

XGBoost y sus amigos



Introducción

Introducción

Tanto **XGBoost**, como **CatBoost** y **LightGBM** han dominado buena parte de la escena en cuanto a los problemas de machine learning estructurados, es decir, con información tabular.

Basta con ver el impacto de los repositorios de estas tres librerías para ver su importancia en el ámbito:

- **XGBoost:**

 Star	21.1k	 Fork	8k
--	-------	--	----

- **LightGBM:**

 Star	12.6k	 Fork	3.3k
--	-------	--	------

- **CatBoost:**

 Star	5.9k	 Fork	898
--	------	--	-----

XGBOOST

XGBoost

El primer release de XGBoost nace en marzo de 2014 y se empieza a emplear en competencias de Kaggle en 2015, convirtiéndose en uno de los algoritmos más populares de la competencia.

Este algoritmo es una reimplementación de Gradient Boosting, con importantes mejoras, algunas de las cuales veremos en esta presentación.

Fue diseñado por Tianqi Chen, de la Universidad de Washington, y rápidamente recibió numerosos colaboradores a su proyecto.

Para más detalles les recomendamos consultar el [paper original](#), el [repositorio](#) y esta [lista](#) con charlas, proyectos, competencias, etc.

XGBoost

XGBoost permite definir como parámetro la función objetivo, de este modo admite una gran cantidad de variantes (véase este [link](#)):

- Regresión.
- Clasificación binaria.
- Clasificación múltiple.
- Ranking.
- Análisis de supervivencia.
- Conteo (modela el output con una regresión Poisson).
- Otro tipo de regresiones como Gamma.

XGBoost

XGBoost permite paralelizar el cómputo de manera eficiente. Además, provee soporte para GPU y para sistemas distribuidos como Kubernetes, Dask o Spark.

Dos de los motivos por los que se impuso XGBoost a los algoritmos previamente existentes (gradient boosting de Sklearn, la librería gbm de R o la implementación de Spark) son su mayor velocidad (por la paralelización que realiza) y un mejor uso de la memoria.

Esto permite, desde luego, escalar a datasets más grandes y realizar una prueba de hiperparámetros más exhaustiva.

XGBoost - Mejoras

Algunas de las mejoras desarrolladas en el algoritmo son las siguientes:

1. Función objetivo regularizada
2. Elección del split
3. Shrinkage
4. Column subsampling
5. Algoritmos de split
6. Manejo de nulos

Función objetivo

Un boosting de árboles puede representarse matemáticamente como una sumatoria de funciones “f”. Cada una de estas funciones es un árbol que mapea una observación “x” con un score. Con lo cual, la suma de todos los scores nos dará la predicción final.

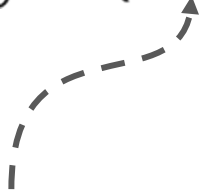
$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), f_k \in F$$

Cada “f” es un CART y “F” es el espacio de todos los posibles CART. “K” es el número total de árboles.


Función objetivo

La función objetivo a optimizar es, en términos generales:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$



Loss o función
de pérdida



Término de
regularización

Entrenamiento aditivo

Encontrar los K árboles simultáneamente es extremadamente complejo de tratar, por este motivo el modelo se entrena de manera aditiva, es decir, un árbol por vez.

$$\hat{y}_i^{(0)} = 0$$

$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$$

$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)$$

...

$$\hat{y}_i^{(t)} = \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

Entrenamiento aditivo

Entonces, ¿cómo encontrar el árbol en cada iteración?
Supongamos que usamos como pérdida el ECM. En ese caso, si reemplazamos nuestro objetivo en la iteración t se vuelve:

$$\text{obj}^{(t)} = \sum_{i=1}^n \left(y_i - \left(\hat{y}_i^{(t-1)} + f_t(x_i) \right) \right)^2 + \sum_{i=1}^t \Omega(f_i)$$

Función objetivo personalizada

Cuando empleamos el ECM podemos manipularlo de manera sencilla, para funciones más complejas se aproxima tomando la expansión de Taylor hasta el segundo orden.

De esta manera, en XGBoost podemos usar una gran variedad de funciones objetivo e, incluso, funciones objetivo personalizadas.

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

Función objetivo personalizada

Luego de hacer la expansión se llega a la siguiente forma general:

$$\sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

Donde:

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right)$$

$$h_i = \partial_{\hat{y}_i^{(t-1)}}^2 l\left(y_i, \hat{y}_i^{(t-1)}\right)$$

Término de regularización

Sea w un vector que asigna un score a cada hoja del árbol y sea q una función que asigna una hoja del árbol a una observación, podemos definir un árbol como:

$$F = \{f(x) = w_q(x)\}$$

Dicho esto volvamos a nuestra función objetivo original.

Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

$$\text{con } \Omega(f) = \boxed{\gamma T} + \boxed{\frac{1}{2} \lambda ||w||^2}$$

Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

con $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$

T es la cantidad
de hojas



Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

con $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$

Entonces gamma es cuánto debe reducir la función de pérdida un nuevo split, como mínimo.

Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

con $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$

Cuanto mayor es gamma más difícil es hacer un nuevo split, resultando un mayor gamma en árboles más simples.

Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

con $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$

Esta es la norma 2 o L2 (también llamada Ridge). También podríamos regularizar con norma L1 (Lasso, que es proporcional al módulo de los pesos) o una mezcla de ambas (ElasticNet).

Término de regularización

En XGBoost la regularización se plantea con los siguientes dos términos:

$$\mathcal{L} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

con $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$

En este caso, penaliza los scores muy grandes, favoreciendo que ninguna hoja sea especialmente importante

Split

Una vez que definimos nuestra loss y nuestro término de regularización podemos obtener el valor de los scores w que minimizan nuestra función objetivo.

Luego, podemos formular un valor de **Gain** como la diferencia entre el score de la rama izquierda y el score de la rama derecha, y el score original, tomando en cuenta la regularización, de la misma manera que en otros árboles. A diferencia de éstos, en este caso se toma la regularización como parte del split.

Puede verse el desarrollo formal de esto [acá](#).

Métodos de árboles

XGBoost implementa diferentes métodos para encontrar los splits, todos se controlan con el parámetro `tree_method`:

- **exact**: encuentra el split probando todas los posibles splits, es el algoritmo más lento.

Además, existen métodos aproximados, en los cuales en vez de samplear todos los posibles puntos de split, los valores de las variables se agrupan en "buckets" o "bins" y se precomputan las estadísticas del mismo.

Métodos de árboles

Luego se prueban sólo esos puntos de corte, acelerando el cómputo. En XGBoost se implementan:

- **approx**: los buckets se recalculan al comienzo de cada nuevo árbol y se usa como peso el hessiano. Pueden ver más detalles en el paper.
- **hist**: esta implementación surge con el modelo **LightGBM**, difiere en que las estadísticas se calculan una sola vez, al comienzo del entrenamiento, y se usa como peso el input provisto por el usuario. Es el algoritmo más rápido.
- **gpu_hist**: equivalente a hist pero soporta GPUs.

Mejoras adicionales

XGBoost implementa una serie de mejoras con respecto a algoritmos previos:

- **Shrinkage (eta)**: básicamente este parámetro lo que hace es quitar peso a cada nuevo árbol que se agrega al ensamble, multiplicando su influencia por una constante.
- **Sampling de columnas**: de la misma forma que Random Forest, esta librería implementa el sampling de columnas para reducir el overfitting e incrementar la variabilidad de los árboles. `colsample_bytree` y `colsample_bylevel`.

Mejoras adicionales

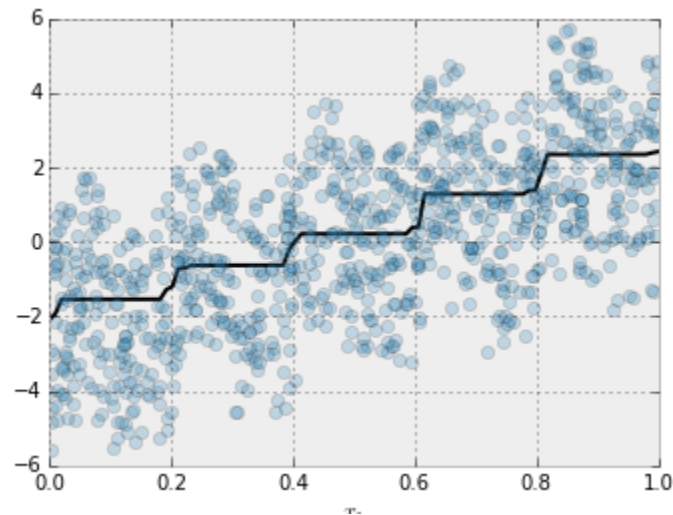
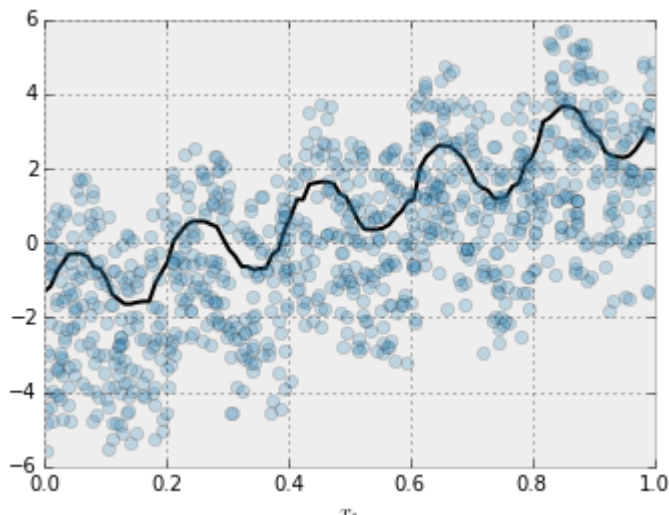
- **Sampling de filas:** qué proporción de las filas tomar para entrenar cada árbol.
- **Ausencia de datos:** a los valores nulos XGBoost le asigna una dirección por default en cada nodo. Los agrupa con los valores de la izquierda o de la derecha del split según cuál sea la dirección que da mejores resultados. De esta manera, no es necesario completar valores nulos para correr este algoritmo.

Mejoras adicionales

- **DART:** en términos generales XGBoost (y los modelos de boosting en general) se basa en el entrenamiento de muchísimos modelos de manera secuencial con un learning rate chico. Esto implica que los primeros árboles son más relevantes que los posteriores. Por este motivo, es incluye un algoritmo llamado DART que aleatoriamente elimina algunos de los árboles para así “liberar” espacio para los árboles siguientes. Más detalles [acá](#).
- DART es un tipo de *booster*.

Mejoras adicionales

- **Restricciones monótonas:** si sube el valor de un feature entonces el valor de la función tiene que ser mayor. A la izquierda un modelo entrenado sin restricciones, a la derecha un modelo con restricciones. [Detalles.](#)



Mejoras adicionales

- **Funciones de pérdida personalizadas:** como mencionamos previamente **XGBoost** las soporta. [Acá](#) hay un ejemplo donde se reimplementa RMSE. Esto puede ser muy importante en ciertos contextos de negocio en lo que sí se conoce el costo de los errores y/o el beneficio de los aciertos. En definitiva, conocer la función objetivo es poder realmente estimar el impacto en el negocio que nuestro modelo podría tener.

XGBoost - otras implementaciones

- **XGBoost** tiene implementación en distintos frameworks que paralelizan, como:
 - [Distributed XGBoost with AWS YARN](#)
 - [Distributed XGBoost on Kubernetes](#)
 - [Distributed XGBoost with XGBoost4J-Spark](#)
 - [Distributed XGBoost with Dask](#)
 - [Distributed XGBoost with Ray](#)

XGBoost - optimización

El overfitting en **XGBoost** lo podemos reducir:

1. Controlando la complejidad del modelo con:
 - a. *max_depth*
 - b. *min_child_weight*
 - c. *gamma*
2. Agregando aleatoriedad, haciendo el entrenamiento más robusto al ruido:
 - a. *subsample*
 - b. *colsample_bytree*
 - c. incorporando más árboles con *n_estimators*, para eso es recomendable reducir el step *eta*

Light GBM

LightGBM

En febrero de 2016 surge la versión beta de LightGBM, un proyecto desarrollado por Microsoft. Este algoritmo se caracteriza por ocupar mucha menos memoria RAM y ser, generalmente, mucho más rápido que el XGBoost original.

En términos generales el algoritmo es muy similar a XGBoost pero incorpora algunas mejoras:

- Histogramas: esto ya lo vimos en XGBoost. Este algoritmo originalmente fue incluido primero en LightGBM.
- GOSS
- Leaf-wise (Best-first) Tree Growth
- Mejoras en el manejo de las variables categóricas
- Optimización en el networking

LightGBM - GOSS

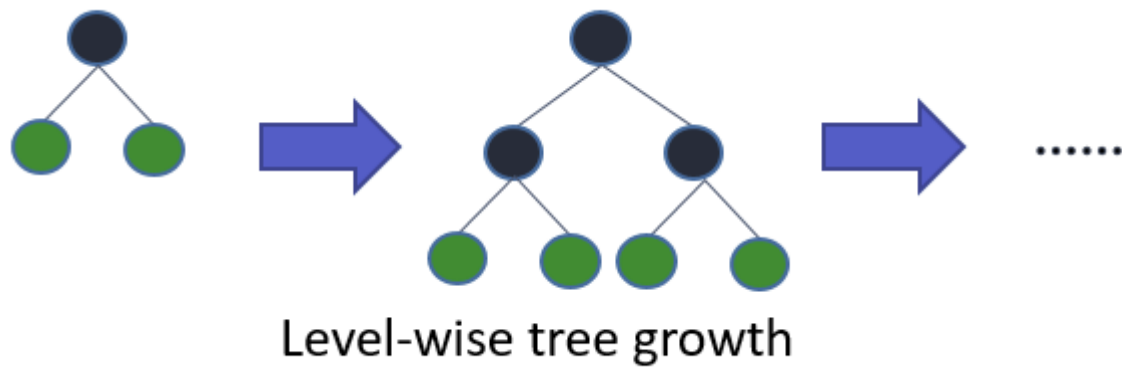
1. LightGBM emplea un algoritmo para buscar los splits llamado **GOSS** (Gradient-based one-side sampling). Este algoritmo hace lo siguiente:
2. Para cada observación calcula el gradiente.
3. Se dejan sólo las instancias con un gradiente elevado y se samplea un porcentaje de las instancias con un gradiente bajo. La idea es que las observaciones con un gradiente chico ya pueden ser bien predichas.
4. Se entrena el árbol con este set de datos, y para evaluar los splits a las observaciones muestreadas se las multiplica por una constante.

LightGBM

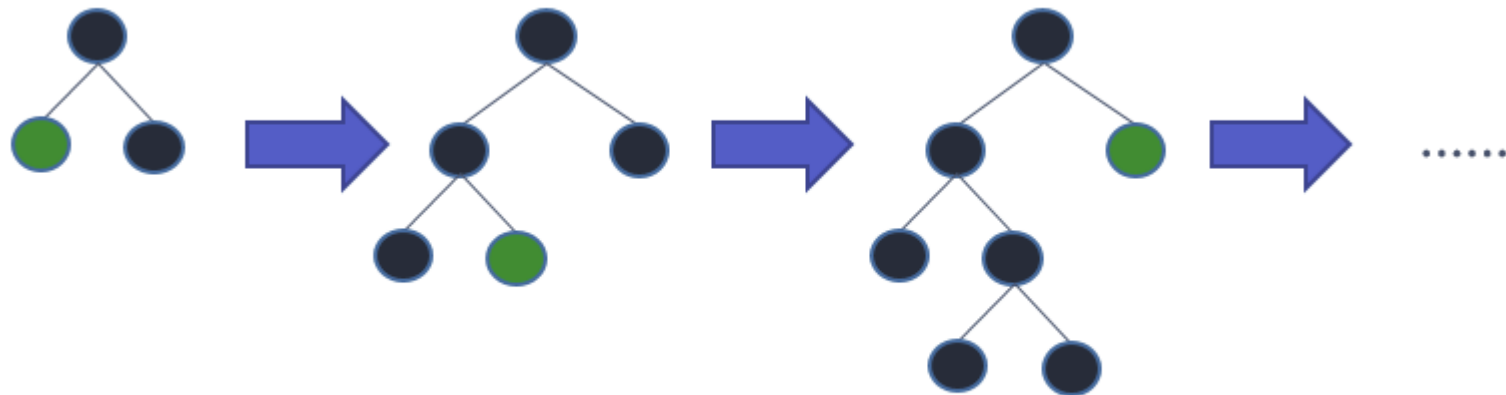
Lightgbm incorpora la creación de los árboles “por hoja” (leaf-wise) y no “por nivel” (level-wise), como lo hacía originalmente **XGBoost**.

Hoy por hoy **XGBoost** incorpora también este método.

LightGBM



LightGBM



Leaf-wise tree growth

LightGBM - Variables categóricas

Realizar one-hot encoding sobre variables categóricas en modelos de boosting de árboles es subóptimo ya que el desbalanceo hace que los árboles deban extenderse demasiado para encontrar una determinada categoría.

Por este motivo, **LightGBM** implementa un método que funciona con categóricas codificadas con enteros. Este método reordena las categorías según su relación con el target y luego procede igual que con cualquier variable numérica, buscando los splits en un histograma.

LightGBM - Variables categóricas

Además, **LightGBM** implementa un método llamado EFB (Exclusive Features Bundling). Este proceso junta variables que son mutuamente excluyentes (es decir que no son simultáneamente distintas de cero), lo cual reduce la *sparsity*.

Este proceso es útil para reducir la dimensionalidad de problema cuando hay muchas variables como *one-hot encoding*.

LightGBM - Optimización

Debido a que el modelo construye árboles *leaf-wise* los principales parámetros a optimizar son:

- **num_leaves**: el número de hojas
- **min_data_in_leaf**: la cantidad de observaciones mínima por hoja.
- **max_depth**: la profundidad máxima.

Es recomendable optimizar estos 3 parámetros conjuntamente.

CatBoost

NGBoost

CatBoost

En 2017 Yandex abre el proyecto de **CatBoost**. Este algoritmo también es muy rápido.

CatBoost permite el manejo de variables categóricas y de texto de manera automática.

Por un lado, las variables con menos categorías que un determinado número son transformadas a dummies.

CatBoost - Variables Categóricas

Por otro lado, permite transformar variables categóricas a numéricas realizando el siguiente procedimiento.

- 1- Primero se reordena el conjunto de observaciones.
- 2- Se transforma en entero la clase
- 3- Se calcula la proporción de casos que caen en la label para la categoría dada (con valor = 1) **hasta** esa observación. Esta última observación se conoce como **Ordered Target Statistics**.

CatBoost - Variables Categóricas

Ordered Target Statistics. Lo interesante de este método es que para prevenir el overfitting el algoritmo crea una suerte de índice temporal sobre el cual calcula la relación entre la categoría y el target.

Es decir, reordena las observaciones y dada una observación K calcula la estadística para K sólo tomando en cuenta la historia de esta variable (excluya a esta observación y a las observaciones posteriores). De este modo, se intenta romper cualquier posible **information leak** en los datos.

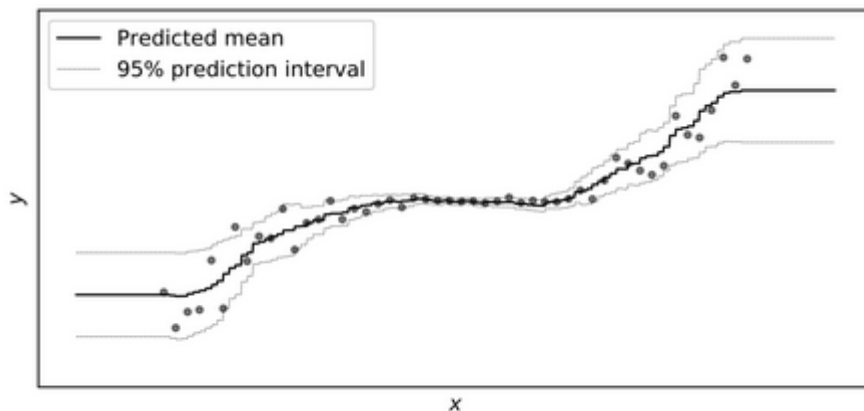
CatBoost - Variables Categóricas

El procedimiento antes descripto CatBoost lo realiza para todas las combinaciones de categorías de variables.

Por ejemplo, supongamos que contamos con dos variables categóricas: género musical (rock, indie o pop) y estilo (danza, clásico). Catboost generará como categorías las combinaciones de las categorías de esas variables: "rock danza", "rock clásico", "indie danza", "indie clásico", "pop danza", y "pop clásico".

NGBoost - Predicción probabilística

Si bien no profundizaremos en él, [NGBoost](#) es un algoritmo que en vez de optimizar la predicción puntual optimiza la predicción de una distribución de probabilidad. De este modo, puede generar intervalos de predicción “nativamente”.



BONUS

TRACK

Referencias

Papers:

- [XGBoost](#)
- [DART](#)
- [LightGBM](#)
- [CatBoost](#)
- [NGBoost](#)

Repositorios:

- [XGBoost](#)
- [LightGBM](#)
- [CatBoost](#)
- [NGBoost](#)

XGBoost	LightGBM
<p>n_estimators: Number of boosted trees to fit.</p> <p>max_depth: Maximum tree depth for base learners. learning_rate: Boosting learning rate (eta)</p> <p>colsample_bytree: Subsample ratio of columns when constructing each tree.</p> <p>subsample: Subsample ratio of the training instance.</p> <p>min_child_weight: Minimum sum of instance weight(hessian) needed in a child.</p> <p>reg_alpha: L1 regularization term on weights</p> <p>reg_lambda: L2 regularization term on weights</p>	
<p>gamma: Minimum loss reduction required to make a further partition on a leaf node of the tree.</p> <p>max_delta_step: Maximum delta step we allow each tree's weight estimation to be.</p> <p>colsample_bylevel: Subsample ratio of columns for each split, in each level. booster: Specify which booster to use: gbtrees, gblinear or dart</p>	<p>num_leaves: max number of leaves in one tree</p> <p>subsample_for_bin: number of data that sampled to construct histogram bins. Setting this to larger value will give better training result, but will increase data loading time. Set this to larger value if data is very sparse</p> <p>min_split_gain: the minimal gain to perform split</p> <p>min_child_samples: minimal number of data in one leaf.</p> <p>subsample_freq: frequency for bagging. 0 means disable bagging; k means perform bagging at every k iteration. to enable bagging, subsample should be set to value smaller than 1.0 as well.</p> <p>boosting_type: puede ser 'gbdt', 'rf', 'dart' y 'goss'.</p>

CatBoost

Algunos parámetros similares a los algoritmos previos: **n_estimators**, **learning_rate**, **max_depth**, **reg_lambda**, **colsample_bylevel**, **subsample**.

border_count (max_bin): The number of splits for numerical features. Allowed values are integers from 1 to 255 inclusively.

od_pval, **od_wait** y **od_type** son parámetros para la detección de overfitting.

nan_mode: 'Forbidden', 'Min', 'Max'.

Además existen parámetros para controlar el manejo de variables categóricas, como **store_all_simple_ctr**, **max_ctr_complexity**, **counter_calc_method**, entre otros.

one_hot_max_size: Use one-hot encoding for all features with a number of different values less than or equal to the given parameter value. Ctrs are not calculated for such features.