# Comparación de algoritmos y estrategias avanzadas

# “No free lunch theorems”

(The Lack of A Priori Distinctions Between Learning Algorithms (David H. Wolpert, 1996))

Se trata de una asunción sobre la imposibilidad de encontrar un algoritmo universal que bata a los demás en todo tipo de datos.

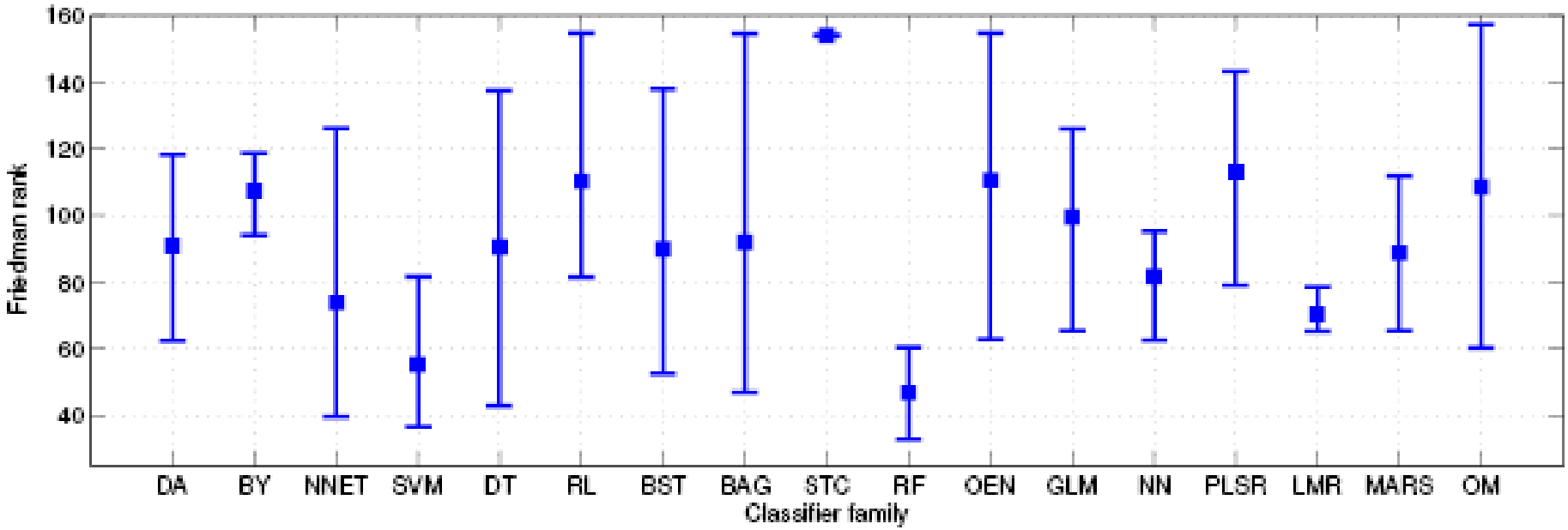
# Resultados del estudio en el documento

"Do we need hundreds of Classifiers to solve Real World Classifications Problems? (Delgado et al, 2014) 121 datasets, 179 algoritmos.

| Data set | #pat. | #inp. | #cl. | %Maj. | Data set | #pat. | #inp. | #cl. | %Maj. |
|---|---|---|---|---|---|---|---|---|---|
| abalone | 4177 | 8 | 3 | 34.6 | energy-y1 | 768 | 8 | 3 | 46.9 |
| ac-inflam | 120 | 6 | 2 | 50.8 | energy-y2 | 768 | 8 | 3 | 49.9 |
| acute-nephritis | 120 | 6 | 2 | 58.3 | fertility | 100 | 9 | 2 | 88.0 |
| adult | 48842 | 14 | 2 | 75.9 | flags | 194 | 28 | 8 | 30.9 |
| annealing | 798 | 38 | 6 | 76.2 | glass | 214 | 9 | 6 | 35.5 |
| arrhythmia | 452 | 262 | 13 | 54.2 | haberman-survival | 306 | 3 | 2 | 73.5 |
| audiology-std | 226 | 59 | 18 | 26.3 | hayes-roth | 132 | 3 | 3 | 38.6 |
| balance-scale | 625 | 4 | 3 | 46.1 | heart-cleveland | 303 | 13 | 5 | 54.1 |
| balloons | 16 | 4 | 2 | 56.2 | heart-hungarian | 294 | 12 | 2 | 63.9 |
| bank | 45211 | 17 | 2 | 88.5 | heart-switzerland | 123 | 12 | 2 | 39.0 |
| blood | 748 | 4 | 2 | 76.2 | heart-va | 200 | 12 | 5 | 28.0 |
| breast-cancer | 286 | 9 | 2 | 70.3 | hepatitis | 155 | 19 | 2 | 79.3 |
| bc-wisc | 699 | 9 | 2 | 65.5 | hill-valley | 606 | 100 | 2 | 50.7 |
| bc-wisc-diag | 569 | 30 | 2 | 62.7 | horse-colic | 300 | 25 | 2 | 63.7 |
| bc-wisc-prog | 198 | 33 | 2 | 76.3 | ilpd-indian-liver | 583 | 9 | 2 | 71.4 |
| breast-tissue | 106 | 9 | 6 | 20.7 | image-segmentation | 210 | 19 | 7 | 14.3 |
| car | 1728 | 6 | 4 | 70.0 | ionosphere | 351 | 33 | 2 | 64.1 |
| ctg-10classes | 2126 | 21 | 10 | 27.2 | iris | 150 | 4 | 3 | 33.3 |
| ctg-3classes | 2126 | 21 | 3 | 77.8 | led-display | 1000 | 7 | 10 | 11.1 |
| chess-krvk | 28056 | 6 | 18 | 16.2 | lenses | 24 | 4 | 3 | 62.5 |
| chess-krvkp | 3196 | 36 | 2 | 52.2 | letter | 20000 | 16 | 26 | 4.1 |
| congress-voting | 435 | 16 | 2 | 61.4 | libras | 360 | 90 | 15 | 6.7 |
| conn-bench-sonar | 208 | 60 | 2 | 53.4 | low-res-spect | 531 | 100 | 9 | 51.9 |
| conn-bench-vowel | 528 | 11 | 11 | 9.1 | lung-cancer | 32 | 56 | 3 | 40.6 |
| connect-4 | 67557 | 42 | 2 | 75.4 | lymphography | 148 | 18 | 4 | 54.7 |
| contrac | 1473 | 9 | 3 | 42.7 | magic | 19020 | 10 | 2 | 64.8 |
| credit-approval | 690 | 15 | 2 | 55.5 | mammographic | 961 | 5 | 2 | 53.7 |
| cylinder-bands | 512 | 35 | 2 | 60.9 | miniboone | 130064 | 50 | 2 | 71.9 |
| dermatology | 366 | 34 | 6 | 30.6 | molec-biol-promoter | 106 | 57 | 2 | 50.0 |
| echocardiogram | 131 | 10 | 2 | 67.2 | molec-biol-splice | 3190 | 60 | 3 | 51.9 |
| ecoli | 336 | 7 | 8 | 42.6 | monks-1 | 124 | 6 | 2 | 50.0 |

The validation methodology is the following. One training and one test set are generated randomly (each with 50% of the available patterns), but imposing that each class has the same number of training and test patterns (in order to have enough training and test patterns of every class). This couple of sets is used only for **parameter tuning** (in those classifiers which have tunable parameters), selecting the parameter values which provide the best accuracy on the test set. The indexes of the training and test patterns (i.e., the data partitioning) are given by the file `conxuntos.dat` for each data set, and are the same for all the classifiers. Then, using the selected values for the tunable parameters, a **4-fold cross validation** is developed using the whole available data. The indexes of the training and test patterns for each fold are the same for all the classifiers, and they are listed in the file `conxuntos_kfold.dat` for each data set. The test results is the average over the 4 test sets. However, for some data sets, which provide **separate data for training and testing** (data sets annealing and audiology-std, among others), the classifier (with the tuned parameter values) is trained and tested on the respective data sets. In this case, the test result is calculated on the test set. We used this methodology in order to keep

| Rank | Acc. | $\kappa$ | Classifier | Rank | Acc. | $\kappa$ | Classifier |
|---|---|---|---|---|---|---|---|
| **32.9** | 82.0 | 63.5 | parRF_t (RF) | 67.3 | 77.7 | 55.6 | pda_t (DA) |
| 33.1 | **82.3** | **63.6** | rf_t (RF) | 67.6 | 78.7 | 55.2 | elm_m (NNET) |
| 36.8 | 81.8 | 62.2 | svm_C (SVM) | 67.6 | 77.8 | 54.2 | SimpleLogistic_w (LMR) |
| 38.0 | 81.2 | 60.1 | svmPoly_t (SVM) | 69.2 | 78.3 | 57.4 | MAB_J48_w (BST) |
| 39.4 | 81.9 | 62.5 | rforest_R (RF) | 69.8 | 78.8 | 56.7 | BG_REPTree_w (BAG) |
| 39.6 | 82.0 | 62.0 | elm_kernel_m (NNET) | 69.8 | 78.1 | 55.4 | SMO_w (SVM) |
| 40.3 | 81.4 | 61.1 | svmRadialCost_t (SVM) | 70.6 | 78.3 | 58.0 | MLP_w (NNET) |
| 42.5 | 81.0 | 60.0 | svmRadial_t (SVM) | 71.0 | 78.8 | 58.23 | BG_RandomTree_w (BAG) |
| 42.9 | 80.6 | 61.0 | C5.0_t (BST) | 71.0 | 77.1 | 55.1 | mlm_R (GLM) |
| 44.1 | 79.4 | 60.5 | avNNet_t (NNET) | 71.0 | 77.8 | 56.2 | BG_J48_w (BAG) |
| 45.5 | 79.5 | 61.0 | nnet_t (NNET) | 72.0 | 75.7 | 52.6 | rbf_t (NNET) |
| 47.0 | 78.7 | 59.4 | pcaNNet_t (NNET) | 72.1 | 77.1 | 54.8 | fda_R (DA) |
| 47.1 | 80.8 | 53.0 | BG_LibSVM_w (BAG) | 72.4 | 77.0 | 54.7 | lda_R (DA) |
| 47.3 | 80.3 | 62.0 | mlp_t (NNET) | 72.4 | 79.1 | 55.6 | svmlight_C (NNET) |
| 47.6 | 80.6 | 60.0 | RotationForest_w (RF) | 72.6 | 78.4 | 57.9 | AdaBoostM1_J48_w (BST) |
| 50.1 | 80.9 | 61.6 | RRF_t (RF) | 72.7 | 78.4 | 56.2 | BG_IBk_w (BAG) |
| 51.6 | 80.7 | 61.4 | RRFglobal_t (RF) | 72.9 | 77.1 | 54.6 | ldaBag_R (BAG) |
| 52.5 | 80.6 | 58.0 | MAB_LibSVM_w (BST) | 73.2 | 78.3 | 56.2 | BG_LWL_w (BAG) |
| 52.6 | 79.9 | 56.9 | LibSVM_w (SVM) | 73.7 | 77.9 | 56.0 | MAB_REPTree_w (BST) |
| 57.6 | 79.1 | 59.3 | adaboost_R (BST) | 74.0 | 77.4 | 52.6 | RandomSubSpace_w (DT) |
| 58.5 | 79.7 | 57.2 | pnn_m (NNET) | 74.4 | 76.9 | 54.2 | lda2_t (DA) |
| 58.9 | 78.5 | 54.7 | cforest_t (RF) | 74.6 | 74.1 | 51.8 | svmBag_R (BAG) |
| 59.9 | 79.7 | 42.6 | dkp_C (NNET) | 74.6 | 77.5 | 55.2 | LibLINEAR_w (SVM) |
| 60.4 | 80.1 | 55.8 | gausspr Radial_R (OM) | 75.9 | 77.2 | 55.6 | rbfDDA_t (NNET) |
| 60.5 | 80.0 | 57.4 | RandomForest_w (RF) | 76.5 | 76.9 | 53.8 | sda_t (DA) |
| 62.1 | 78.7 | 56.0 | svmLinear_t (SVM) | 76.6 | 78.1 | 56.5 | END_w (OEN) |
| 62.5 | 78.4 | 57.5 | fda_t (DA) | 76.6 | 77.3 | 54.8 | LogitBoost_w (BST) |
| 62.6 | 78.6 | 56.0 | knn_t (NN) | 76.6 | 78.2 | 57.3 | MAB_RandomTree_w (BST) |
| 62.8 | 78.5 | 58.1 | mlp_C (NNET) | 77.1 | 78.4 | 54.0 | BG_RandomForest_w (BAG) |
| 63.0 | 79.9 | 59.4 | RandomCommittee_w (OEN) | 78.5 | 76.5 | 53.7 | Logistic_w (LMR) |
| 63.4 | 78.7 | 58.4 | Decorate_w (OEN) | 78.7 | 76.6 | 50.5 | ctreeBag_R (BAG) |
| 63.6 | 76.9 | 56.0 | mlpWeightDecay_t (NNET) | 79.0 | 76.8 | 53.5 | BG_Logistic_w (BAG) |
| 63.8 | 78.7 | 56.7 | rda_R (DA) | 79.1 | 77.4 | 53.0 | lvq_t (NNET) |
| 64.0 | 79.0 | 58.6 | MAB_MLP_w (BST) | 79.1 | 74.4 | 50.7 | pls_t (PLSR) |
| 64.1 | 79.9 | 56.9 | MAB_RandomForest_w (BST) | 79.8 | 76.9 | 54.7 | hdda_R (DA) |
| 65.0 | 79.0 | 56.8 | knn_R (NN) | 80.6 | 75.9 | 53.3 | MCC_w (OEN) |
| 65.2 | 77.9 | 56.2 | multinom_t (LMR) | 80.9 | 76.9 | 54.5 | mda_R (DA) |
| 65.5 | 77.4 | 56.6 | gcvEarth_t (MARS) | 81.4 | 76.7 | 55.2 | C5.0Rules_t (RL) |
| 65.5 | 77.8 | 55.7 | glmnet_R (GLM) | 81.6 | 78.3 | 55.8 | lssvmRadial_t (SVM) |
| 65.6 | 78.6 | 58.4 | MAB_PART_w (BST) | 81.7 | 75.6 | 50.9 | JRip_t (RL) |
| 66.0 | 78.5 | 56.5 | CVR_w (OM) | 82.0 | 76.1 | 53.3 | MAB_Logistic_w (BST) |
| 66.4 | 79.2 | 58.9 | treebag_t (BAG) | 84.2 | 75.8 | 53.9 | C5.0Tree_t (DT) |
| 66.6 | 78.2 | 56.8 | BG_PART_w (BAG) | 84.6 | 75.7 | 50.8 | BG_DecisionTable_w (BAG) |
| 66.7 | 75.5 | 55.2 | mda_t (DA) | 84.9 | 76.5 | 53.4 | NBTree_w (DT) |


160
140
120
100
80
60
40
Friedman rank
DA
BY
NNET
SVM
DT
RL
BST
BAG
STC
RF
OEN
GLM
NN
PLSR
LMR
MARS
OM
Classifier family

## 4. Conclusion

This paper presents an exhaustive evaluation of 179 classifiers belonging to a wide collection of 17 families over the whole UCI machine learning classification database, discarding the large-scale data sets due to technical reasons, plus 4 own real sets, summing up to 121 data sets from 10 to 130,064 patterns, from 3 to 262 inputs and from 2 to 100 classes. **The best results are achieved by the parallel random forest** (parRF_t), implemented in R with caret, tuning the parameter mtry. The parRF_t achieves in average 94.1% of the maximum accuracy over all the data sets (Table 5, lower part), and overcomes the 90% of the maximum accuracy in 102 out of 121 data sets. Its average accuracy over all the data sets is 82.0%, while the maximum average accuracy (achieved by the best classifier for each data set) is 86.9%. The random forest in R and tuned with caret (rf_t) is slightly worse (93.6% of the maximum accuracy), although it achieves slightly better average accuracy (82.3%) than parRF_t. The LibSVM implementation of SVM in C with Gaussian kernel (svm_C), tuning the regularization and kernel spread, achieves 92.3% of the maximum accuracy. Six RFs and five SVMs are included among the 20 best classifiers, which are the bests families. The parRF_t may be considered as a reference ("gold-standard") to compare with new classifier proposals in order to assess their performance for general classification in general (not requiring special features as large-scale, on-line learning, non-stationary data, etc.). Other classifiers with good results are the extreme learning machine with Gaussian kernel, the C5.0 decision tree and the multi-layer perceptron (avNNet_t, a committee of 5 multi-layer perceptrons randomly initialized tuning the size and decay rate). The best boosting and bagging ensembles use LibSVM as base classifiers (in Weka), being slightly better than the single LibSVM classifier, and adaboost_R (ensemble of decision trees trained using Adaboost.M1). For **two-class data sets**, avNNet_t is the best (95% of the maximum accuracy), being the parRF_t also very good (94.3%). It is also the best when the complexity, #patterns and #inputs of the data set increase, being also good when #patterns decrease (rf_t is the best) and #classes increase (svm_C is the best). The probabilistic neural network in Matlab, tuning the Gaussian kernel spread (pnn_m), and the direct kernel perceptron in C (dkp_C), a very simple and fast neural network proposed by us (Fernández-Delgado et al., 2014), are also very near to the top-20. The remaining families of classifiers, including other neural networks (radial basis functions, learning vector quantization and cascade correlation), discriminant analysis, decision trees other than C5.0, rule-based classifiers, other bagging and boosting ensembles, nearest neighbors, Bayesian, GLM, PLSR, MARS, etc., are not competitive at all. Most of the best classifiers are implemented in R and tuned using caret, which seems the best alternative to select a classifier implementation.

134. **parRF_t** uses a parallel implementation of random forest using the `randomForest` package with mtry=2:2:8.

50. **LibSVM_w** uses the library LibSVM (Chang and Lin, 2008), calls from Weka for classification with Gaussian kernel, using the values of C and gamma selected for svm_C and tolerance=0.001.

34. **avNNet_t**, from the `caret` package, creates a committee of 5 MLPs (the number of MLPs is given by parameter repeat) trained with different random weight initializations and bag=false. The tunable parameters are the #hidden neurons (size) in {1, 3, 5} and the weight decay (values $\{0, 0.1, 10^{-4}\}$). This low number of hidden neurons is to reduce the computational cost of the ensemble.

# Otro artículo (Junio 2016) abordando el mismo problema y que hace alusión al anterior:

**bst** Hyperparameters: shrinkage = $\{0.05, 0.1\}$. Free hyperparameter, number of boosts, from 100 to 3000 by 200, at most ndat.

**elm** Hyperparameter: number of hidden units = at most 24 values equally spaced between 20 and ndat/2

**gbm** Hyperparameters: interaction-depth = 1..5, shrinkage=$\{0.05, 0.1\}$. number of boosts is a free hyperparameter, tested from 50 to 500 with increments of 20 to at most ndat.

**glmnet** Hyperparameters $\alpha$, 8 values equally spaced between 0 and 1. Free hyperparameter $\lambda$, 20 values geometrically spaced between $10^{-5}$ to $10^3$.

**knn** Hyperparameter k:= 1, and at most 23 random values from 3 to ndat/4.

**lvq** Hyperparameter: size of the codebook = at most 8 values equally spaced between 2 and 2× nfeat

**nb** Hyperparameters: usekernel = { true, false }, fL = $\{0, 0.1, 1, 2\}$

**nnet** Hyperparameter: number of hidden units = at most 8 values equally spaced between 3 and nfeat/2, decay = $\{0, 0.01, 0.1\}$.

**rf** Hyperparameter mtry = $\{0.5, 1, 2, 3, 4, 5\}\sqrt{nfeat}$ up to a value of nfeat/2. Number of trees is a free hyperparameters, tested from 500 to 3000 by 500 up to ndat/2.

**rknn** Hyperparameters: mtry = 4 values equally spaced between 2 and nfeat−2, k= 1, and at most 23 random values from 3 to ndat/4. The number of classifiers is a free hyperparameter from 5 to 100, in steps of 20.

**sda** Hyperparameter: $\lambda = 8$ values geometrically spaced from $10^{-8}$ to $10^3$

**svmLinear** . Hyperparameter: C = $2^{-5}, 2^0, 2^5, 2^{10}, 2^{15}$.

**svmPoly** Hyperparameter C as in the linear kernel and degree from 2 to 5.

**svmRadial** Hyperparameters C as in the linear SVM and $\gamma = 2^{-15}, 2^{-10.5}, 2^{-6}, 2^{-1.5}, 2^3$.

Comparison of 14 different families of classification algorithms on 115 binary datasets

Jacques Wainer
email: wainer@ic.unicamp.br
Computing Institute
University of Campinas
Campinas, SP, 13083-852, Brazil

June 6, 2016

Figure 1 displays the heat map of the rank distribution.

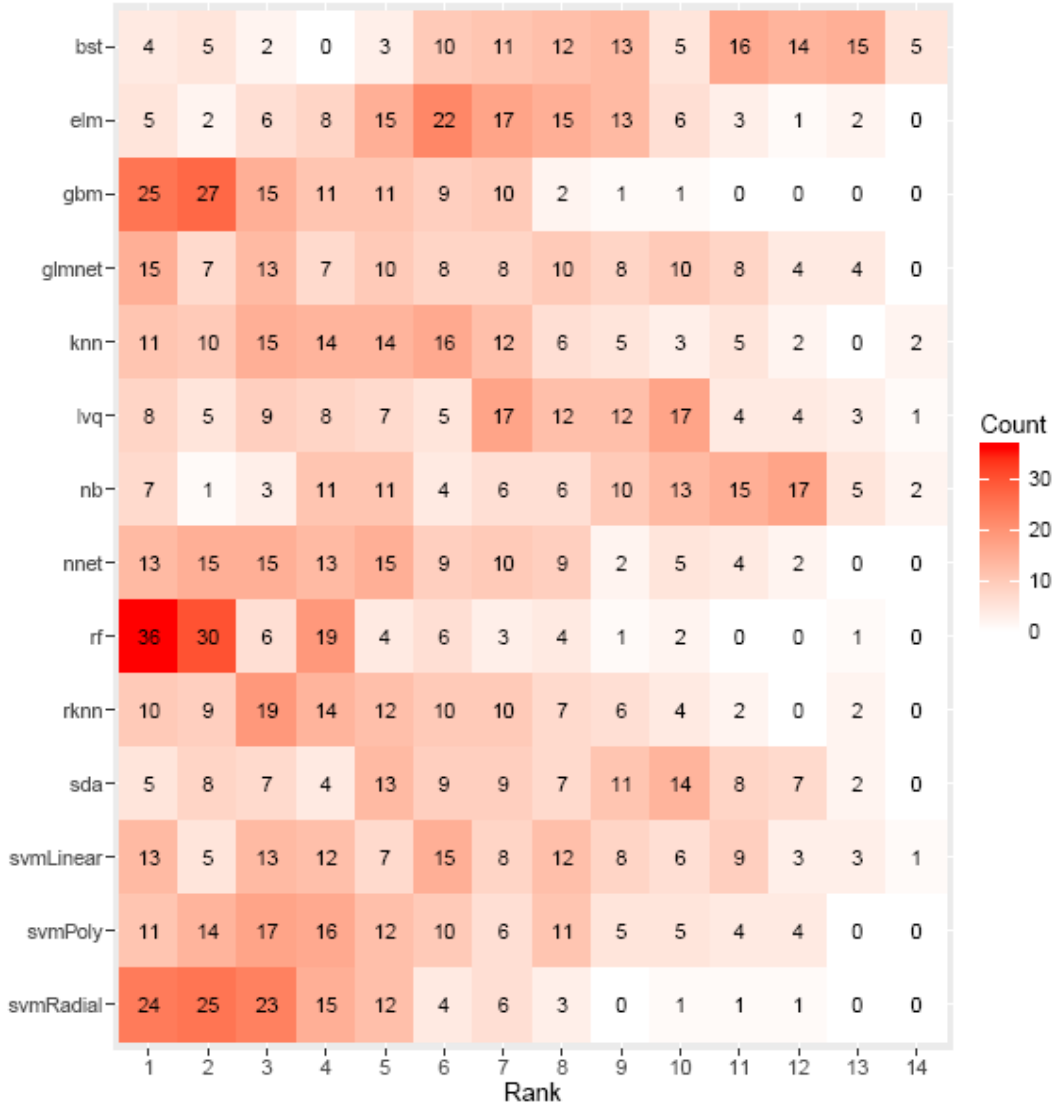



Figure 1: The heat map with the distribution of the number of times each classifier achieved a particular rank.

| | rf | svmRadial | gbm | nnet | rknn | svmPoly | knn | svmLinear | glmnet | elm | lvq | sda | nb |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| svmRadial | 1.00 | | | | | | | | | | | | |
| gbm | 1.00 | 1.00 | | | | | | | | | | | |
| nnet | **0.00** | **0.01** | **0.04** | | | | | | | | | | |
| rknn | **0.00** | **0.00** | **0.00** | 1.00 | | | | | | | | | |
| svmPoly | **0.00** | **0.00** | **0.02** | 1.00 | 1.00 | | | | | | | | |
| knn | **0.00** | **0.00** | **0.01** | 1.00 | 1.00 | 1.00 | | | | | | | |
| svmLinear | **0.00** | **0.00** | **0.00** | 0.75 | 1.00 | 0.86 | 0.93 | | | | | | |
| glmnet | **0.00** | **0.00** | **0.00** | 0.73 | 1.00 | 0.84 | 0.92 | 1.00 | | | | | |
| elm | **0.00** | **0.00** | **0.00** | 0.07 | 0.54 | 0.12 | 0.19 | 1.00 | 1.00 | | | | |
| lvq | **0.00** | **0.00** | **0.00** | **0.00** | 0.09 | **0.01** | **0.01** | 0.72 | 0.75 | 1.00 | | | |
| sda | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | 0.15 | 0.17 | 0.90 | 1.00 | | |
| nb | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | 0.05 | 0.40 | 0.94 | |
| bst | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.03** | 0.82 |

Table 2: The p-values of the Nemenyi pairwise comparison procedure. P-values below 0.05 are in bold and denote that the difference in error rate of the two corresponding classifiers is statistically significant

# Esquemas básicos de construcción y comparación de modelos

## Esquema [1] planteado en este módulo de machine learning

1) Selección de variables con stepwise y stepwise repetido

2) Tuneado de parámetros con validación cruzada y caret, a través de grids (rejillas de valores).

3) Comparación de algoritmos mediante validación cruzada repetida, observación gráfica de sesgo-varianza.

**Eficaz, pero exigente computacionalmente**

## Esquema [2] rápido

1) Selección de variables con gráficos de importancia de un modelo tuneado de gbm y/o random forest. Punto de corte sobre todo para eliminar variables irrelevantes, o no selección, o selección de las variables con importancia>0 (que pueden ser muchas).

2) Tuneado de parámetros vía CV o training-test si los datos son grandes

3) Selección del algoritmo-modelo con menor error obtenido en los sucesivos tuneados.

El método rápido [2] tiene la principal ventaja de la velocidad.

Desventajas del método rápido [2]

1) Selección de variables mediante importancia puede seleccionar variables redundantes (que tienen la misma información).

2) Las medidas de importancia tienden a favorecer variables continuas sobre discretas, al usarse aquellas en más subdivisiones de árboles por ofrecer muchos puntos de corte.

3) La comparación de algoritmos solo tendría en cuenta el sesgo: se pueden escoger modelos con varianza alta, que son peligrosos en la práctica.

## Esquema [3] datos muy grandes

Supongamos que el dataset excede nuestra capacidad de computación para usar CV u otras estrategias más exigentes. Por ejemplo, 300.000 observaciones y muchas variables.

Una propuesta que se suele usar es la siguiente, sin que esto signifique que no se puedan probar otras cosas:

1) Depuración de datos y selección de variables con todo el dataset. Obviamente se puede usar remuestreo aquí para el proceso de selección de variables. De gran importancia es reducir el dataset en cuanto a variables eliminando todo el ruido que sea posible (variables con muchos missings, unir categorías, eliminación de dummies con pocas observaciones, sustituir variables categoricas con alta cardinalidad por continuas, etc.)

2) Partición del dataset en grupos/submuestras independientes que sean manejables en nuestro entorno computacional.
3) Tuneado y observación en paralelo de esas submuestras; puede usarse training-test, CV, lo que se pueda.
4) Selección de los algoritmos/datasets/tuneados más prometedores
5) Comparación de los mejores modelos con todo el dataset, vía training-test o training-test repetido.

Como suele ocurrir en machine learning, el proceso suele ser cíclico, no secuencial, y es posible tener que volver varias veces a apartados anteriores desde los apartados 4) o 5).
Igualmente no se deben olvidar en el proceso los conceptos de sobreajste y sesgo-varianza.

# El ChaLearn AutoML Challenge

(Documentos challenge.pdf y appendix.pdf)

Competiciones ChaLearn AutoML Challenge entre 2015 y 2018

1) Cada equipo participante tenía que aportar un procedimiento automático de construcción de algoritmo predictivo y código.

2)
Limitación de hardware, potencia y memoria,
Limitación del tiempo de proceso (20 minutos aproximadamente).
No limitación en lenguaje (usaron mucho python y python con R)

3) Variedad de datasets (regresión, clasificación binaria y multiclase)

Problemas a los que se enfrentaban los concursantes:

1) Qué medidas de preprocesado automático usar

2) Qué métodos de selección automática de variables usar (si se usa alguno)

3) Qué algoritmos usar

3) Cómo tunear parámetros

4) Qué medidas de remuestreo utilizar para comparar algoritmos-modelos

## 1) Qué medidas de preprocesado automático usar

Los concursantes abordaron este problema generalmente así con poco preprocesado:

a) Tratamiento de missings básico :imputación por la mediana en continuas o variables indicadoras de missing en discretas y continuas.

b) Normalización simple de variables continuas

c) Codificación dummy para categóricas.

## 2) Qué métodos de selección automática de variables usar (si se usa alguno)

a) 2/3  usaron reducción de dimensionalidad tipo PCA (componentes principales).

b) 1/3 usaron selección de variables propiamente dicha (puede combinarse con a))

c) Como se utilizaron mucho métodos basados en árboles y estos tienen incluída una selección de variables, muchos no hicieron selección de variables.

d) Como la gran mayoría de participantes usó ensamblados (en árboles o ensamblado más general) , y estos  suelen ser robustos a variables irrelevantes, no se hizo demasiado esfuerzo en selección de variables.

e) Sí parece que usaron métodos sencillos para eliminar variables irrelevantes antes de todo.

## 3) Qué algoritmos usar

a) Un 75% usaron métodos basados en árboles (random forest, gradient boosting…). Solos o ensamblados con otros. Aparte de la potencia predictiva de estos métodos, desde el punto de vista de una competición limitada en el tiempo son muy adecuados, pues mejoran a menudo con el tiempo y se pueden detener en cualquier momento.

b) Un 50% usaron métodos lineales (regresión logística, regresión). Un 1/3 usaron alguno de estos métodos : Redes neuronales, knn o naive bayes.

c) En general se usó casi siempre el ensamblado, solo un 18% no lo usó. Algunos usaron métodos cíclicos de ensamblado, incorporando cada iteración el modelo que resulta mejor. Pero el término es confuso (pueden llamar ensamblado a un simple random forest).

e) Pocos usaron SVM. Quizá no por falta de eficacia, sino por su coste computacional.

## 4) Cómo tunearon parámetros

Paréntesis técnico : métodos de tuneado alternativos)

a) **Grid search** (rejilla). Método más básico y más utilizado. Inconvenientes: rigidez y mala escalabilidad.

b) **Random search** (búsqueda aleatoria). Si hay parámetros mucho más importantes que otros  puede ser preferido a grid search. Mejora con el tiempo .

c) **Bayesian Search Optimization** (BSO). Método iterativo que pretende ir Corrige los valores de los parámetros en la dirección de decrecimiento del error. Es en este Menor coste computacional. Mejora con el tiempo. Pero debe ser desarrollado para cada al algoritmo concreto.

**Random search en R**

```
fitControl <- trainControl(method = "repeatedcv",
                           number = 10,
                           repeats = 10,
                           classProbs = TRUE,
                           summaryFunction = twoClassSummary,
                           search = "random")

set.seed(825)
fit <- train(Class ~ ., data = training,
                  method = "gbm",
                  metric = "ROC",
                  tuneLength = 30,
                  trControl = fitControl)
```

**Bayesian Search Optimization en R**

Se puede usar el paquete MlBayesOpt.

Los concursantes usaron mucho BSO pero también otros métodos de tuneado. No está reflejado en la documentación.

## 4) Qué medidas de remuestreo utilizar para comparar algoritmos-modelos

Normalmente los concursantes usaron validación cruzada con k grupos.
Por el coste computacional, se usaba CV sin repetir.

(fin del ChaLearn Challenge)

# Estrategias y conceptos para tratar con grandes datos, tiempo y recursos limitados

## Hardware

Servidores online : MLASS (Machine Learning as a Service), como Amazon, Google, IBM, Spark,etc.

## Software

R no maneja demasiado bien la memoria

Algunos **trucos para reducir el tiempo de proceso en R** en machine learning son:

- Usar el paquete data.table siempre que se pueda para manejo y reestructuración de datos. Es enormemente eficiente.
- Reducir el archivo a solamente las variables (columnas) que se van a utilizar en los modelos, antes de hacer tuneados o evaluar modelos.
- En Rstudio, borrar los objetos del workspace que no se usen con la función rm(objeto).
- Para operaciones básicas con datos, mejor usar matrices que data frames (aunque a veces no se puede pues las matrices solo admiten o bien todas las columnas numéricas o todas character, y aquí se ve otra ventaja de construir dummies).
- No usar loops si no son necesarios
- A menudo se pueden usar las library parallel y doParallel en caret (ver Ejemplo Parallel más adelante).

Algunas especificaciones del funcionamiento de la memoria de R se pueden ver en:
http://adv-r.had.co.nz/memory.html

- En el sentido de la velocidad y potencia, Python es en general superior a R

- R es más universal y un lenguaje orientado a datos y estadística. más de 150.000 paquetes en CRAN. Comunidad de usuarios de R gigantesca.

- En Machine Learning, Python dispone de más potencia . Técnicas exigentes computacionalmente como Deep Learning son impracticables en R.

- Por último, muchos algoritmos son fácilmente paralelizables, como Random Forest. Si el software y hardware lo permite la paralelización hace más rápido el proceso. Igualmente muchos procesos de remuestreo y pruebas de tuneado también son paralelizables.

# Estrategias de partición y remuestreo

Ante archivos con NxM muy grande, se puede utilizar una muestra para el tuneado y selección de modelos.

Muestreo aleatorio simple

Muestreo aleatorio estratificado

# Ejemplo Parallel

```
library(parallel)
library(doParallel)

GS_T0 <- Sys.time()
cluster <- makeCluster(detectCores() - 1) # number of cores, convention to leave 1 core for OS
registerDoParallel(cluster) # register the parallel processing

gbmgrid<-expand.grid(shrinkage=c(0.1,0.01,0.001,0.0001),
 n.minobsinnode=c(10,20),
 n.trees=c(100,200,500,1000,2000,5000),
 interaction.depth=c(2))

set.seed(12345)
control<-trainControl(method = "CV",number=10,savePredictions = "all",classProbs=TRUE)

gbm<- train(factor(spam)~.,data=spam,tuneGrid=gbmgrid,
 method="gbm",trControl=control,distribution="bernoulli", bag.fraction=1,verbose=FALSE)


stopCluster(cluster) # shut down the cluster
registerDoSEQ(); #  force R to return to single threaded processing

GS_T1 <- Sys.time()
GS_T1-GS_T0

gbm
```

```
Time difference of 12.71587 mins
```

El tiempo de proceso es de 12.71 minutos, frente a 38 minutos que tardaba sin parallel.

# La función predict en R

En el proceso de tuneado, comparación y evaluación de modelos a menudo queremos observar qué tal funciona nuestro modelo tentativo sobre un conjunto test de datos externo.

Para aplicar nuestro modelo sobre datos externos y observar su performance es necesario realizar los siguientes pasos:

1) Crear el objeto-modelo con caret o con cualquier paquete de R de modelización predictiva. Con caret el trainControl es “none”.

2) Aplicar la función predict(modelos, datostest) que usa ese modelo sobre los datostest y se obtienen como resultado las predicciones.

```r
# ****************************************
# LA FUNCIÓN PREDICT EN R
# ****************************************

# ****************************************
# Ejemplo 1: variable dependiente binaria
# ****************************************

# Para el ejemplo, dividimos el archivo en una muestra train para
# construir el modelo y una test para aplicarlo sobre ella.
#
# En cualquer aplicación práctica, los datos test son otro data frame
# diferente del utilizado para construir el modelo.

load("spam.Rda")
sample_size = floor(0.3*nrow(spam))

# sample_size=1380

# Se crean los índices para train test
set.seed(12345)
indices = sample(seq_len(nrow(spam)),size = sample_size)

# Se crean los archivos train test
train =spam[indices,]
test =spam[-indices,]

# 1) Creamos un objeto modelo con caret, method = "none" en trainControl
# Los parámetros se fijan con los mejores obtenidos con el proceso
# de tuneado.

gbmgrid<-expand.grid(shrinkage=c(0.01),
 n.minobsinnode=c(10),
 n.trees=c(5000),
 interaction.depth=c(2))

set.seed(12345)
control<-trainControl(method = "none",savePredictions = "all",classProbs=TRUE)

gbm<- train(factor(spam)~.,data=train,tuneGrid=gbmgrid,
 method="gbm",trControl=control,
 distribution="bernoulli", bag.fraction=1,verbose=FALSE)

gbm
```

```r
# 2) Aplicamos el objeto modelo creado sobre datos test

# a) Con probabilidades es predicciones1 y con corte 0.5 predicciones2
# pero lo pone en un factor y es difícil de tratar

predicciones1<-predict(gbm,test,type = "prob")
predicciones2<-as.data.frame(predict(gbm,test,type="raw"))

# Lo mejor es crear un data frame con las probabilidades y la clase
# más probable según corte 0.5

library(dplyr)

prediccionestodas<-predict(gbm,test,type = "prob")%>%
mutate('pred'=names(.)[apply(., 1, which.max)])

prediccionestodas

# Se obtienen tres columnas, las dos primeras son las probabilidades predichas
# y la tercera la predicción con punto de corte 0.5
#       email         spam  pred
# 1    1.164679e-01 8.835321e-01  spam
# 2    2.292862e-04 9.997707e-01  spam
# 3    7.174668e-03 9.928253e-01  spam
# 4    7.174668e-03 9.928253e-01  spam

# 3) Medidas de performance sobre datos test
#
# Se utilizará la función cofusionMatrix de caret, para lo que hay
# que pasar a factor las columnas predichas y original,
# y la función roc del paquete pORC para calcular el auc

prediccionestodas$pred<-as.factor(prediccionestodas$pred)
test$spam<-as.factor(test$spam)

salconfu<-confusionMatrix(prediccionestodas$pred,test$spam)
salconfu

library(pROC)

curvaroc<-roc(response=test$spam,predictor=prediccionestodas$spam)
auc<-curvaroc$auc

plot(roc(response=test$spam,predictor=prediccionestodas$spam))
```

El siguiente ejemplo utiliza predict en el caso de **variable dependiente continua**. Se usa el archivo compress.

```r
# ****************************************
# Ejemplo 2: variable dependiente continua
# En este caso no se pone classProbs=TRUE en control
# ****************************************
load("compress.Rda")

sample_size = floor(0.3*nrow(compress))

# sample_size=1380

# Se crean los índices para train test
set.seed(12345)
indices = sample(seq_len(nrow(compress)),size = sample_size)

# Se crean los archivos train test
train =compress[indices,]
test =compress[-indices,]

# 1) Creamos un objeto modelo con caret, method = "none" en trainControl
# Los parámetros se fijan con los mejores obtenidos con el proceso
# de tuneado. En este caso no se pone classProbs=TRUE en control al ser la
# variable dependiente continua

gbmgrid<-expand.grid(shrinkage=c(0.1),
 n.minobsinnode=c(10),
 n.trees=c(20000),
 interaction.depth=c(2))

set.seed(12345)
control<-trainControl(method = "none",savePredictions = "all")

gbm<-train(cstrength~age+water+cement+blast,data=train,tuneGrid=gbmgrid,
 method="gbm",trControl=control)

# 2) Se obtienen las predicciones con predict, se pasa a data frame y
# se renombra la columna de predicciones

predicciones<-as.data.frame(predict(gbm,test))

colnames(predicciones)<-"predi"

# 3) Se calcula medida de performance

errorMSE<-mean((predicciones$predi-test$cstrength)^2)
```

# Uso de h2o

h2o es una plataforma de machine learning con librerías rápidas y facilidad de relación con lenguajes y otros servicios como R, python, Spark.

En R se puede acceder a h2o mediante el paquete h2o.

## Un par de detalles para trabajar con h2o en R:

1) En h2o hay que traducir el data frame a formato h2o como se ve en el script.

2) Es conveniente borrar del data frame anteriormente las columnas que no se van a usar. La variable dependiente es mejor dejarla en la columna final. Las input en las columnas restantes. Input e Output se nombran por separado en h2o como se verá en el script.

3) Los parámetros y documentación de cada algoritmo se pueden ver en la web de h2o.
http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science.html

Se realiza aquí un ejemplo de comparación de tiempos. Se compararán:

-Con caret CV de 10 grupos.
-Con h20 nthreads=1 CV de 10 grupos.
-Con h20 nthreads=8 CV de 10 grupos.

**Con caret CV de 10 grupos.**

```
library(caret)
start_time <- Sys.time()

gbmgrid<-expand.grid(shrinkage=c(0.1),
 n.minobsinnode=c(10),
 n.trees=c(2000),
 interaction.depth=c(2))

set.seed(12345)
control<-trainControl(method = "CV",number=10,savePredictions = "all",classProbs=TRUE)

gbm<- train(factor(spam)~.,data=spam,tuneGrid=gbmgrid,
 method="gbm",trControl=control,distribution="bernoulli", bag.fraction=1,verbose=FALSE)

gbm

# Control tiempo final

end_time <- Sys.time()
tiempo_total<-end_time - start_time

tiempo_total


# Time difference of 2.185471 mins
```

**Con h20 nthreads=1 CV de 10 grupos.**

```
library(h2o)

# Pongo un solo cluster para que los resultados sean reproducibles.
# Para más velocidad se puede poner 8, es mejor, y lo haremos después

h2o.init(nthreads=1)

# Para h2o Lo primero es traducir el archivo a h2o
spam$spam<-as.factor(spam$spam)
train<- as.h2o(spam)

# Después hay que verificar los números de variables en  las columnas
# y si es necesario reordenarlas. En spamm y=58, x=1:57

start_time <- Sys.time()

gbm <- h2o.gbm(x = 1:57,   y = 58,  training_frame = train,ntrees = 2000,learn_rate=0.1,min_rows = 10,nfolds=10)

gbm
end_time <- Sys.time()
tiempo_total<-end_time - start_time

tiempo_total

# Time difference of 6.228595 mins
```

**Con h20 nthreads=8 CV de 10 grupos.**
(Se omite el código aquí )

```
# Time difference of 1.597189 mins
```

Como se ve, es más rápido h2o que caret, en general. Se puede observar en el ordenador que el uso de CPU por el java implicado es total, con lo cual es casi imposible realizar otras operaciones mientras se ejecutan los procesos más complicados de h2o como el AutoML que se verá más adelante.

# AutoML en h2o

El paquete h2o contiene una utilidad de Auto Machine Learning ML.

Por defecto, prueba los algoritmos gbm, glm-logística (con regularización), random forest y redes neuronales (mediante el módulo deeplearning, puede usar varias capas y funciones de activación modernas).

También utiliza ensamblado (Stacked Ensemble) a partir de los algoritmos base y optimizando los pesos.

Desde el punto de vista práctico, es muy interesante pues puede permitirnos ganar tiempo en el proceso de selección de algoritmos y tuneado.

Tiene las opciones de:

a) decirle cuántos modelos puede probar (max_models=)

b) decirle por cuánto tiempo va a experimentar modelos (max_runtime_secs =)

```
# ***********************************
# Pruebas con AutoML
# ***********************************

start_time <- Sys.time()

aml <- h2o.automl(x = 1:57,y=58,training_frame = train,max_models = 20,seed = 1,keep_cross_validation_predictions=TRUE,nfolds=4)

lb <- aml@leaderboard
print(lb, n = nrow(lb))  # Print all rows instead of default (6 rows)

aml@leader

end_time <- Sys.time()
end_time - start_time
```

| | model_id | auc | logloss | mean_per_class_error | rmse | mse |
|---|---|---|---|---|---|---|
| 1 | GBM_grid_1_AutoML_20191031_123255_model_5 | 0.9899220 | 0.1166451 | 0.04125068 | 0.1780367 | 0.03169708 |
| 2 | GBM_grid_1_AutoML_20191031_120520_model_5 | 0.9899220 | 0.1166451 | 0.04125068 | 0.1780367 | 0.03169708 |
| 3 | StackedEnsemble_BestOfFamily_AutoML_20191031_123255 | 0.9892149 | 0.1260249 | 0.04258895 | 0.1826217 | 0.03335070 |
| 4 | StackedEnsemble_AllModels_AutoML_20191031_123255 | 0.9890311 | 0.1245808 | 0.04369052 | 0.1820434 | 0.03313980 |
| 5 | GBM_4_AutoML_20191031_123255 | 0.9887581 | 0.1253731 | 0.04657351 | 0.1860117 | 0.03460036 |
| 6 | GBM_4_AutoML_20191031_120520 | 0.9887581 | 0.1253731 | 0.04657351 | 0.1860117 | 0.03460036 |
| 7 | GBM_grid_1_AutoML_20191031_123255_model_1 | 0.9884785 | 0.1313290 | 0.04548866 | 0.1863740 | 0.03473526 |
| 8 | GBM_grid_1_AutoML_20191031_120520_model_1 | 0.9884785 | 0.1313290 | 0.04548866 | 0.1863740 | 0.03473526 |
| 9 | GBM_3_AutoML_20191031_123255 | 0.9884494 | 0.1270644 | 0.04654799 | 0.1873124 | 0.03508595 |
| 10 | GBM_3_AutoML_20191031_120520 | 0.9884494 | 0.1270644 | 0.04654799 | 0.1873124 | 0.03508595 |
| 11 | GBM_2_AutoML_20191031_120520 | 0.9883138 | 0.1267497 | 0.04614865 | 0.1866449 | 0.03483631 |
| 12 | GBM_2_AutoML_20191031_123255 | 0.9883138 | 0.1267497 | 0.04614865 | 0.1866449 | 0.03483631 |
| 13 | GBM_1_AutoML_20191031_120520 | 0.9880694 | 0.1287181 | 0.05080516 | 0.1886606 | 0.03559283 |
| 14 | GBM_1_AutoML_20191031_123255 | 0.9880694 | 0.1287181 | 0.05080516 | 0.1886606 | 0.03559283 |
| 15 | GBM_5_AutoML_20191031_123255 | 0.9868428 | 0.1362874 | 0.04882203 | 0.1922119 | 0.03694540 |
| 16 | GBM_5_AutoML_20191031_120520 | 0.9868428 | 0.1362874 | 0.04882203 | 0.1922119 | 0.03694540 |
| 17 | GBM_grid_1_AutoML_20191031_123255_model_3 | 0.9862433 | 0.1691614 | 0.05680628 | 0.2043406 | 0.04175508 |
| 18 | GBM_grid_1_AutoML_20191031_120520_model_3 | 0.9862433 | 0.1691614 | 0.05680628 | 0.2043406 | 0.04175508 |
| 19 | XRT_1_AutoML_20191031_120520 | 0.9844652 | 0.1917081 | 0.05299948 | 0.2065353 | 0.04265685 |
| 20 | XRT_1_AutoML_20191031_123255 | 0.9844652 | 0.1917081 | 0.05299948 | 0.2065353 | 0.04265685 |
| 21 | GBM_grid_1_AutoML_20191031_120520_model_2 | 0.9843333 | 0.1623932 | 0.05700233 | 0.2070002 | 0.04284909 |
| 22 | GBM_grid_1_AutoML_20191031_123255_model_2 | 0.9843333 | 0.1623932 | 0.05700233 | 0.2070002 | 0.04284909 |
| 23 | DRF_1_AutoML_20191031_120520 | 0.9838381 | 0.2024319 | 0.05476578 | 0.2085319 | 0.04348555 |
| 24 | DRF_1_AutoML_20191031_123255 | 0.9838381 | 0.2024319 | 0.05476578 | 0.2085319 | 0.04348555 |
| 25 | GBM_grid_1_AutoML_20191031_120520_model_4 | 0.9835526 | 0.1665152 | 0.05713943 | 0.2088287 | 0.04360942 |
| 26 | GBM_grid_1_AutoML_20191031_123255_model_4 | 0.9835526 | 0.1665152 | 0.05713943 | 0.2088287 | 0.04360942 |
| 27 | DeepLearning_grid_1_AutoML_20191031_123255_model_1 | 0.9712639 | 0.7182333 | 0.05468288 | 0.2189523 | 0.04794011 |
| 28 | GLM_grid_1_AutoML_20191031_123255_model_1 | 0.9707173 | 0.2339986 | 0.07331911 | 0.2468420 | 0.06093099 |
| 29 | GLM_grid_1_AutoML_20191031_120520_model_1 | 0.9707173 | 0.2339986 | 0.07331911 | 0.2468420 | 0.06093099 |
| 30 | DeepLearning_grid_1_AutoML_20191031_123255_model_3 | 0.9666055 | 0.9164175 | 0.05705496 | 0.2773550 | 0.07692581 |
| 31 | DeepLearning_grid_1_AutoML_20191031_123255_model_2 | 0.9589211 | 0.4164966 | 0.09895435 | 0.3000169 | 0.09001016 |

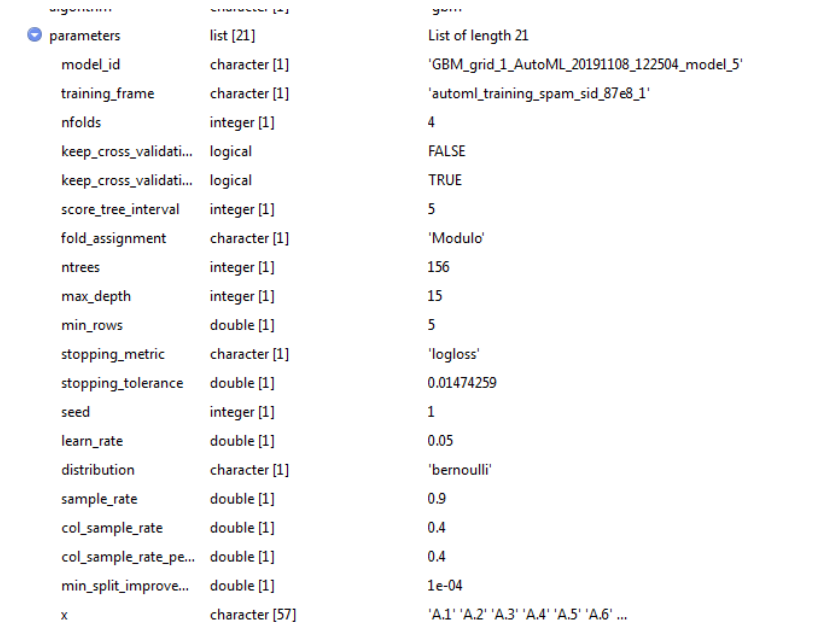
```
32                DeepLearning_1_AutoML_20191031_120520 0.9579238 0.3834131           0.09639185 0.3306654 0.10933960
33                DeepLearning_1_AutoML_20191031_123255 0.9557812 0.3183863           0.10052864 0.2953333 0.08722175
34  DeepLearning_grid_1_AutoML_20191031_120520_model_1 0.9461720 1.1339748           0.09166600 0.3031953 0.09192736
35  DeepLearning_grid_1_AutoML_20191031_120520_model_3 0.9232688 2.2476027           0.09781935 0.3939663 0.15520942
36  DeepLearning_grid_1_AutoML_20191031_120520_model_2 0.9155976 1.0080475           0.15737538 0.3979943 0.15839944
37  DeepLearning_grid_1_AutoML_20191031_123255_model_4 0.8180945 1.7805829           0.21677442 0.4928262 0.24287767
38  DeepLearning_grid_1_AutoML_20191031_123255_model_5 0.7893197 2.7688535           0.23069518 0.5037914 0.25380576
```

La tabla anterior presenta el resultado de aplicar AutoML con CV de 4 grupos. El mejor modelo es GBM, con un AUC de 0.9899 en validación cruzada. A continuación se muestran los detalles del mejor modelo:

```
Model Details:
==============

H2OBinomialModel: gbm
Model ID:  GBM_grid_1_AutoML_20191031_123255_model_5
Model Summary:
  number_of_trees number_of_internal_trees model_size_in_bytes min_depth max_depth mean_depth min_leaves max_leaves mean_leaves
1             156                      156              555922        15        15   15.00000        104        450   276.16025
```

## En Rstudio en leader+parameters se observan los parámetros del GBM creado:



| | | |
|---|---|---|
| parameters | list [21] | List of length 21 |
| model_id | character [1] | 'GBM_grid_1_AutoML_20191108_122504_model_5' |
| training_frame | character [1] | 'automl_training_spam_sid_87e8_1' |
| nfolds | integer [1] | 4 |
| keep_cross_validati... | logical | FALSE |
| keep_cross_validati... | logical | TRUE |
| score_tree_interval | integer [1] | 5 |
| fold_assignment | character [1] | 'Modulo' |
| ntrees | integer [1] | 156 |
| max_depth | integer [1] | 15 |
| min_rows | double [1] | 5 |
| stopping_metric | character [1] | 'logloss' |
| stopping_tolerance | double [1] | 0.01474259 |
| seed | integer [1] | 1 |
| learn_rate | double [1] | 0.05 |
| distribution | character [1] | 'bernoulli' |
| sample_rate | double [1] | 0.9 |
| col_sample_rate | double [1] | 0.4 |
| col_sample_rate_pe... | double [1] | 0.4 |
| min_split_improve... | double [1] | 1e-04 |
| x | character [57] | 'A.1' 'A.2' 'A.3' 'A.4' 'A.5' 'A.6' ... |

El GBM ganador usa 156 iteraciones, shrink(learning rate)=0.05, min_rows (observaciones en el último nodo)=5 , y usa al estilo de xgboost muestreo de 90% de observaciones (sample_rate), y sortea variables antes de cada árbol (col_sample_rate)=0.4.

La información de la tabla también se puede obtener en la consola con `aml@leader@parameters`.

Si se desea replicar el modelo se pueden apuntar los parámetros y modificarlos a gusto.
También se puede conservar como objeto-modelo el modelo leader , con getModel.

En el ejemplo siguiente se comprueban los errores en el modelo escogido gbm1, y se comparan por ejemplo con poner el col_sample_rate=1 y col_sample_rate_per_tree=1 (utilizar todas las variables en cada árbol, el gradient boosting normal).

```
aml@leader

# "GBM_grid_1_AutoML_20191108_122504_model_5"

modelo2 <- h2o.getModel("GBM_grid_1_AutoML_20191108_122504_model_5")

aml@leader@parameters

gbm1<- h2o.gbm(x = 1:57,   y = 58,  training_frame = train,
 ntrees = 156,learn_rate=0.05,min_rows = 5,nfolds=10,sample_rate=0.9,
 col_sample_rate=0.4,col_sample_rate_per_tree=0.4)

gbm1

gbm2<- h2o.gbm(x = 1:57,   y = 58,  training_frame = train,
 ntrees = 200,learn_rate=0.05,min_rows = 5,nfolds=10,sample_rate=0.9,
 col_sample_rate=1,col_sample_rate_per_tree=1)

gbm2
```

Se observa que las tabla de CV son parecidas para gbm1:

| | mean | sd | cv_1_valid | cv_2_valid | cv_3_valid |
|---|---|---|---|---|---|
| accuracy | 0.9582583 | 0.0081292745 | 0.960084 | 0.9655172 | 0.9497717 |
| auc | 0.9866073 | 0.0031040339 | 0.99278986 | 0.9901595 | 0.9848485 |

Y para gbm2:

| | mean | sd | cv_1_valid | cv_2_valid | cv_3_valid |
|---|---|---|---|---|---|
| accuracy | 0.95782185 | 0.005042103 | 0.9551569 | 0.9663677 | 0.9506438 |
| auc | 0.9870111 | 0.0032695893 | 0.9884001 | 0.9885125 | 0.97575295 |

El mejor modelo de AutoML vale como punto de partida, y al menos ya sabemos que podemos aspirar a un auc cercano al 0.986.