 1:    0.8974359 0.8833333 0.8333333 0.9333333
##
## Aggregated Result: acc.test.mean=0.8974359,bac.test.mean=0.8833333,tpr.test.mean=0.8333333,tnr.test.m
##
```

Vemos que conseguimos un *bac* superior al que hemos ido obteniendo hasta el momento. Vemos que para obtener el mejor resultado, fue necesario utilizar una replicación cuadruple de la clase minoritaria.

```
errores_ajuste_smote_kknn
```

```
## Resample Result
## Task: ubIonosphere
## Learner: classif.kknn.smoted.tuned
## Aggr perf: acc.test.mean=0.8974359,bac.test.mean=0.8833333,tpr.test.mean=0.8333333,tnr.test.mean=0.93
## Runtime: 0.734991
```

```
errores_ajuste_smote_kknn$models[[1]]$learner.model$opt.result
```

```
## Tune result:
## Op. pars: sw.rate=8
## bac.test.mean=0.8947619,tpr.test.mean=0.8095238,acc.test.mean=0.9188034,tnr.test.mean=0.9800000
```

Puede ser interesante ver como cambia el error al cambiar el *rate*. Vemos que multiplicar por 8 las instancias de la clase minoritaria tiene un efecto beneficioso.

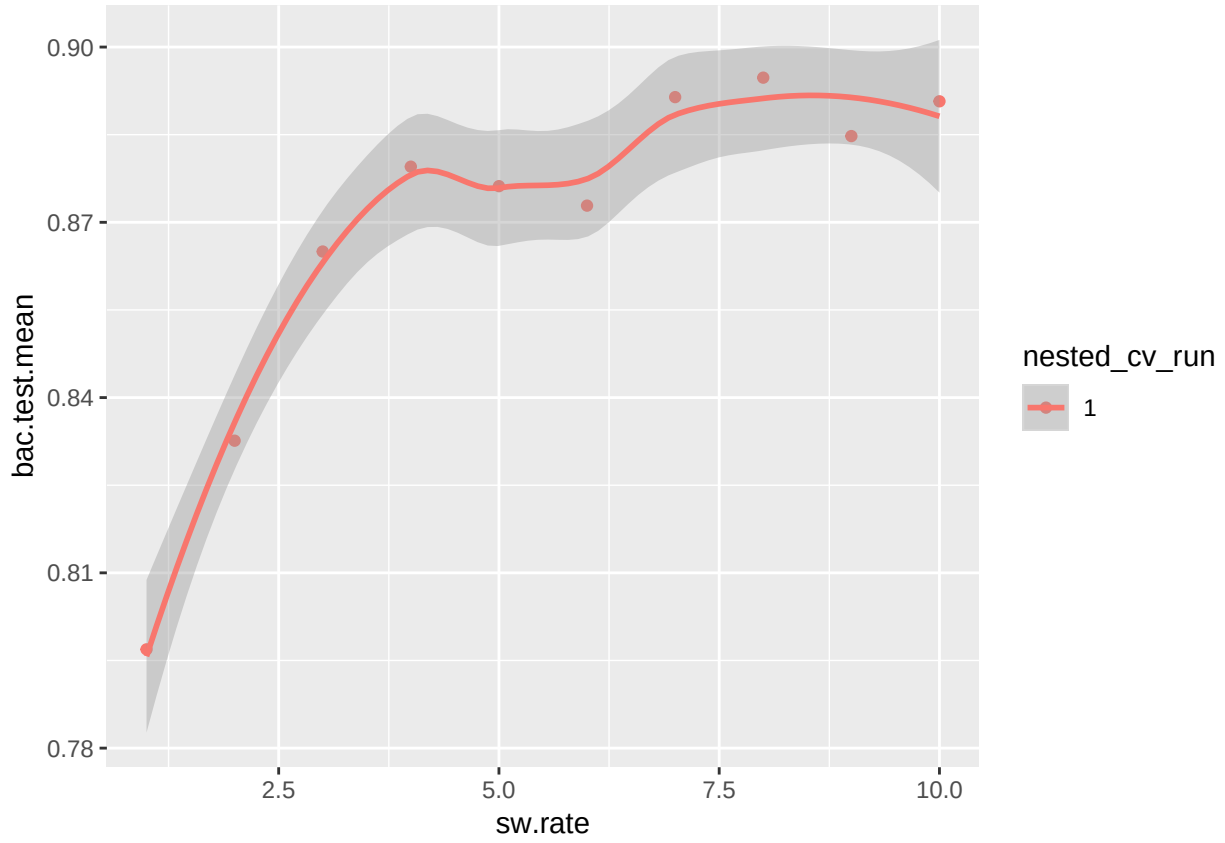
```
efectos <- generateHyperParsEffectData(errores_ajuste_smote_kknn)
head(efectos)
```

```
## $data
##    sw.rate bac.test.mean tpr.test.mean acc.test.mean tnr.test.mean
## 1        1     0.7969048     0.6071429     0.8504274     0.9866667
## 2        2     0.8326190     0.6785714     0.8760684     0.9866667
## 3        3     0.8650000     0.7500000     0.8974359     0.9800000
## 4        4     0.8795238     0.7857143     0.9059829     0.9733333
## 5        5     0.8761905     0.7857143     0.9017094     0.9666667
## 6        6     0.8728571     0.7857143     0.8974359     0.9600000
## 7        7     0.8914286     0.8095238     0.9145299     0.9733333
## 8        8     0.8947619     0.8095238     0.9188034     0.9800000
## 9        9     0.8847619     0.8095238     0.9059829     0.9600000
## 10      10     0.8907143     0.8214286     0.9102564     0.9600000
##    iteration exec.time nested_cv_run
## 1          1      0.03             1
## 2          2      0.08             1
## 3          3      0.08             1
## 4          4      0.06             1
## 5          5      0.07             1
## 6          6      0.07             1
## 7          7      0.07             1
## 8          8      0.07             1
## 9          9      0.08             1
## 10        10      0.06             1
##
## $measures
```

```
## [1] "bac.test.mean" "tpr.test.mean" "acc.test.mean" "tnr.test.mean"
##
## $hyperparams
## [1] "sw.rate"
##
## $diagnostics
## [1] FALSE
##
## $optimization
## [1] "TuneControlGrid"
##
## $nested
## [1] TRUE
```

```
plotHyperParsEffect(efectos, x="sw.rate", y="bac.test.mean", loess.smooth = TRUE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Además de ajustar el *rate* de SMOTE, también podemos optimizar el threshold (esto lo podemos hacer al ajustar hiper-parámetros, siempre que el método permita predecir probabilidades, que es el caso de *knn*).

```
ps = makeParamSet(
  makeDiscreteParam("sw.rate", values = seq(1, 10, 1))
)

evaluacion_grid <- makeResampleDesc("CV", iters=3, stratify = TRUE)
control_grid <- makeTuneControlGrid(tune.threshold = TRUE, tune.threshold.args = list(measure=list(bac)))
```

```
learner_smote_knn = makeSMOTEWrapper(learner_knn_prob)

learner_ajuste_smote_knn <- makeTuneWrapper(learner_smote_knn, resampling = evaluacion_grid, par.set =
                                            measures = list(bac, tpr, acc, tnr), show.info = FALSE)

set.seed(0)
errores_ajuste_smote_knn <- resample(learner_ajuste_smote_knn, task_io, particion_datos, measures = li
```

```
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## [Resample] iter 1:    0.9145299 0.9071429 0.8809524 0.9333333
##
## Aggregated Result: acc.test.mean=0.9145299,bac.test.mean=0.9071429,tpr.test.mean=0.8809524,tnr.test.m
##
```

Parece que usando un *rate* de 10 y un threshold de 0.35, podemos mejorar un poco más el *bac*.

```
errores_ajuste_smote_knn
```

```
## Resample Result
## Task: ubIonosphere
## Learner: classif.kknn.smoted.tuned
## Aggr perf: acc.test.mean=0.9145299,bac.test.mean=0.9071429,tpr.test.mean=0.8809524,tnr.test.mean=0.93
## Runtime: 2.63562
```

```
errores_ajuste_smote_knn$models[[1]]$learner.model$opt.result
```

```
## Tune result:
## Op. pars: sw.rate=10
## Threshold: 0.35
## bac.test.mean=0.9052381,tpr.test.mean=0.8571429,acc.test.mean=0.9188034,tnr.test.mean=0.9533333
```

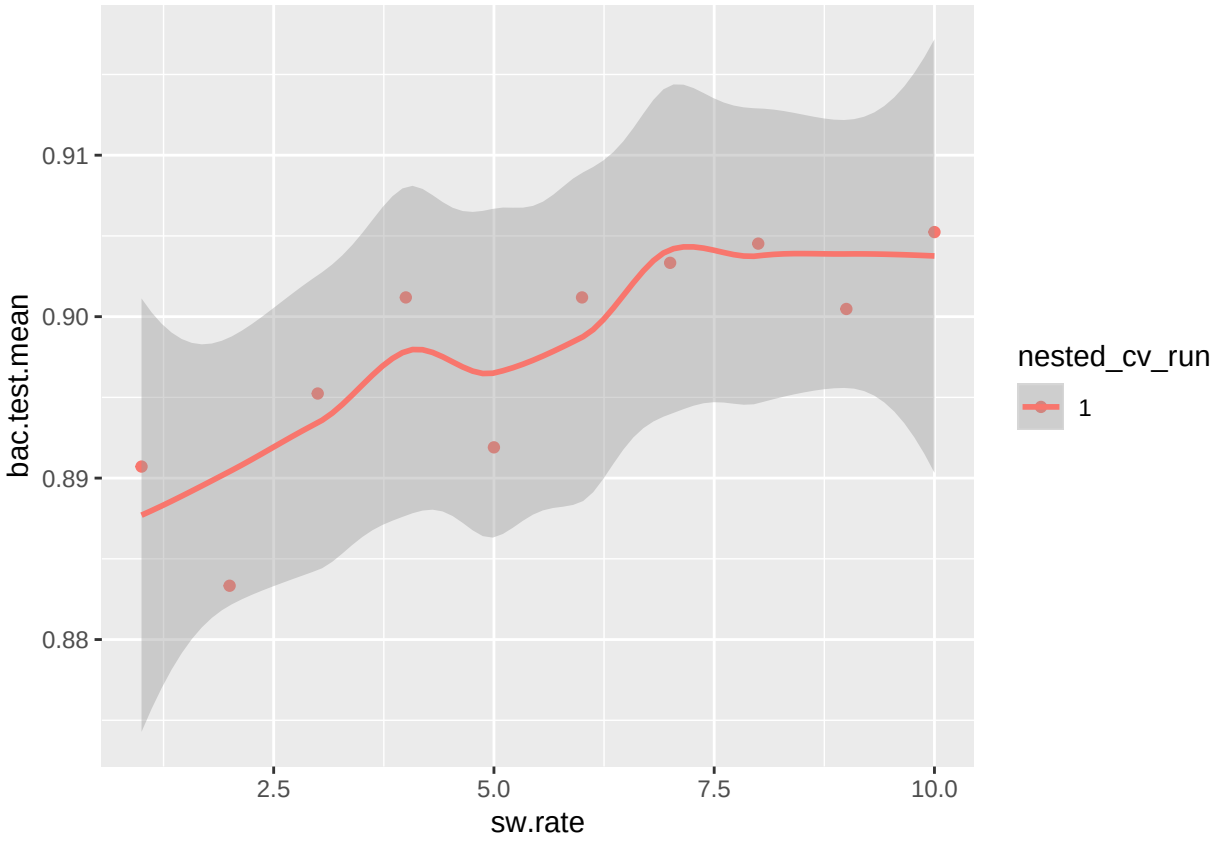
```
efectos <- generateHyperParsEffectData(errores_ajuste_smote_knn)
head(efectos)
```

```
## $data
##    sw.rate bac.test.mean tpr.test.mean acc.test.mean tnr.test.mean
## 1        1     0.8907143     0.8214286     0.9102564     0.9600000
## 2        2     0.8833333     0.8333333     0.8974359     0.9333333
## 3        3     0.8952381     0.8571429     0.9059829     0.9333333
## 4        4     0.9011905     0.8690476     0.9102564     0.9333333
## 5        5     0.8919048     0.8571429     0.9017094     0.9266667
## 6        6     0.9011905     0.8690476     0.9102564     0.9333333
## 7        7     0.9033333     0.8333333     0.9230769     0.9733333
## 8        8     0.9045238     0.8690476     0.9145299     0.9400000
## 9        9     0.9004762     0.8809524     0.9059829     0.9200000
## 10      10     0.9052381     0.8571429     0.9188034     0.9533333
##    iteration exec.time  threshold nested_cv_run
## 1          1      0.04 0.08097934             1
## 2          2      0.08 0.08097934             1
## 3          3      0.06 0.04994472             1
## 4          4      0.07 0.09994472             1
## 5          5      0.08 0.14563881             1
## 6          6      0.08 0.18097934             1
```

```
## 7           7      0.07 0.41793157             1
## 8           8      0.08 0.21916288             1
## 9           9      0.08 0.14594874             1
## 10         10      0.08 0.35312150             1
##
## $measures
## [1] "bac.test.mean" "tpr.test.mean" "acc.test.mean" "tnr.test.mean"
##
## $hyperparams
## [1] "sw.rate"
##
## $diagnostics
## [1] FALSE
##
## $optimization
## [1] "TuneControlGrid"
##
## $nested
## [1] TRUE
```

```r
plotHyperParsEffect(efectos, x="sw.rate", y="bac.test.mean", loess.smooth = TRUE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Incluso podemos ajustar el mismo tiempo el hiper-parámetro k, aunque esto va a tardar más tiempo.

```r
library("parallelMap")
parallelStartSocket(4)
```

```
## Starting parallelization in mode=socket with cpus=4.
```

```
ps = makeParamSet(
  makeDiscreteParam("sw.rate", values = seq(1, 10, 1)),
  makeDiscreteParam("k", values = seq(1, 10, by=2))
)

evaluacion_grid <- makeResampleDesc("CV", iters=3, stratify = TRUE)
control_grid <- makeTuneControlGrid(tune.threshold = TRUE, tune.threshold.args = list(measure=list(bac)

learner_smote_kknn = makeSMOTEWrapper(learner_kknn_prob)

learner_ajuste_smote_kknn <- makeTuneWrapper(learner_smote_kknn, resampling = evaluacion_grid, par.set =
                                             measures = list(bac, tpr, acc, tnr), show.info = TRUE)

set.seed(0)
errores_ajuste_smote_kknn <- resample(learner_ajuste_smote_kknn, task_io, particion_datos, measures = li
```

```
## Exporting objects to slaves for mode socket: .mlr.slave.options
## Resampling: holdout
## Measures:             acc       bac       tpr       tnr
## Mapping in parallel: mode = socket; level = mlr.resample; cpus = 4; elements = 1.
##
## Aggregated Result: acc.test.mean=0.9059829,bac.test.mean=0.8952381,tpr.test.mean=0.8571429,tnr.test.m
##
```

```
parallelStop()
```

```
## Stopped parallelization. All cleaned up.
```

Pero en este caso, ajustar k no permite mejorar *bac*.

```
errores_ajuste_smote_kknn
```

```
## Resample Result
## Task: ubIonosphere
## Learner: classif.kknn.smoted.tuned
## Aggr perf: acc.test.mean=0.9059829,bac.test.mean=0.8952381,tpr.test.mean=0.8571429,tnr.test.mean=0.93
## Runtime: 13.3638
```

```
errores_ajuste_smote_kknn$models[[1]]$learner.model$opt.result
```

```
## Tune result:
## Op. pars: sw.rate=10; k=7
## Threshold: 0.44
## bac.test.mean=0.9178571,tpr.test.mean=0.8690476,acc.test.mean=0.9316239,tnr.test.mean=0.9666667
```

```
efectos <- generateHyperParsEffectData(errores_ajuste_smote_kknn)
head(efectos)
```

```
## $data
##    sw.rate k bac.test.mean tpr.test.mean acc.test.mean tnr.test.mean
## 1        1 1     0.8207143     0.6547619     0.8675214     0.9866667
## 2        2 1     0.8319048     0.6904762     0.8717949     0.9733333
```

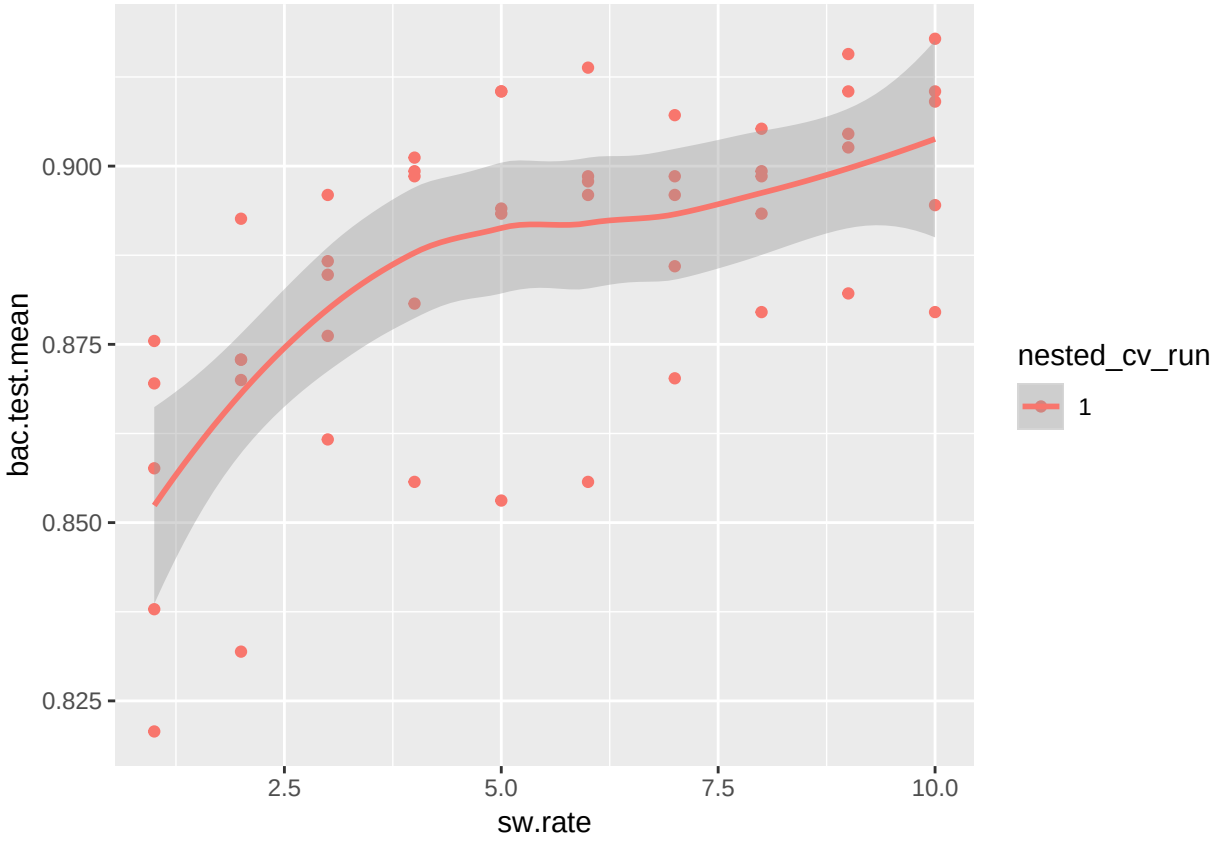
```
## 3        3 1     0.8616667     0.7500000     0.8931624     0.9733333
## 4        4 1     0.8557143     0.7380952     0.8888889     0.9733333
## 5        5 1     0.8530952     0.7261905     0.8888889     0.9800000
## 6        6 1     0.8557143     0.7380952     0.8888889     0.9733333
## 7        7 1     0.8702381     0.7738095     0.8974359     0.9666667
## 8        8 1     0.8795238     0.7857143     0.9059829     0.9733333
## 9        9 1     0.8821429     0.7976190     0.9059829     0.9666667
## 10      10 1     0.8795238     0.7857143     0.9059829     0.9733333
## 11       1 3     0.8378571     0.7023810     0.8760684     0.9733333
## 12       2 3     0.8728571     0.7857143     0.8974359     0.9600000
## 13       3 3     0.8847619     0.8095238     0.9059829     0.9600000
## 14       4 3     0.8992857     0.8452381     0.9145299     0.9533333
## 15       5 3     0.8933333     0.8333333     0.9102564     0.9533333
## 16       6 3     0.8985714     0.8571429     0.9102564     0.9400000
## 17       7 3     0.8959524     0.8452381     0.9102564     0.9466667
## 18       8 3     0.8985714     0.8571429     0.9102564     0.9400000
## 19       9 3     0.9045238     0.8690476     0.9145299     0.9400000
## 20      10 3     0.9104762     0.8809524     0.9188034     0.9400000
## 21       1 5     0.8576190     0.7619048     0.8846154     0.9533333
## 22       2 5     0.8728571     0.7857143     0.8974359     0.9600000
## 23       3 5     0.8761905     0.7857143     0.9017094     0.9666667
## 24       4 5     0.8807143     0.8214286     0.8974359     0.9400000
## 25       5 5     0.9104762     0.8809524     0.9188034     0.9400000
## 26       6 5     0.8959524     0.8452381     0.9102564     0.9466667
## 27       7 5     0.8859524     0.8452381     0.8974359     0.9266667
## 28       8 5     0.8992857     0.8452381     0.9145299     0.9533333
## 29       9 5     0.9026190     0.8452381     0.9188034     0.9600000
## 30      10 5     0.8945238     0.8690476     0.9017094     0.9200000
## 31       1 7     0.8695238     0.7857143     0.8931624     0.9533333
## 32       2 7     0.8926190     0.8452381     0.9059829     0.9400000
## 33       3 7     0.8959524     0.8452381     0.9102564     0.9466667
## 34       4 7     0.8985714     0.8571429     0.9102564     0.9400000
## 35       5 7     0.9104762     0.8809524     0.9188034     0.9400000
## 36       6 7     0.8978571     0.8690476     0.9059829     0.9266667
## 37       7 7     0.9071429     0.8809524     0.9145299     0.9333333
## 38       8 7     0.9052381     0.8571429     0.9188034     0.9533333
## 39       9 7     0.9157143     0.9047619     0.9188034     0.9266667
## 40      10 7     0.9178571     0.8690476     0.9316239     0.9666667
## 41       1 9     0.8754762     0.7976190     0.8974359     0.9533333
## 42       2 9     0.8700000     0.8333333     0.8803419     0.9066667
## 43       3 9     0.8866667     0.8333333     0.9017094     0.9400000
## 44       4 9     0.9011905     0.8690476     0.9102564     0.9333333
## 45       5 9     0.8940476     0.8214286     0.9145299     0.9666667
## 46       6 9     0.9138095     0.8809524     0.9230769     0.9466667
## 47       7 9     0.8985714     0.8571429     0.9102564     0.9400000
## 48       8 9     0.8933333     0.8333333     0.9102564     0.9533333
## 49       9 9     0.9104762     0.8809524     0.9188034     0.9400000
## 50      10 9     0.9090476     0.9047619     0.9102564     0.9133333
##    iteration exec.time  threshold nested_cv_run
## 1          1      0.14 0.99993389             1
## 2          2      0.06 0.99993389             1
## 3          3      0.10 0.99993389             1
## 4          4      0.06 0.99993389             1
## 5          5      0.07 0.99993389             1
```

```
## 6          6      0.06 0.99993389             1
## 7          7      0.06 0.99993389             1
## 8          8      0.08 0.99993389             1
## 9          9      0.06 0.99993389             1
## 10        10      0.06 0.99993389             1
## 11        11      0.03 0.04994472             1
## 12        12      0.07 0.04994472             1
## 13        13      0.06 0.04994472             1
## 14        14      0.06 0.04994472             1
## 15        15      0.07 0.09994472             1
## 16        16      0.08 0.04994472             1
## 17        17      0.06 0.04994472             1
## 18        18      0.07 0.04994472             1
## 19        19      0.06 0.04994472             1
## 20        20      0.08 0.04994472             1
## 21        21      0.04 0.01188104             1
## 22        22      0.06 0.08097934             1
## 23        23      0.06 0.23614543             1
## 24        24      0.08 0.08097934             1
## 25        25      0.06 0.03097934             1
## 26        26      0.07 0.08097934             1
## 27        27      0.07 0.03097934             1
## 28        28      0.07 0.35165245             1
## 29        29      0.06 0.37421104             1
## 30        30      0.08 0.08097934             1
## 31        31      0.03 0.01917594             1
## 32        32      0.06 0.06916289             1
## 33        33      0.06 0.07020643             1
## 34        34      0.06 0.08097934             1
## 35        35      0.06 0.09994472             1
## 36        36      0.06 0.08097934             1
## 37        37      0.08 0.09994472             1
## 38        38      0.07 0.14595854             1
## 39        39      0.07 0.09994472             1
## 40        40      0.06 0.43825187             1
## 41        41      0.05 0.02570890             1
## 42        42      0.05 0.02542802             1
## 43        43      0.08 0.13097934             1
## 44        44      0.07 0.11902066             1
## 45        45      0.07 0.38201185             1
## 46        46      0.07 0.21249641             1
## 47        47      0.08 0.18827424             1
## 48        48      0.08 0.28265517             1
## 49        49      0.06 0.19994472             1
## 50        50      0.06 0.13938446             1
##
## $measures
## [1] "bac.test.mean" "tpr.test.mean" "acc.test.mean" "tnr.test.mean"
##
## $hyperparams
## [1] "sw.rate" "k"
##
## $diagnostics
## [1] FALSE
```

```
##
## $optimization
## [1] "TuneControlGrid"
##
## $nested
## [1] TRUE
```

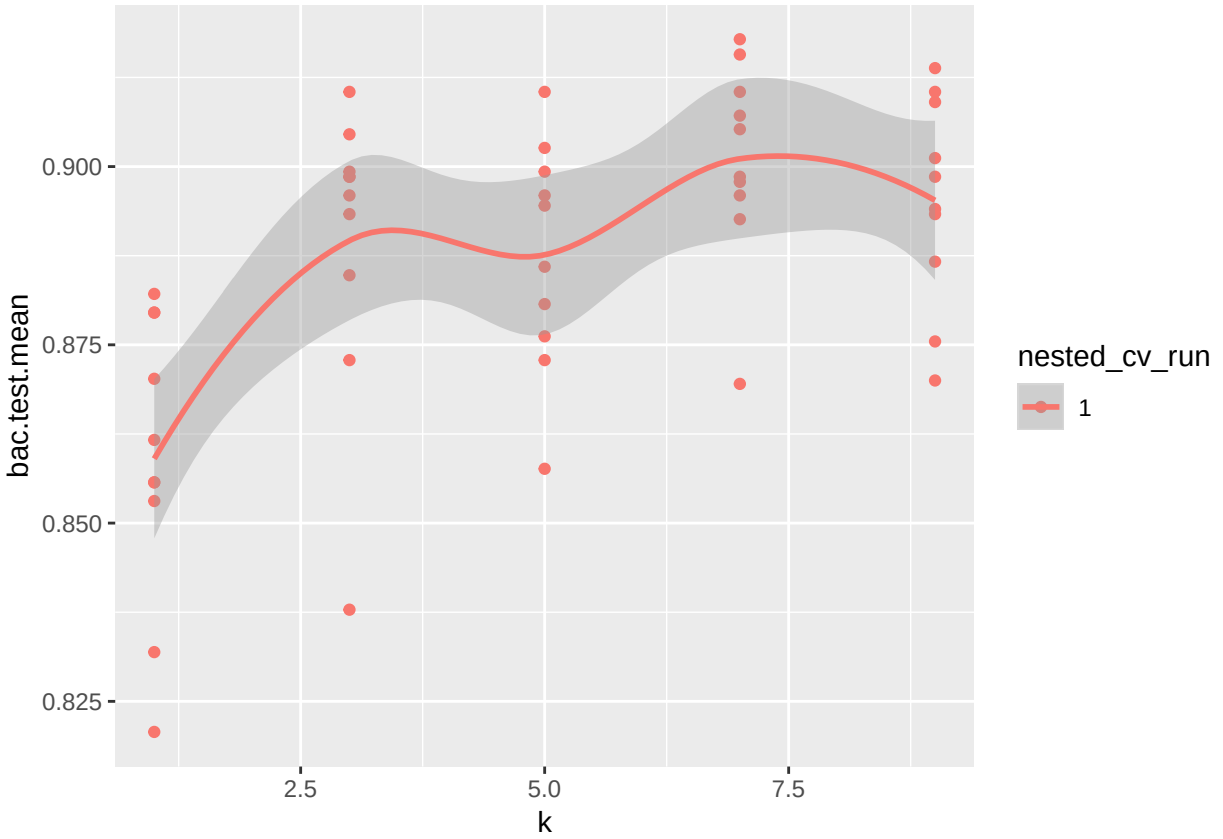
```
plotHyperParsEffect(efectos, x="sw.rate", y="bac.test.mean", loess.smooth = TRUE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

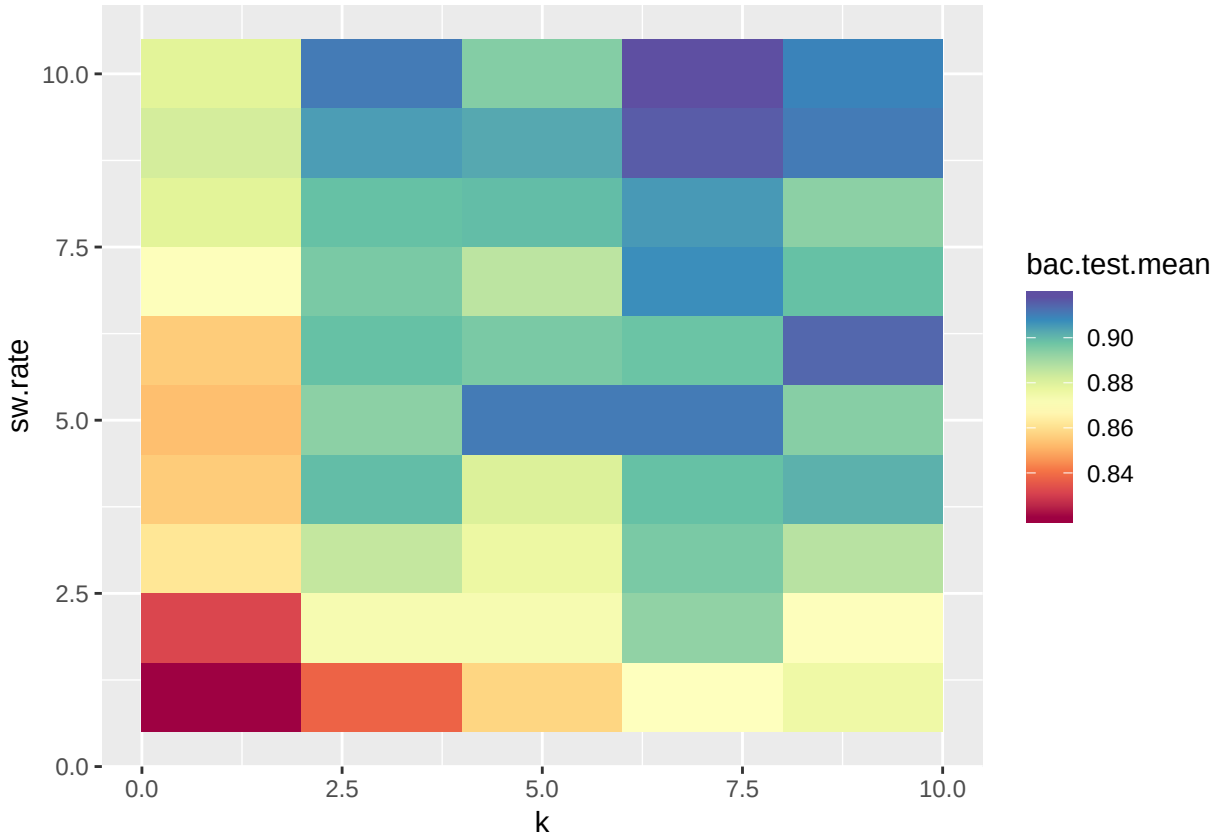



```
plotHyperParsEffect(efectos, x="k", y="bac.test.mean", loess.smooth = TRUE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
plotHyperParsEffect(efectos, x="k", y="sw.rate", z="bac.test.mean", plot.type = "heatmap")
```



Nota: hay métodos que permiten dar pesos a las instancias. En ese caso, una aproximación más rápida que SMOTE podría ser usar *makeWeightedClassesWrapper* en lugar de *makeSMOTEWrapper*. En este caso, el nombre del hiper-parámetro que mide el peso (o importancia) de los datos de la clase minoritaria frente a los de la mayoritaria es *wcw.weight*. Desgraciadamente, **kknn** no permite usar pesos con las instancias (pero *rpart* si).