Comparación de algoritmos y estrategias avanzadas en Machine Learning

Contenido

No free lunch theorems

En el artículo *The Lack of A Priori Distinctions Between Learning Algorithms* (David H. Wolpert, 1996, Neural Computation, Vol. 8, nº7) se aborda el problema de encontrar un algoritmo universal que bata a los demás en algún criterio, en todo tipo de datos. Se concluye, como es intuitivo, que no existe ese algoritmo, y que, dependiendo de muchos factores, en cada caso particular hay algoritmos que funcionan mejor.

Se denominan "no free lunch theorems" a esos conceptos y conjeturas. No hay comida gratis, no hay un algoritmo panacea a pesar de Kaggle y el Xgboost. Hay que trabajarse los datos en cada caso, comparar algoritmos y tomar decisiones, que en muchos casos también tienen que tener en cuenta motivos prácticos, de coste o sentido común, y no solamente medidas objetivas como el AUC o la tasa de fallos.

Artículos de comparación de algoritmos

Artículo 1: 179 algoritmos, 121 datasets (2014)

En esta sección veremos resultados de dos artículos que realizan una comparación empírica de una batería de algoritmos sobre un conjunto extenso de datasets.

El primer artículo data de 2014:

Do we need hundreds of Classifiers to solve Real World Classifications Problems? (Delgado et al, 2014, Journal of Machine Learning Research (2014) 3133-3181).

En este artículo se enfrentan 179 algoritmos a 121 datasets en problemas de clasificación.

Es un artículo de acceso libre en internet para ser descargado:

http://jmlr.org/papers/v15/delgado14a.html

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

Tabla tomada del artículo *Do we need hundreds of Classifiers to solve Real World Classifications Problems?* (Delgado et al, 2014, Journal of Machine Learning Research (2014).

Como se ve en la tabla anterior, los datasets tratados tiene gran variedad. Archivos con 120 observaciones como ac-inflam, o con 130.064 como miniboone. Hay archivos con solo 2 variables input como congress-voting, o con 202 como arrhytmia. Hay problemas multiclase como chess-krvk o bien con clasificación binaria como blood. El porcentaje de la clase mayoritaria también varía.

Entre los algoritmos utilizados cabe destacar:

- Varias versiones de SVM
- Varias versiones de Random Forest
- Redes con avnnet y nnet
- Versión simple de regresión logística y con parámetros de regularización (glmnet).
- Gradient boosting no está representado en el paquete gbm ni en xgboost que no existía entonces, por lo cual los resultados no son muy actuales, al ser gbm un algoritmo-paquete muy competitivo. El siguiente artículo sí lo incorpora, reflejando la fuerza de ese algoritmo. Sí está C5.0_t, gradient boostin con árboles C5.0

Para comparar los algoritmos para cada dataset:

1) Se dividen los datos en 50% training y 50% test. Se tunean los parámetros de cada algoritmo escogiendo los valores de los parámetros que dan mejor performance sobre el conjunto test , construyendo los algoritmos con los datos train.

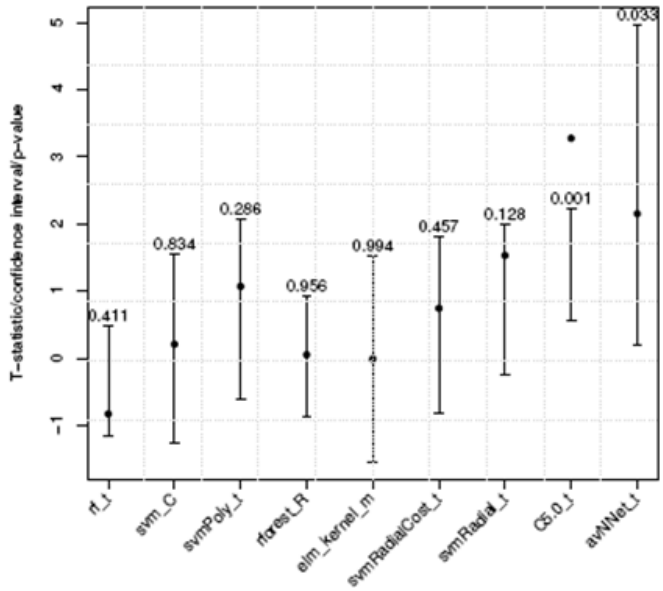
2) Una vez escogidos los mejores parámetros, se realiza validación cruzada de 4 grupos una sola vez y se obtiene la medida de performance con cada algoritmo.

Rank	Acc.	κ	Classifier	Rank	Acc.	κ	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)

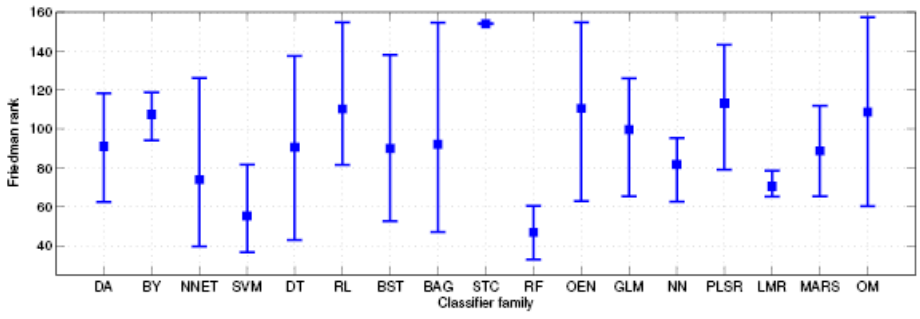
En cada dataset, hay un algoritmo ganador en cuanto a la medida de accuracy, (rango 1), un segundo (rango 2) , etc. Sobre todos los datasets so obtiene la media del rango que ha obtenido cada algoritmo. Son 179 algoritmos, y el mejor algoritmo (par_RF, una versión en paralelo de random forest) obtiene rango promedio 32.9. Con lo cual se verifica una vez más que no hay un algoritmo claramente dominador (en cuyo caso el mejor algoritmo tendría un rango promedio más bajo).

De la tabla se pueden sacar algunas conclusiones, que se matizarán más adelante:

- Los cinco primeros puestos son versiones de Random Forest y Support Vector Machines. En el puesto 10 aparece avnnet con rango promedio 44.1. Se observa que avnnet es mejor que nnet como ya se ha comentado.
- Una versión de la logística con parámetros de regularización , glmnet_R, aparece con rango medio 65 . La logística simple, que tiene la desventaja de no contar con parámetros a tunear, tiene rango promedio 67.



Las diferencias pueden no ser tan grandes. En el gráfico anterior se realizan contrastes de hipótesis sobre las diferencias en accuracy del algoritmo ganador paRF_t y los primeros algoritmos y se elabora un intervalo de confianza. Aquellos intervalos que contienen al cero y con p-valor alto (>0.1) son algoritmos cuya performance no difiere significativamente del algoritmo ganador paRF_t y podrían ser en cierto modo equivalentes a él. Concretamente versiones de rforest y de svm, aparte del algoritmo basado en redes elm (extreme learning machines).



La comparación en los términos anteriores puede no ser justa. Si se meten 50 versiones de random forest simplemente por azar puede quedar primero ese algoritmo. El gráfico anterior agrupa la performance por familias, junto con un intervalo de confianza basado en el test de Friedman de rangos (este test es algo muy utilizado para comparar algoritmos).

Se observa cómo las familias RF y SVM funcionan muy bien, y también se ve la escasa variabilidad de los rangos de la familia LMR (logística y logística multinomial), lo cual ofrece un buen desempeño. Como se ha dicho, los métodos boosting BST no presentan su mejor versión (no se usa el paquete gbm) y por ello no admiten una buena comparación. Igualmente las familias de redes dependen mucho de los paquetes utilizados, de ahí su alta variabilidad.

El artículo es de interés y muestra un esfuerzo por responder a la pregunta del título. Parece que ciertas familias funcionan mejor que otras, pero con probar la mejor versión de cada familia solo haría falta usar posiblemente las familias que se estudian en este módulo de machine learning (incluso dejar las redes solamente para otro tipo de problemas complejos tratados con deep learning).

Algunos comentarios :

1) El artículo compara problemas multiclase con clasificación binaria. Tiene una sola sección destinada a los resultados concretos en datasets de clasificación binaria.

2) Como se ha comentado, las versiones de gradient boosting no son las mejores

3) Poner muchas versiones de un mismo algoritmo puede enmascarar los resultados. A pesar de ello, se observa el buen desempeño comparativo de rf y svm.

4) Como se puede observar al final del artículo, el rango de valores de tuneado es muy escaso. Esto, que puede estar motivado por el esfuerzo computacional, puede favorecer a algunos algoritmos frente a otros.

5) Obviamente se utilizan unos paquetes concretos y para muchos de estos algoritmos existen muchos paquetes alternativos y versiones, que no se han probado y podrían dar lugar a otros resultados.

6) Por supuesto, nada indica que los datasets utilizados sean una muestra representativa de los datasets del mundo. Son datasets que han sido utilizados por su interés en anteriores artículos científicos y por lo tanto “por su interés” puede significar que tienen separabilidad no lineal, ser extraños o complejos y no ser representativos del conjunto de problemas reales.

Algunas de estas cuestiones (no todas) son mejoradas en el siguiente artículo. Este nuevo artículo es posterior al anterior (2016 frente a 2014) y se centra solo en datasets de clasificación binaria.

Artículo 2: 14 algoritmos, 115 datasets (2016)

Comparison of 14 different families of classification algorithms on 115 binary datasets. Wainer, Jacques. (2016).

Puede ser descargado en:

https://arxiv.org/abs/1606.00930

Trabaja sobre 115 datasets con 14 familias de algoritmos, escogiendo la mejor versión de cada familia.

El modo de tuneado y comparación para cada dataset es el siguiente:

Se divide el dataset en dos partes iguales; se tunean los hiperparámetros en cada parte mediante validación cruzada de 5 grupos y se aplican los mejores hiperparámetros en el modelo sobre la otra parte, obteniendo una medida de error en cada parte. Luego se promedian las medidas y ese es el error que se utiliza para la comparación con otros algoritmos.

Las 14 familias utilizadas son gbm, knn (vecino más próximo), lvq, nnet, elm, lvq (learning vector quantization, una especie de red neuronal), sda (análisis discriminante), nb (naive bayes) , glmnet (logística con regularización), rf, svm, svmpoly, svmrbf, rknn (bagging de knn), bst (boosting de clasificadores lineales, no de árboles).

Los parámetros de tuneado aparecen en la siguiente tabla. Llama la atención que no utiliza avnnet y en nnet el tuneado es escaso. También el tuneado del shrinkage en gbm se queda algo corto y debido a la fecha del artículo xgboost no es utilizado.

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 α, 8 values equally spaced between 0 and 1. Free hyperparameter λ, 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

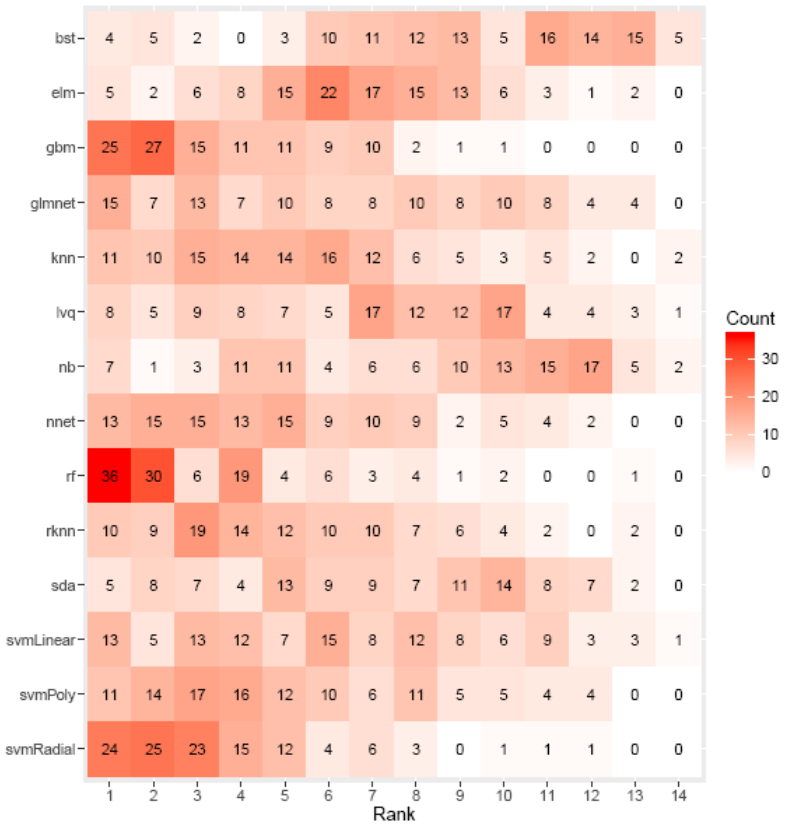
svmLinear . Hyperparameter: C $= 2^{-5}, 2^0, 2^5, 2^{10}, 2^{15}$.

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

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

En la tabla siguiente aparecen los resultados. Cada algoritmo es probado en los 115 datasets y se obtiene el número de veces que cae en cada rango. Por ejemplo, gbm aparece 25 veces en primer lugar, 27 en segunda, etc. Cada fila es la distribución de rangos de cada algoritmo. Una vez más se confirma el "no free lunch" pues aún los algoritmos menos eficientes quedan alguna vez como primeros en algún dataset.

Figure 1 displays the heat map of the rank distribution.



15 binary datasets.

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

Se observa que los algoritmos estrella son random forest, gradient boosting y svmRadial. Glmnet (logística con regularización) tiene un desempeño mediano, al igual que las redes (cuya versión y tuneado no es óptima).

Si comparamos si hay diferencias estadísticamente significativas entre las familias de algoritmos vemos que los tres mejores (svmradial, gbm, rf) no difieren en su performance. En estas comparaciones por pares hay otro grupo de desempeño formado por nnet, rknn, svmPoly, knn, svmLinear, glmnet.

	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

Recapitulación sobre tuneado y el intercambio Sesgo-Varianza

Se recuerdan mediante una tabla algunas nociones sobre el efecto que puede tener el tuneado de parámetros en cada algoritmo en términos de sesgo y varianza. Recordemos que si se utilizan técnicas de machine learning tipo caja negra (redes, rf, gbm, svm) en lugar de los métodos clásicos regresión y regresión logística, es para obtener un sesgo significativamente más bajo y que merezca la pena la pérdida de explicabilidad.

Por ello una estrategia sencilla será tunear inicialmente los parámetros de forma agresiva, para bajar el sesgo, y si se consigue una mejor clara en términos de error respecto a los métodos clásicos, ir perfilando el modelo para reducir la varianza y que no sobreajuste.

Tras la tabla de sesgo-varianza se muestra una tabla informal de lo que conviene tener en cuenta habitualmente al probar estos algoritmos dependiendo de la escala de nuestros datos en número de variables y de observaciones.

Tabla orientativa Sesgo-Varianza

		Quiero bajar el Sesgo	Quiero bajar la Varianza
General (1)	número de variables	+	-
	introducir interacciones entre variables, creación de dummies	+	-
	Complejidad del modelo	+	-
	tamaño muestral	+	+
Logística, regresión			
	p-valor (stepwise, backward, forward) (2)	+	-
	categorías poco representadas	+=	-
	no linealidad, separabilidad no lineal (3)	-	-
Redes			
	número de nodos	+	-
	linealidad (3)	-	-
	número de iteraciones en algoritmo de optimización (usar early stopping puede reducir la varianza)	+	-
	probar diferentes algoritmos, funciones de activación (4)	+	-
	en bprop: learn rate,momentum (en general si subimos momentum bajamos learn rate). learn rate bajos requieren nº de iteraciones más altas	-	+
	Ensamblado de redes (5)	=	+
Árboles	linealidad (6)	-	-
	maxbranch	+	-
	nleaves(número de hojas)	+	-
	maxdepth(profundidad, relacionado con el número de hojas)	+	-
	leafsize (número de observaciones mínimo para un nodo)	-	+
	p-valor (si está contemplado)	+	-
Bagging árboles			

	porcentaje de muestras (7)	+	-
	iteraciones	+=	+
Random forest			
	porcentaje de muestras	+	-
	iteraciones	-	+
	porcentaje de variables a sortear en cada nodo	+=	-
Gradient Boosting		Quiero Bajar el Sesgo	Quiero Bajar la Varianza
	párametro de regularización (shrink) (8)	-	+
	iteraciones	+	-
	proporción muestreo stochastic gradient boosting (7)	+	-
SVM	C	+	-
	gamma cuando kernel RBF	+	-
	grado cuando kernel polinomial	+	-

Comentarios

La tabla es de carácter general. Sirve para **orientarnos** en la búsqueda de buenos modelos. Pero no significa que tomando esas acciones se reduzca forzosamente el sesgo o la varianza, sino que si se quieren reducir éstos, son acciones que se pueden probar con buenas posibilidades.

Por ejemplo, en la tabla pone que se puede aumentar el número de variables en el modelo para reducir el sesgo. Sí, es algo que se puede probar, pero esto no significa que introduciendo una variable que no tiene ninguna relación con la variable dependiente vaya a reducir el sesgo: probablemente aumente ambos, sesgo y varianza.

Del mismo modo se razona en el resto de items: aumentar el número de nodos en la red puede ser una estrategia para reducir el sesgo, pero a menudo tomar un número exagerado de nodos puede también aumentar el sesgo.

1) En general, hacer el modelo más complejo suele reducir el sesgo y aumentar la varianza. Esto incluye aumentar el número de variables, utilizar variables categóricas con muchas categorías, incluir interacciones, utilizar modelos no lineales (por ejemplo regresión polinómica), etc.

Aumentar el tamaño muestral tiene incidencia sobre el sesgo y la varianza. Ambas cantidades disminuyen aumentando n.

2) El p-valor de corte en procesos de selección de variables mide la exigencia en la inclusión de variables o efectos en el modelo. A menor p-valor, más exigentes somos en la inclusión de variables; reducir el p-valor implica entonces menos variables y por lo tanto más sesgo pero menos varianza. Aumentar el p-valor permite la inclusión de más variables y por lo tanto modelos más complejos que pueden tener menos sesgo pero más varianza.

3) La linealidad en modelos de regresión o separabilidad lineal en modelos de clasificación mejora en general ambos sesgo y varianza. Si se puede conseguir esta linealidad con transformaciones sencillas y utilizar regresión o regresión logística (en clasificación) puede ser mejor que probar algoritmos más complicados.

4) Monitorizar modelos con excesivo número de aspectos a probar (activación, algoritmo, etc.) y centrándonos en el resultado que se tiene en datos test puede llevar a un sobreajuste implícito (mareamos demasiado nuestros datos).

5) En general las técnicas de ensamblado reducen la varianza y a veces el sesgo. Si no aportan demasiado en ninguno de los dos sentidos es mejor no utilizarlas.

6) La linealidad afecta negativamente a los árboles. En estos casos mejor utilizar otras técnicas.

7) En Bagging y Random Forest se utiliza el muestreo para evitar sobreajustes y ganar en capacidad de generalización. Porcentajes altos (1,0.90 ,etc.) reducen esta capacidad (aumentan varianza) a veces para reducir el sesgo. Porcentajes demasiado bajos (0.30) en general no compensan pues aumentan demasiado el sesgo con poca reducción en varianza. Lo habitual es el rango 0.7-1 (con reemplazo).

8) El parámetro de regularización en gradient boosting se usa para ir mejorando las predicciones en cada iteración. Un valor demasiado alto (0.3) hará que las predicciones den bandazos y no converjan a una solución fina. Un valor demasiado bajo (0.0001) necesitará muchas iteraciones para converger y a menudo se puede quedar en mínimos locales dependiendo del problema. Valores habituales están en torno a (0.01-0.2).

Dos esquemas básicos de construcción y comparación de modelos

Hasta ahora se ha planteado un esquema básico de construcción y comparación de modelos con la intención de abordar los conceptos primarios del machine learning:

Esquema [1] planteado en este módulo de machine learning

1) Se ha planteado un modo sencillo de abordar la selección de variables mediante métodos stepwise, intentando evitar el sobreajuste y la dependencia de los datos training a través del remuestreo (stepwise repetido en submuestras).

2) El tuneado de parámetros en cada algoritmo se ha realizado con validación cruzada y caret, a través de grids (rejillas de valores).

3) La comparación de algoritmos se ha realizado mediante validación cruzada repetida, insistiendo en observar gráficamente los efectos sesgo-varianza de los algoritmos-modelos utilizados.

Este esquema es básico y está orientado a la comprensión del funcionamiento de estos algoritmos y del intercambio sesgo-varianza, pero tiene la desventaja de ser demasiado exignte computacionalmente. Se puede hacer más básico para solventar los problemas de excesivo tiempo computacional en grandes matrices de datos:

Esquema [2] rápido

1) Se seleccionarían variables a través de los gráficos de importancia obtenidos de un modelo tuneado de gbm y/o random forest. Se aplicaría un punto de corte sobre todo para eliminar variables irrelevantes, o simplemente no se realizaría selección, o se seleccionarían las variables con importancia>0 (que pueden ser muchas).

2) El tuneado de parámetros se haría vía CV o training-test si los datos son grandes, y se seleccionaría directamente el algoritmo-modelo con menor error obtenido en los sucesivos tuneados.

Este modo de proceder tiene la ventaja de velocidad computacional, y la selección de variables permite descartar las malas y si después se utilizan algoritmos basados en árboles estos ya tienen selección incluida.

También tiene sus desventajas:

1) La selección de variables mediante importancia en árboles puede tomar varias variables que son predictivas pero entre ellas tienen casi la misma información. Estas variables aparecerán todas en la lista cuando, alternativamente, un método wrapper (stepwise) solo usaría posiblemente una o algunas de ellas, reduciendo errores y sobreajustes. Además 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.

2) La comparación de algoritmos solo tendría en cuenta el sesgo, y además si no se hace mucho remuestreo, al repetir los resultados con diferentes semillas podrían escogerse diferentes algoritmos cada vez, resultando en una dependencia del azar no deseable en la selección de modelos.

Por otro lado, el esquema 1 planteado también tiene limitaciones, sobre todo de proceso computacional. Se verá cómo son los planteamientos más modernos para obtener vía automática un modelo predictivo de machine learning que sea robusto (aunque siempre de "caja negra", no explicativo por supuesto).

El ChaLearn AutoML Challenge

Documentos challenge.pdf y appendix.pdf

La serie de competiciones ChaLearn AutoML Challenge entre 2015 y 2018 consistió en una serie de rondas de predicción automática sobre varios datasets. Aunque el esquema variaba algo con los años, en general se conservaban las siguientes características:

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

2) El concurso estaba limitado a unas especificaciones de hardware, potencia y memoria, y el tiempo de proceso estaba limitado dependiendo del dataset (20 minutos aproximadamente). Aunque el código no estaba restringido un 82% de los concursantes usaron Python por la velocidad de proceso y que se les ofrecía la librería de uso scikit-learn y script para evaluación de métricas. También se usaron Java, C++ y R, solos o en combinación con Python.

3) Los datasets no eran conocidos por los participantes. Hay mucha variedad; regresión, clasificación binaria y multiclase; en alguna edición solamente clasificación binaria. La evaluación era sobre datos test y las medidas de diagnóstico podían ser muy variadas.

El punto 2) es particularmente importante pues obliga a los concursantes a establecer métodos eficientes. En la práctica siempre se van a tener limitaciones de tiempo y recursos y las estrategias utilizadas en el concurso pueden servir como ideas base para aplicaciones prácticas.

Varios son los 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

Veremos en cada una de estas facetas cuales eran las estrategias usadas habitualmente por los concursantes.

Esta información está en las página 33-37 del documento challenge.pdf y 24-26 del appendix.pdf.

1) Qué medidas de preprocesado automático usar

Los concursantes abordaron este problema generalmente así:

a) No se profundiza demasiado en el preprocesado, utilizando lo básico del kit ofrecido en el paquete python. El tratamiento de missings consiste básicamente en imputación por la mediana en continuas o variables indicadoras de missing en discretas y continuas.

b) Se utiliza normalización simple de variables continuas y codificación dummy para categóricas.

Tengamos en cuenta que esta manera de proceder viene inducida por el tipo de concurso, automático y limitado en el tiempo. Como contrapartida, por ejemplo en Kaggle donde solo se trata de un archivo cada vez, se conoce el contexto y no hay limitaciones a este respecto, se usa mucho preprocesado y feature engineering.

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

Aquí cabe destacar:

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 (embedded) una cierta 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.

No olvidemos que se trata de un concurso automático, donde no hay interpretación ni conocimiento del contexto, y al ser limitado en tiempo no se pueden usar métodos exhaustivos de selección, sobre todo en archivos grandes.

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) Al menos 2/3 usaron alguna forma de regularización (en las regresiones por ejemplo, pero el término es confuso pues en gradient boosting hay una constante de regularización obligatoria).

d) 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.

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

4) Cómo tunear parámetros

En este módulo de Machine Learning se ha visto el método más básico, que es grid search. Sin embargo, existen métodos que en ocasiones son más eficientes en términos de tiempo de

cómputo. Comentamos aquí los tres métodos más habituales, para los que existen paquetes en R y python:

a) **Grid search** (rejilla). Es el método más básico y más utilizado. Sus inconvenientes son la rigidez (es necesario prefijar los valores de búsqueda), y la escalabilidad (cuantos más parámetros el espacio de búsqueda aumenta exponencialmente).

b) **Random search** (búsqueda aleatoria). Se busca en el espacio de los parámetros sorteando sus valores. Cuando hay parámetros mucho más importantes que otros puede ser preferido a grid search. Otra ventaja es que mejora con el tiempo y se puede detener en cualquier momento, lo cual es muy práctico a la hora de controlar el tiempo computacional.

c) **Bayesian Search Optimization** (BSO). Los métodos anteriores son métodos de fuerza bruta, se prueban de manera independiente los parámetros y se observa su performance. BSO es un método iterativo que pretende ir corrigiendo los valores de los parámetros en la dirección de decrecimiento del error. Es en este sentido un método con menor coste computacional que los anteriores, y por ello es muy utilizado en problemas con un gran número de parámetros como los modelos de Deep Learning, donde suele ser impracticables los métodos anteriores.

Su principal deventaja es que hay que implementarlo basándose en cada algoritmo concreto, elaborar una función de error, etc. Aunque es una teoría no demasiado complicada, a veces la programación o planteamiento es diferente según los paquetes utilizados.

Hay que remarcar que unos métodos de tuneado pueden tener mejor performance dependiendo del algoritmo a tunear, del número de parámetros y de las características de los datos.

Random search en R

Se puede usar caret para ello. Simplemente en trainControl se pone search="random", y en la función train, no se pone tuneGrid, sino tuneLength=M, donde M es el número de evaluaciones de combinaciones de parámetros a probar. Por ejemplo:

```
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.

https://cran.r-project.org/web/packages/MlBayesOpt/vignettes/MlBayesOpt.html

Como BSO depende del modelo usado, hay que estar seguro de que el paquete en cuestión puede aplicar BSO sobre ese método. El paquete MlBayesOpt permite aplicar BSO sobre SVM, Random Forest y xgboost.

Finalmente, los documentos no reflejan de manera demasiado concreta el método utilizado de optimización de parámetros por los concursantes, solo algunos remarcan el uso de Bayesian Optimization, y módulos que de una manera u otra lo aplican, como SMAC o Auto-Weka.

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

a) Normalmente los concursantes usaron validación cruzada con k grupos. En algunos momentos de la optimización o tuneado, algunos solo evaluaron en un grupo (lo que equivale a training-test) para acelerar el proceso sobre todo para descartar algoritmos o espacios de parámetros poco prometedores.

b) Por el coste computacional, se usaba CV sin repetir. También usaron el criterio "leaderboard", que es mirar qué tal se daban las predicciones sobre los datos test ofrecidos en el concurso, pues a modo de Kaggle se ofrecía mostrar el error cometido sobre esos datos test.

Aquí se puede hacer un paréntesis sobre el tuneado y evaluación y comparación de modelos: La teoría estricta afirma que al tunear sobre los mismos datos sobre los que se van a comparar los modelos se puede caer en sobreajuste. Lo ideal sería tunear en datos aparte (con CV por ejemplo), y probar el resultado en datos test. El error de cada modelo estaría mejor estimado así.

Por razones de coste computacional y de falta de disponibilidad de suficientes datos, en la práctica se realiza el Esquema [2] rápido mencionado arriba, tuneando y evaluando en los mismos datos con CV, sin repetir con ordenación aleatoria con diferentes semillas ni evaluar sesgo varianza.

Pero por las razones teóricas de sobreajuste siempre se intenta compensar y paliar el sobreajuste implícito en selección de modelos con otras medidas de corrección de varianza (regularización, preferir modelos sencillos frente a complicados ante la duda, ensamblado, etc.).

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

El esquema [1] planteado en este módulo es consistente, pero exhaustivo y exige demasiado computacionalmente.

Esquemas más básicos quizá permitan llegar a buenos modelos con un coste computacional menor y esto puede merecer la pena o no (perder precisión o fiabilidad a cambio de tiempo). Hemos visto como abordaban este problema los concursantes del ChaLearn AutoML Challenge.

En esta sección plantearemos algunos esquemas de estrategia y los compararemos.

Hardware

Obviamente lo primero es disponer del hardware más potente posible (es lo que hacen los de Kaggle en primer lugar). Las pruebas de tuneado, codificación, remuestreo etc. se benefician mucho de más potencia. Algoritmos más exigentes se pueden utilizar con cierto hardware y son casi inviables con menos potencia. Tanto la velocidad de proceso como la memoria RAM son importantes (casi más la RAM).

Existen por lo tanto plataformas online que ofrecen MLASS (Machine Learning as a Service), como Amazon, Google, IBM, etc. para ofrecer con coste el uso de ordenadores potentes por el tiempo pagado. Hay plataformas con Spark que ofrecen la posibilidad de ser más eficientes en abordar el problema de grandes datos, siempre contando con buenos servidores por supuesto.

Software

Hay software más adecuado que otro para aprovechar el hardware en machine learning. R desgraciadamente no maneja demasiado bien la memoria, todos los datos deben estar en la RAM.

Algunas especificaciones del funcionamiento de la memoria de R se pueden ver en:

http://adv-r.had.co.nz/memory.html

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).

En el sentido de la velocidad y potencia, Python es en general superior a R por manejar mejor la memoria. R tiene la ventaja por otro lado de ser más universal y un lenguaje orientado a datos y estadística fácil de abordar, por lo que existen en CRAN más de 150.000 paquetes y la comunidad de usuarios de R es gigantesca, abordando todo tipo de disciplinas. En visualización R también es superior a Python.

Si nos restringimos a Machine Learning, Python dispone de más potencia (a cambio de menos variedad de paquetes), y por ejemplo 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

Si se dispone de un archivo de datos grande (NxM grande, donde N es el número de observaciones y M de variables con las categóricas en dummies) es posible trabajar con una muestra para el tuneado y comparación básica, y ante ciertas dudas solo probar al final los mejores modelos, restringidos al mínimo, sobre todos los datos.

A menudo hay que tener en cuenta que no es necesario utilizar siempre **todos** los datos. Por ejemplo, si hay 1.000.000 observaciones y 20 variables es muy posible que un modelo funcione bien aunque esté construido con 50.000 observaciones. En este sentido es importante el valor de N/nvar, que es un número que permite predecir bien la performance (ver documento Challenge.pdf). Por ejemplo, 50.000/20 son 2500 observaciones por variable (las variables categóricas supuestamente divididas en dummies), y a partir de 500 o 1000 observaciones por variable el algoritmo puede que funcione de manera similar en una muestra training moderada que en la muestra completa.

La selección de la muestra puede ser por muestreo aleatorio simple, o por ejemplo estratificada en la variable dependiente si esta es categórica, para conservar las proporciones poblacionales.

Muestreo aleatorio simple

Se extrae una muestra sin reemplazamiento de los datos, asignando la misma probabilidad de aparición a cada observación. Previamente se decide el tamaño o fracción muestral.

```
load("saheartbis.Rda")

# Muestreo aleatorio simple
# Para extraer muestra aleatoria simple de 100 observaciones

library(dplyr)

muestra<-sample_n(saheartbis,size=100)
```

Muestreo aleatorio estratificado

Se extrae una muestra sin reemplazamiento de los datos, pero conservando en la muestra la misma proporción de categorías de la variable dependiente que en la muestra grande original (suponemos aquí que la variable dependiente es binaria).

Este método nos protege cuando la variable dependiente está muy desequilibrada, así nos garantizamos tener suficientes valores de la categoría más pequeña. Hay otros métodos como undersampling: por ejemplo se pueden tomar todas las observaciones de esa categoría pequeña, que suele ser la de interés y un cierto número de la categoría grande hasta completar un tamaño muestral manejable.

```
# Muestreo aleatorio estratificado por la variable dependiente
# table(saheartbis$chd)
# prop.table(table(saheartbis$chd))
# Se ve que hay un 65% de No y un 35% de Yes

# Al estratificar mantenemos esos % en la muestra obtenida
```

```
# Tomamos sample_frac(0.25) (1/4 de los 462 para tener aproximadamente
100 observaciones finales)

muestra2<- saheartbis %>%
  group_by(chd) %>%
  sample_frac(0.25)


table(muestra2$chd)
prop.table(table(muestra2$chd))
```

Control del tiempo de proceso en R

Para ver cuánto se tarda en un proceso, sencillamente se guarda el tiempo actual en una variable, se guarda el tiempo al finalizar el proceso y se resta. Para ello se usa la variable de sistema Sys.time() que mide el tiempo actual.

En el ejemplo siguiente se compara cuanto tiempo se tarda en realizar el mismo tuneado sobre una fracción del archivo spam, y sobre todo el archivo. Para ello se divide primero el archivo en train-test y se tunea sobre train.

```
# ****************************************
# EJEMPLOS CONTROL DEL TIEMPO EN R
# ****************************************

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



# Ejemplo spam CV de 10 grupos con 30% observaciones training y
tuneado de una gran rejilla

# Después comparamos con hacerlo con todas las observaciones

# ****************************************
# DIVISIÓN SIMPLE DE UN DATA FRAME EN TRAIN TEST
# ****************************************

# Primero obtenemos una muestra de 30% de observaciones, dividiendo en
train test el archivo

# Se calcula cuantas observaciones son el 30% en el archivo spam
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,]
```

```
# Como hemos dicho vamos a trabajar solo con los train

# Control tiempo inicio
start_time <- Sys.time()

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=train,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 12.04829 mins
```

Con el archivo train, de un 30% de observaciones , se tardan 12.04 en hacer el tuneado.

Después se realiza el mismo proceso con todo el archivo spam (se omite el código) y se tardan 38 minutos:

```
# Time difference of 38.96596 mins
```

En ambos casos la solución de los mejores parámetros es diferente, pero se observa que si tomamos la solución construida con la muestra del 30%, n.trees=5000, shrinkage=0.01 y n.minobsinnode=10 , al aplicar esta solución en validación cruzada sobre todos los datos (la segunda tabla), la accuracy es de 0.9537, muy cercana al óptimo tuneando con todos los datos, que da 0.9569.

Por lo tanto la solución con un 30% de los datos es suficientemente buena, tardando 12 minutos, frente a la solución con todos los datos, que tarda 38 minutos para una mejora en la accuracy de 0.003.

Ejemplo Parallel

Utilizando todos los datos de spam, se usan los máximos cores que se puede en el ordenador para realizar las pruebas del grid de parámetros en paralelo.

Para tuneado de gbm y muchos otros paquetes se puede aprovechar esta característica.

```
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.

El resultado de aplicar predict puede variar según el paquete o programa utilizado. En clasificación binaria, con la opción type=”prob” en predict se obtienen dos columnas de probabilidades predichas, una por cada clase. con la opción type=”raw” se obtiene una columna con la predicción de clase usando el punto de corte 0.5.

3) Calcular medidas de performance, si en los datos test se dispone de la variable dependiente. Si no se dispone de ella, no se podrá, las predicciones obtenidas con predict son el único resultado, que es el objetivo práctico de todo modelo predictivo.

En el ejemplo siguiente sobre **variable dependiente binaria** se usa un truco para tener en el mismo archivo de salida las predicciones por probabilidad y la predicción de clase. Par calcular

las medidas de performance sobre los datos test se usan la función confusionMatrix del paquete caret y la función roc del paquete pROC.

```
# ****************************************
# 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

# 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.

```
# ****************************************
# 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 h20 mediante el paquete h2o.

En la instalación es necesaria la última versión de java; el instalador en R la solicita.

h20 tiene los algoritmos más conocidos. Su paquete de redes, deeplearning, ofrece más parámetros que los habituales, destacando el uso de funciones de activación más modernas y eficientes como ReLu, maxout, etc.. Aunque no trabaja con verdaderas redes deeplearning como CNN .

h2o permite utilizar varios cores del ordenador mediante la opción nthreads. Con nthreads=1 es más lento pero los resultados en ese caso son reproducibles.

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 depediente 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 AutoML. 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 (Stecked 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. Sobre todo porque tiene las opciones de:

a) decirle cuántos modelos puede probar (max_models=)

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

Además utilizar AutoML con validación cruzada nos permite observar el nivel de error al que podemos aspirar. Los modelos resultantes se pueden retocar y mejorar, y pueden ser un punto de partida.

```
# ***********************************
# 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
```

Lo mejor sin embargo es obtener la información concreta a partir del objeto AutoML creado, que hemos llamado aml. En éste objeto, 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.