



## Boosting Práctico

### Carga de Datos

Primero vamos a cargar y a preparar minimamente (split de test y train) los dos datasets que vamos a emplear: titanic para clasificación y USA\_Housing para regresión.

#### Clasificación

```
In [1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('data/titanic_modified.csv')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Survived    891 non-null   int64
1   Pclass      891 non-null   int64
2   Sex         891 non-null   int64
3   Age         891 non-null   float64
4   SibSp       891 non-null   int64
5   Parch       891 non-null   int64
6   MissingAge  891 non-null   int64
7   Spouse      891 non-null   int64
dtypes: float64(1), int64(7)
memory usage: 55.8 KB
```

```
In [2]: from sklearn.model_selection import train_test_split

X = df.drop(columns=['Survived'])
y = df.Survived

classes = X.columns.values.tolist()

X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                    test_size=0.2,
                                                    random_state=55)
```

## Regresión

```
In [3]: df_reg = pd.read_csv('data/USA_Housing.csv')
X_reg = df_reg.drop(columns=['Price', 'Address'])
y_reg = df_reg['Price']
df_reg.head(2)
```

Out[3]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Address
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06	208 Mi Ferry 674\nLaura NE 3
1	79248.642455	6.002900	6.730821	3.09	40173.072174	1.505891e+06	188 Joh Views 079\r Kathleen,

```
In [4]: X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_reg, y_reg,
                                                                              test_size=0.2
                                                                              random_state=
```

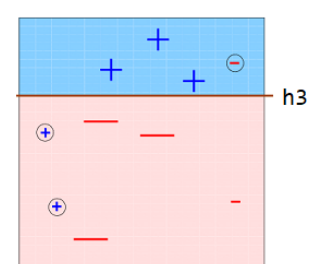
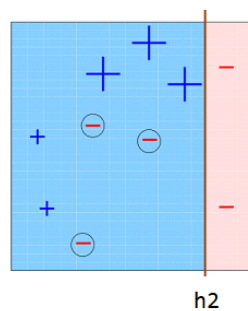
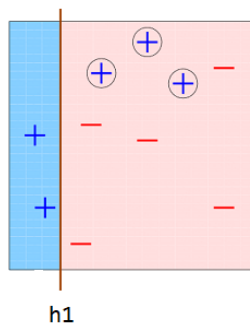
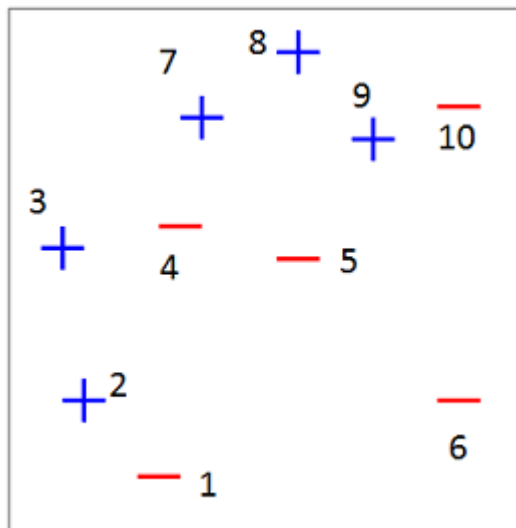
## Algoritmos de Boosting (I): AdaBoost y GradientBoost

En esta técnica los modelos van intentando mejorar su predecesor, recibiendo los errores del mismo, e intentando mejorar su resultado

Estos algoritmos son de los más poderosos cuando trabajamos con datos tabulares

### AdaBoost

AdaBoost (adaptive boosting) consiste en crear varios predictores sencillos en secuencia, de tal manera que el segundo ajuste bien lo que el primero no ajustó, que el tercero ajuste un poco mejor lo que el segundo no pudo ajustar y así sucesivamente.



$$H_{\text{final}} = \text{sign} \left( \alpha_1 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \end{array} + \alpha_2 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \end{array} + \alpha_3 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \end{array} \right) = \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \end{array}$$

Más info: <https://youtu.be/LsK-xG1cLYA>

### AdaBoostClassifier

```
In [5]: from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import classification_report

ada_clf = AdaBoostClassifier(n_estimators=200, random_state=42)

ada_clf.fit(X_train, y_train)

y_pred = ada_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.82	0.85	0.84	110
1	0.75	0.71	0.73	69
accuracy			0.80	179
macro avg	0.79	0.78	0.79	179
weighted avg	0.80	0.80	0.80	179

### ¿Qué hiperparámetros debería tocar en el AdaBoostClassifier?

1. **n\_estimators** : número de árboles que participarán en la corrección secuencial del error del modelo. Si corregimos el error a la perfección el algoritmo termina de entrenar. Cuantos más estimadores, mejor corregiremos el error pero mayor probabilidad de caer en overfitting. Valores superiores a 100 suelen sobreajustar el modelo aunque dependerá de la complejidad y volumen de los datos. Este es el valor que generalmente tocaremos.
2. **learning\_rate** : no suele tener valores superiores a 1. Cuanto más alto, más aporta cada nuevo árbol, más preciso, pero caemos en overfitting. En general, no lo tocaremos.
3. **estimator** : se suele dejar por defecto, aunque podría encajar un SVM o una RegresiónLogística. En general, tampoco lo tocaremos. Ah y hay que pasarle un objeto del tipo de modelo que quieras emplear.

No, no hay hiperparámetros para los árboles internos. Recuerda que son "stumps" o tocones, árboles binarios con un único nivel de profundidad, no hay más que tocar.

Antes de hacer una optimización, veamos la regresión con AdaBoostRegressor (que realmente se centra en poner pesos a los errores en la regresión de los modelos anteriores, por lo menos en [la implementación de sklearn](#), que implementa el algoritmo conocido como AdaBoost.R2)

### AdaBoostRegressor

```
In [6]: from sklearn.ensemble import AdaBoostRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, mean_absolute_error
import numpy as np

ada_reg = AdaBoostRegressor(n_estimators=200,
                             random_state=42)

ada_reg.fit(X_train_reg,
            y_train_reg)

y_pred_reg = ada_reg.predict(X_test_reg)
print("MAE:", mean_absolute_error(y_test_reg, y_pred_reg))
print("RMSE:", np.sqrt(mean_squared_error(y_test_reg, y_pred_reg)))
```

MAE: 100947.80757390938

RMSE: 130082.4839115346

Puede comparar con sesiones anteriores y verás que es un error menor que en RandomForest (sin optimizar hiperparámetros)

## Optimización de hiperparámetros

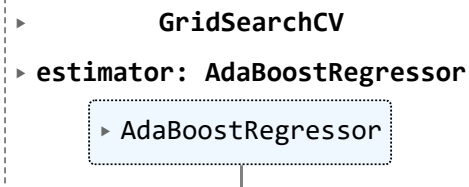
```
In [7]: from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

param_grid = {
    "n_estimators": [100,400],
    "learning_rate": [0.1,1,3.2]
}

model_grid = GridSearchCV(ada_reg,
                           param_grid= param_grid,
                           cv = 5,
                           scoring = "neg_mean_squared_error") # Se intenta maximizar

model_grid.fit(X_train_reg,y_train_reg)
```

```
Out[7]:
```



```

  ▸ GridSearchCV
  ▸ estimator: AdaBoostRegressor
    ▸ AdaBoostRegressor

```

```
In [8]: model_grid.best_params_
```

```
Out[8]: {'learning_rate': 3.2, 'n_estimators': 400}
```

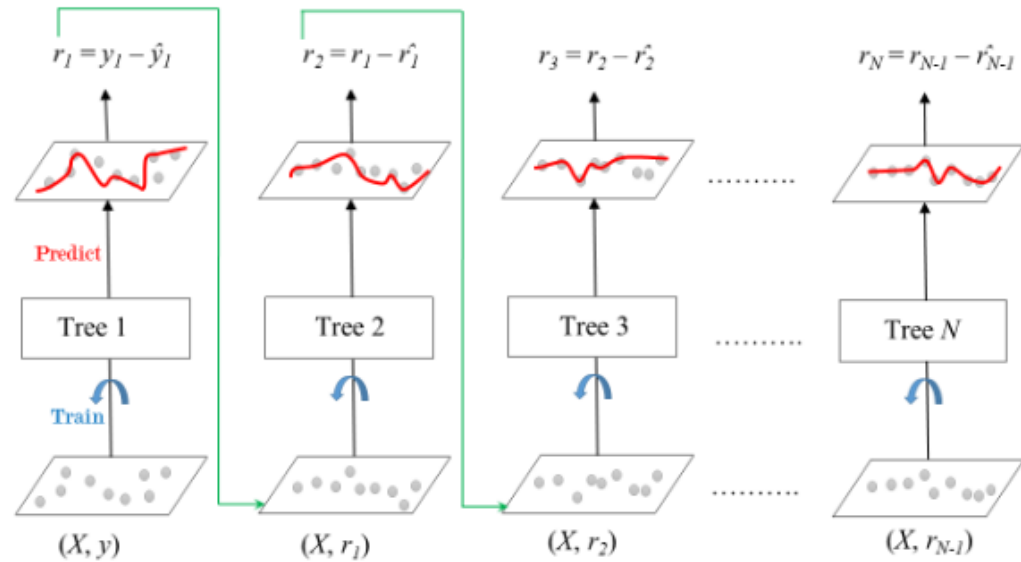
```
In [9]: np.sqrt(-model_grid.best_score_)
```

```
Out[9]: 128959.73670565053
```

## GradientBoosting

Al igual que el AdaBoost, el GradientBoost trabaja sobre un conjunto secuencial de modelos (árboles de decisión), tratando de corregir a su predecesor. Sin embargo, cuando el AdaBoost iba actualizando los pesos de cada observación, el GradientBoosting intenta ajustar y minimizar los errores (residuos) del modelo predecesor.

El modelo final será una combinación lineal de todos los estimadores.

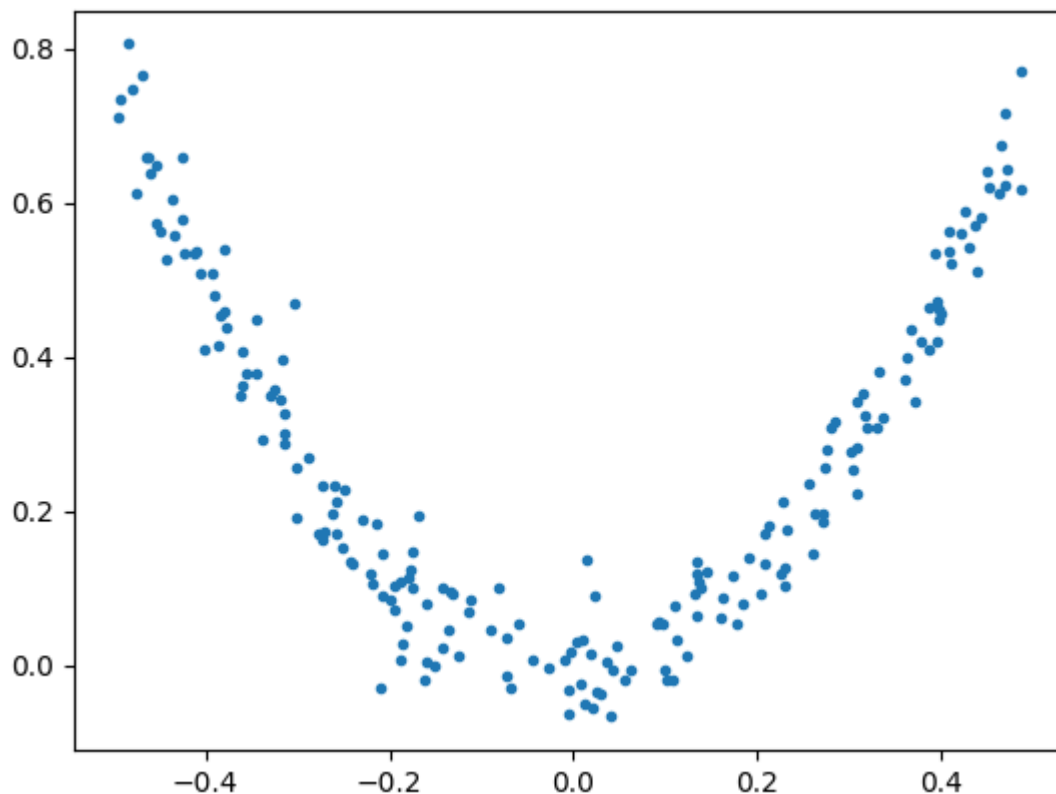


### GradientBoostingRegressor (cómo funciona, más o menos)

Para exponer con un poco más de detalle el funcionamiento de los algoritmos de boosting, vamos a construir un problema muy similar al que usamos para ver los árboles de decisión aplicados a regresión y luego ese mismo problema lo vamos a resolver simulando un mecanismo de boosting.

```
In [10]: np.random.seed(42)
X = np.random.rand(200,1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * np.random.randn(200)
X_test_sim = np.random.rand(100,1) - 0.5
y_test_sim = 3*X_test_sim[:, 0]**2 + 0.05 * np.random.randn(100)
plt.plot(X,y, '.')
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x7f99e17ee3a0>]
```



```
In [11]: from sklearn.tree import DecisionTreeRegressor
```

```
tree_reg1 = DecisionTreeRegressor(max_depth=2,
                                   random_state=42)
tree_reg1.fit(X,y)
```

```
Out[11]: ▾ DecisionTreeRegressor
```

```
DecisionTreeRegressor(max_depth=2, random_state=42)
```

```
In [12]: y2 = y - tree_reg1.predict(X)
```

```
In [13]: tree_reg2 = DecisionTreeRegressor(max_depth=2,
                                             random_state=42)
tree_reg2.fit(X,y2)
```

```
Out[13]: ▾ DecisionTreeRegressor
```

```
DecisionTreeRegressor(max_depth=2, random_state=42)
```

```
In [14]: y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2,
                                   random_state=42)
tree_reg3.fit(X,y3)
```

```
Out[14]: ▾ DecisionTreeRegressor
```

```
DecisionTreeRegressor(max_depth=2, random_state=42)
```

```
In [15]: X_new = np.array([[0.8]])
y_pred = sum([tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3)])
```

```
In [17]: y_pred_sim = sum([tree.predict(X_test_sim) for tree in (tree_reg1, tree_reg2, tr
```

```
In [18]: y_tree_reg1_pred = tree_reg1.predict(X_test_sim)
```

```
In [19]: print("MAE, arbol 1:", mean_absolute_error(y_test_sim, y_tree_reg1_pred))
print("MAE, sim GBT:", mean_absolute_error(y_test_sim, y_pred_sim))
```

MAE, arbol 1: 0.10329841215879393

MAE, sim GBT: 0.06395037885778372

Pintemos como está trabajando nuestro boosting de tres árboles:

```
In [20]: def plot_predictions(regressors, X, y, axes, label=None, style="r-", data_style=
x1 = np.linspace(axes[0], axes[1], 500)
y_pred = sum(regressor.predict(x1.reshape(-1, 1)) for regressor in regressor
plt.plot(X[:, 0], y, data_style, label=data_label)
plt.plot(x1, y_pred, style, linewidth=2, label=label)
if label or data_label:
    plt.legend(loc="upper center", fontsize=16)
plt.axis(axes)
```

```
In [21]: plt.figure(figsize=(11,11))

plt.subplot(321)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$h_1(x_1)$",
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Residuals and tree predictions", fontsize=16)

plt.subplot(322)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$h(x_1)$",
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Ensemble predictions", fontsize=16)

plt.subplot(323)
plot_predictions([tree_reg2], X, y2, axes=[-0.5, 0.5, -0.5, 0.5], label="$h_2(x_1)$",
plt.ylabel("$y - h_1(x_1)$", fontsize=16)

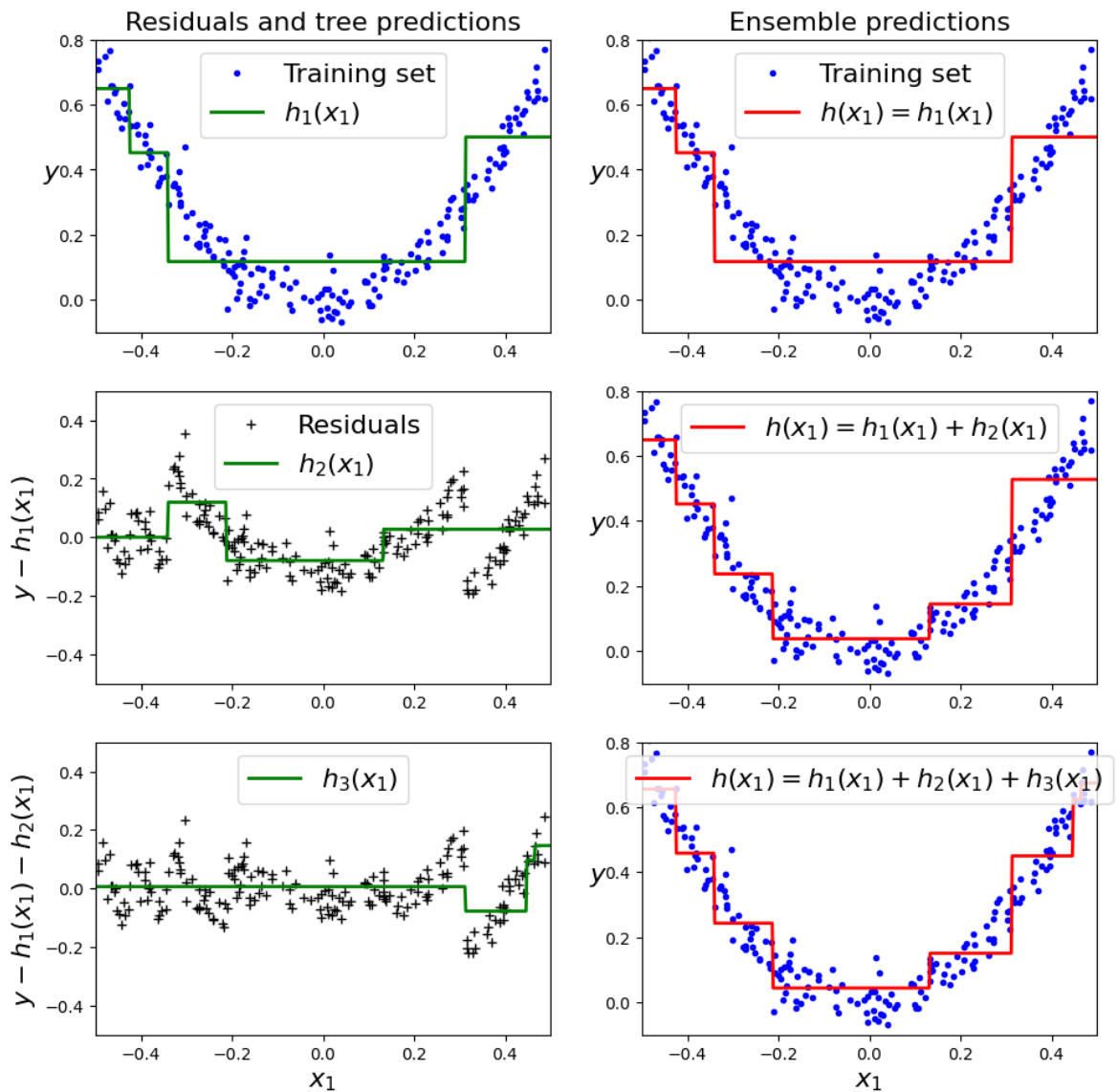
plt.subplot(324)
plot_predictions([tree_reg1, tree_reg2], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="$h_2(x_1)$",
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.subplot(325)
plot_predictions([tree_reg3], X, y3, axes=[-0.5, 0.5, -0.5, 0.5], label="$h_3(x_1)$",
plt.ylabel("$y - h_1(x_1) - h_2(x_1)$", fontsize=16)
plt.xlabel("$x_1$", fontsize=16)

plt.subplot(326)
plot_predictions([tree_reg1, tree_reg2, tree_reg3], X, y, axes=[-0.5, 0.5, -0.1, 0.8],
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.show()
```





El truco del gradient reside en lo siguiente:

$$\hat{y} = \alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x) + \dots$$

Es decir ahora tienen unos pesos y esos pesos se obtienen de aplicar gradiente descendente al error cuadrático medio de  $y_{\text{real}}$  con  $\hat{y}$ . Es decir en este tipo de algoritmos no sólo son parámetros los puntos de cortes de cada uno de los árboles sino también el coeficiente que determina el peso de cada árbol en la decisión final.

Y ahora ya sí:

```
In [22]: from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor()

gbr.fit(X_train_reg, y_train_reg)
y_pred = gbr.predict(X_test_reg)

print("MAE:", mean_absolute_error(y_test_reg, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test_reg, y_pred)))
```

MAE: 85550.98776896918

RMSE: 106434.51271746648

Mucho mejor, eh... Sí ya vas saboreando el poder del boosting. Veamos si también podemos tocarlo ya en clasificación.

### GradientBoostingClassifier

```
In [23]: from sklearn.ensemble import GradientBoostingClassifier

gbc = GradientBoostingClassifier()

gbc.fit(X_train, y_train)
y_pred = gbc.predict(X_test)

print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.83	0.88	0.85	110
1	0.79	0.71	0.75	69
accuracy			0.82	179
macro avg	0.81	0.80	0.80	179
weighted avg	0.81	0.82	0.81	179

La mejora no es tan evidente (de hecho en algún caso no la hay, no el recall\_medio), pero mejora el accuracy medio de los modelos ajustados de RandomForest,... pero este sin ajustar.

### ¿Qué hiperparámetros debería tocar en el GradientBoosting?

Como los RandomForest, los GBT basados en árboles tienen hiperparámetros de modelo y tienen hiperparámetros de submodelo:

De modelo:

1. **n\_estimators** : número de árboles que participarán en la corrección secuencial del error del modelo. Si corregimos el error a la perfección el algoritmo termina de entrenar. Cuantos más estimadores, mejor corregiremos el error pero mayor probabilidad de caer en overfitting. Valores superiores a 100 suelen sobreajustar el modelo aunque dependerá de la complejidad y volumen de los datos.
2. **learning\_rate** : no suele tener valores superiores a 1. Cuanto más alto, más aporta cada nuevo árbol, más preciso, pero caemos en overfitting. **Importante:** un learning rate bajo y alto número de estimadores no necesariamente tiene por qué aumentar la precisión y si va a inducir en altos costes computacionales.

Observa que al incorporar un mecanismo de optimización de función de pérdida (el gradiente descendente sobre el error cuadrático medio) incluye al menos un hiperparámetro de learning\_rate, pero también tiene otros relacionados (que no vamos a profundizar, como tolerance (tol) o máximo número de iteraciones)

De submodelo (árboles de decisión):

3. `max_depth` : profundidad de los árboles. Cuanto más profundos, más complejo es el modelo, pero menos generaliza. De nuevo, cuanto más complejo es el problema, mayor profundidad necesitaremos. No más de 20/30 es lo normal.
4. `max_features` : features a tener en cuenta en los splits del árbol. Cuanto más bajo, mejor generalizará y menos overfitting. Numero menor a la cantidad de features del dataset, sino dará error. Ojo porque por defecto está puesto a la raíz cuadrada del número total de features.
5. `min_samples_split` : mínima cantidad de muestras en un nodo antes de ser splitado. 2 por defecto. Números bajos suelen dar buenos resultados (<50). Cuanto más alto, mejor generaliza, pero más baja la precisión.
6. `min_samples_leaf` : mínima cantidad de puntos permitidos en un `leaf node` , es decir, un nodo que no va a volver a ser splitado. Valores bajos funcionan bien (<50).

---

## Algoritmos de Boosting (II): XGBoost, LightGBM, CatBoost

Como ya sabes una de las limitaciones de los algoritmos de boosting es que los modelos entrenan secuencialmente, no hay otra forma de hacerlo y eso hace que de primeras sean lentos. Por eso con el tiempo han surgido implementaciones del GBT que han buscado acelerar los procesos sobre todo en los entrenamientos de los submodelos intermedios (por ejemplo la obtención de los valores de corte de las features de ramas diferentes en un árbol se pueden hacer en paralelo, o las variables numérica se pueden binnear y eso acelera el proceso de obtención de valores y no afecta excesivamente al árbol).

Tres implementaciones "ganan" a día de hoy y son las que vamos a ver, que no solo son más rápidas que un GBT sino que además añaden sus "cositas".

## XGBoost

**XGBoost** (exTreme Gradient Boosting) es un algoritmo que se enfoca en la velocidad de computación y el desempeño del modelo. Hace especial hincapié en la regularización (con una forma particular de aplicar penalización a la función de pérdida) y en cómo construir los árboles en cada submodelo.

XGBoost permite regularizar el modelo y puede manejar missings, por lo que no es necesario tener el dataset perfectamente limpio

### XGBRegressor

```
In [24]: import xgboost

xgb_reg = xgboost.XGBRFRegressor(random_state=42)
xgb_reg.fit(X_train_reg, y_train_reg)
y_pred = xgb_reg.predict(X_test_reg)
```

```
print("MAE:", mean_absolute_error(y_test_reg, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test_reg, y_pred)))
```

MAE: 118737.76186077629

RMSE: 149788.570052806

### XGBClassifier

```
In [25]: xgb_clas = xgboost.XGBClassifier(random_state=42)

xgb_clas.fit(X_train, y_train)
y_pred = xgb_clas.predict(X_test)
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.82	0.89	0.86	110
1	0.80	0.70	0.74	69
accuracy			0.82	179
macro avg	0.81	0.79	0.80	179
weighted avg	0.81	0.82	0.81	179

### ¿Qué hiperparámetros debería tocar en el XGB?

Parámetros del modelo:

1. `n_estimators` : igual que para el GradientBoosting. Un detalle es que el número de submodelos puede ser menor
2. `learning_rate` : o también llamado `eta` . Como el learning rate del GradientBoosting, sirve para modular el aporte de cada nuevo árbol que se añade a la secuencia. Tiene el mismo comportamiento que un learning\_rate en otros modelos, a mayor valor más rápido converge menor es la capacidad de predicción, a menor valor más tarda en converger, mayor es la capacidad (en general) y se corre el riesgo de overfitting.
- `scale_pos_weight` : es un factor de escala para aplicar a la función de pérdida para los casos en clasificación en las que el dataset está desbalanceado.

Parámetros de los submodelos:

3. `max_depth` : máximo nivel de profundidad de los árboles.
4. `min_child_weight` : equivalente a "min\_sample\_split" en otros modelos con árboles de sklearn.
5. `subsample` : muestreo del dataset para cada árbol (equivalente a max\_samples en Random Forest)
6. `colsample_bytree` : fracción del total de variables por árbol (con un objetivo más o menos similar al "max\_features" en Random Forest)

Si quieres afinar más todavía el XGBoost consulta [esta completa guía](#).

Vamos a completar la lista de modelos y al final haremos la optimización de hiperparámetros para los tres

# LightGBM

Similar a XgBoost, desarrollado por Microsoft, es ligeramente más rápido que el resto de los que vamos a ver en esta sesión. Sin entrar en muchos detalles tiene una forma diferente de construcción de los árboles que le hace más rápido con grandes volúmenes. En general, puede generalizar peor que XGBoost que incluye la regularización como algo estándar. Maneja variables categóricas sin procesar aunque no tan bien como lo hace CatBoost.

## LGBMRegressor

```
In [26]: from lightgbm import LGBMRegressor
from sklearn.metrics import mean_absolute_error

lgbm_reg = LGBMRegressor(random_state=42)
lgbm_reg.fit(X_train_reg, y_train_reg)
y_pred = lgbm_reg.predict(X_test_reg)
print("MAE:", mean_absolute_error(y_test_reg, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test_reg, y_pred)))
```

```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing
was 0.000311 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1255
[LightGBM] [Info] Number of data points in the train set: 4000, number of used fe
atures: 5
[LightGBM] [Info] Start training from score 1232594.044782
MAE: 87803.38881114805
RMSE: 109959.13432006085
```

## LGBMClassifier

```
In [27]: df_t = pd.read_csv("./data/titanic.csv")
```

```
In [28]: df_t.head()
```

Out[28]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	
<b>0</b>	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2
<b>1</b>	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2
<b>2</b>	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9
<b>3</b>	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1
<b>4</b>	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0

```
In [29]: features_lgbm = ["Pclass", "Sex", "Embarked", "Age", "Fare"]
features_cat = ["Sex", "Embarked"]
```

```
In [30]: df_t["Sex"] = df_t["Sex"].astype("category")
df_t["Embarked"] = df_t["Embarked"].astype("category")
```

```
In [31]: df_t.dtypes
```

```
Out[31]: PassengerId      int64
Survived      int64
Pclass      int64
Name      object
Sex      category
Age      float64
SibSp      int64
Parch      int64
Ticket      object
Fare      float64
Cabin      object
Embarked      category
dtype: object
```

```
In [32]: train_set, test_set = train_test_split(df_t, test_size=0.2, random_state= 42)
```

```
In [33]: X_train_lgbm = train_set[features_lgbm]
y_train_lgbm = train_set["Survived"]

X_test_lgbm = test_set[features_lgbm]
y_test_lgbm = test_set["Survived"]
```

```
In [34]: from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score
lgbm_clas = LGBMClassifier(random_state=42)

lgbm_clas.fit(X_train_lgbm, y_train_lgbm, categorical_feature= features_cat)
```

[illegible]



```
Out[34]: ▼      LGBMClassifier
          LGBMClassifier(random_state=42)
```

	precision	recall	f1-score	support
0	0.82	0.87	0.84	105
1	0.79	0.73	0.76	74
accuracy			0.81	179
macro avg	0.81	0.80	0.80	179
weighted avg	0.81	0.81	0.81	179

**¿Qué hiperparámetros debería tocar en el LightGBM?** Relacionados con el modelo en general:

- `n_estimators` : Número de submodelos o estimadores máximos que contendrá mi modelo ensamblado final.
- `learning_rate` : Tasa de aprendizaje. Un valor más pequeño significa un entrenamiento más lento pero puede mejorar la precisión del modelo. Un valor alto acelera el entrenamiento pero puedes perder precisión o capacidad (posibilidad de no converger al óptimo correcto)
- `max_bin` : Controla el número máximo de bins en los que se dividen las variables numéricas continuas para acelerar la generación de árboles. Por defecto está a 255.

Relacionados con los submodelos:

1. `max_depth` – Similar to XGBoost, this parameter instructs the trees to not grow beyond the specified depth. A higher value increases the chances for the model to overfit.
2. `num_leaves` – This parameter is very important in terms of controlling the complexity of the tree. The value should be less than  $2^{(\text{max\_depth})}$  as a leaf-wise tree is much deeper than a depth-wise tree for a set number of leaves. Hence, a higher value can induce overfitting.
3. `min_data_in_leaf` – The parameter is used for controlling overfitting. A higher value can stop the tree from growing too deep but can also lead the algorithm to learn less (underfitting). According to the LightGBM's official documentation, as a best practice, it should be set to the order of hundreds or thousands.
4. `feature_fraction` – Similar to `colsample_bytree` in XGBoost
5. `bagging_fraction` – Similar to `subsample` in XGBoost

## Catboost

Su nombre combina *Category* y *Boosting*, y funciona bien cuando tenemos muchas variables categóricas (de hecho, [las admite como inputs](#)) y texto (veremos en futuros sprints como tratar variables tipo texto libre, es decir frases, párrafos en lenguaje natural, etc)

```
In [36]: import pandas as pd
df = pd.read_csv('https://raw.githubusercontent.com/tidyverse/ggplot2/main/data-
df.head()
```

Out[36]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

In [37]: `X = df.drop(columns=['price'])`  
`y = df.price`

In [38]: `X_train2, X_test2, y_train2, y_test2 = train_test_split(X,y,test_size=0.2,`  
`random_state=1)`

In [39]: `from catboost import CatBoostRegressor, Pool`  
`from sklearn.metrics import mean_squared_error`  
`cb = CatBoostRegressor(n_estimators=200,`  
`loss_function='RMSE',`  
`learning_rate=0.4,`  
`random_state=1,`  
`verbose = False`  
`)`  
`pool_train = Pool(X_train2, y_train2,`  
`cat_features=['cut','color','clarity'])`  
`pool_test = Pool(X_test2, cat_features=['cut','color','clarity'])`  
`cb.fit(pool_train)`  
`y_pred = cb.predict(pool_test)`  
`print('RMSE:', np.sqrt(mean_squared_error(y_test2,y_pred)))`

RMSE: 529.5700855716823

In [40]: `from catboost import CatBoostClassifier`  
`cb = CatBoostClassifier(iterations = 200,`  
`learning_rate = 0.9,`  
`verbose = False)`  
`pool_train = Pool(X_train, y_train)`  
`pool_test = Pool(X_test)`  
`cb.fit(pool_train)`

Out[40]: `<catboost.core.CatBoostClassifier at 0x7f99df017970>`

In [41]: `y_pred_cat = cb.predict(pool_test)`  
`print(classification_report(y_test,y_pred_cat))`

	precision	recall	f1-score	support
0	0.79	0.87	0.83	110
1	0.76	0.64	0.69	69
accuracy			0.78	179
macro avg	0.78	0.76	0.76	179
weighted avg	0.78	0.78	0.78	179

## Hiperparámetros:

CatBoost ofrece una variedad de hiperparámetros para controlar el entrenamiento del modelo y mejorar su rendimiento. Algunos de los hiperparámetros más importantes son:

Del modelo:

- `iterations` : Número de árboles a entrenar. Es equivalente a `n_estimators` en otros modelos de boosting.
- `l2_leaf_reg` : Coeficiente para el término de regularización L2 del coste. Puede ser utilizado para balancear la sobre-adaptación del modelo.
- `border_count` : Número de divisiones para características numéricas. Es similar a `max_bin` en LightGBM. (XGBoost, Catboost y LightGBM hacen binning de las variables numéricas y usa los índices en vez de los valores reales para hacer los árboles internos, por eso el número de bins o "categorias" para discretizar los valores numéricos es un hiperparámetro)
- `cat_features` : Índices de características categóricas. CatBoost manejará estas características internamente, convirtiéndolas de manera óptima. [OJO ESTE NO ES UN HIPERPARAMETRO EN EL MISMO SENTIDO CON EL QUE TRATAMOS AL RESTO, pero permite emplearla como hiperparámetro en vez de hacer feature selection en el train]
- `learning_rate` : Tasa de aprendizaje del algoritmo. Un valor más bajo hace que el modelo sea más robusto a la sobre-adaptación, pero puede requerir más iteraciones.

De los árboles:

- `depth` : Profundidad de los árboles. Controla la complejidad del modelo. (Como `max_depth` en otros modelos basados en árboles)

## Ajuste de hiperparámetros

Con todos los algoritmos vistos en este notebook, es posible aplicar la optimización de hiperparámetros, utilizando las técnicas que vimos anteriormente

```
In [42]: from sklearn.model_selection import RandomizedSearchCV
import xgboost

params_xgb = {'max_depth': [3, 6, 12],
               'learning_rate': [0.1, 0.2, 0.3, 0.4],
               'subsample': [0.3, 0.6, 1],
               'colsample_bytree': [0.5, 1],
```

```

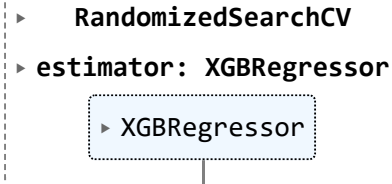
        'n_estimators': [100, 250, 500, 750],
    }

xgb = xgboost.XGBRegressor()

clf = RandomizedSearchCV(estimator=xgb,
                        cv=4,
                        n_iter=10,
                        param_distributions=params_xgb,
                        scoring='neg_mean_squared_error'
                        )
clf.fit(X_train_reg, y_train_reg)

```

Out[42]:



In [43]: `clf.best_params_`

Out[43]:

```

{'subsample': 0.6,
 'n_estimators': 250,
 'max_depth': 3,
 'learning_rate': 0.3,
 'colsample_bytree': 0.5}

```

In [44]:

```

y_pred = clf.predict(X_test_reg)
print("MAE:", mean_absolute_error(y_test_reg, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test_reg, y_pred)))

```

MAE: 87579.44626139004  
RMSE: 110164.82499637989

## Para LightGBM

In [45]:

```

params_lgb = {'max_depth': [3, 6, 12],
              'learning_rate': [0.1, 0.2, 0.3, 0.4],
              'bagging_fraction': [0.3, 0.6, 1],
              'feature_fraction': [0.5, 1],
              'n_estimators': [100, 250, 500, 750],
              'max_bins': [125, 250]
              }

lgb = LGBMRegressor()

clf = RandomizedSearchCV(estimator=lgb,
                        cv=4,
                        n_iter=2,
                        param_distributions=params_lgb,
                        scoring='neg_mean_squared_error'
                        )
clf.fit(X_train_reg, y_train_reg)

```

```
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000122 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 623
[LightGBM] [Info] Number of data points in the train set: 3000, number of used features: 5
[LightGBM] [Info] Start training from score 1237664.563886
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000137 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 624
[LightGBM] [Info] Number of data points in the train set: 3000, number of used features: 5
[LightGBM] [Info] Start training from score 1235174.782152
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
```

```

[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000088 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 623
[LightGBM] [Info] Number of data points in the train set: 3000, number of used features: 5
[LightGBM] [Info] Start training from score 1226241.472329
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000084 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 624
[LightGBM] [Info] Number of data points in the train set: 3000, number of used features: 5
[LightGBM] [Info] Start training from score 1231295.360761
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] feature_fraction is set=0.5, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=0.5
[LightGBM] [Warning] bagging_fraction is set=0.3, subsample=1.0 will be ignored. Current value: bagging_fraction=0.3
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] feature_fraction is set=1, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=1.0 will be ignored. Current value: bagging_fraction=1
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] feature_fraction is set=1, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=1.0 will be ignored. Current value: bagging_fraction=1

```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -1inf
```



<https://campusonline.thebridge.tech/ultra/courses/421/outline/lti/launchFrame?toolHref=https://campusonline.thebridge.tech/webapp/> 25/33

[illegible]

<https://campusonline.thebridge.tech/ultra/courses/421/outline/lti/launchFrame?toolHref=https://campusonline.thebridge.tech/webapp/> 27/33

```

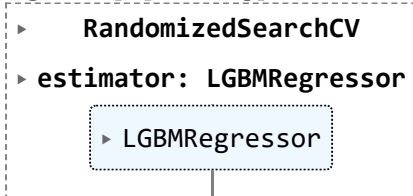
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] feature_fraction is set=1, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=1.0 will be ignored. Current value: bagging_fraction=1
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] feature_fraction is set=1, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=1.0 will be ignored. Current value: bagging_fraction=1
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Warning] feature_fraction is set=1, colsample_bytree=1.0 will be ignored. Current value: feature_fraction=1
[LightGBM] [Warning] bagging_fraction is set=1, subsample=1.0 will be ignored. Current value: bagging_fraction=1
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_leaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000143 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 624
[LightGBM] [Info] Number of data points in the train set: 3000, number of used features: 5
[LightGBM] [Info] Start training from score 1231295.360761
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf

```

[illegible]

<https://campusonline.thebridge.tech/ultra/courses/421/outline/lti/launchFrame?toolHref=https://campusonline.thebridge.tech/webapp/> 30/33

Out[45]:



```
clf.best_params_
```

```
{'n_estimators': 100,  
  'max_depth': 6,  
  'max_bins': 125,  
  'learning_rate': 0.1,  
  'feature_fraction': 1,  
  'bagging_fraction': 1}
```

## Para CatBoost

```
params_cat = {'depth': [3, 6, 12],
              'learning_rate': [0.1, 0.2, 0.3, 0.4],
              #'bagging_fraction': [0.3, 0.6, 1], No hay hiperparámetro equivalente
              'colsample_bylevel': [0.5, 1],
              'iterations': [100, 250, 500, 750],
              "border_count": [125, 250]}
```

```

    }

cat_model = CatBoostRegressor(verbose = False)

clf = RandomizedSearchCV(estimator=cat_model,
                        cv=4,
                        n_iter=2,
                        param_distributions=params_cat,
                        scoring='neg_mean_squared_error'
                        )
clf.fit(X_train_reg, y_train_reg)

```

Out[47]:

```

RandomizedSearchCV
  estimator: CatBoostRegressor
    CatBoostRegressor

```

In [48]: `clf.best_params_`

```

Out[48]: {'learning_rate': 0.3,
          'iterations': 750,
          'depth': 3,
          'colsample_bylevel': 0.5,
          'border_count': 250}

```

En los ejercicios veremos un par de hiperparámetros relacionados con la técnica de histogram gradient boosting para LightGBM y CatBoost.

## Tabla comparativa cuándo usar cuál modelo

## Tabla comparativa: Elección de algoritmos de boosting

Característica / Algoritmo	Gradient Boosted Trees (sklearn)	XGBoost	CatBoost	LightGBM
<b>Manejo de grandes conjuntos de datos</b>	Menos eficiente	Eficiente	Eficiente	Muy eficiente
<b>Velocidad de entrenamiento</b>	Rápido	Muy rápido	Rápido	Muy rápido
<b>Uso de recursos computacionales</b>	Moderado	Moderado/Alto	Moderado	Bajo
<b>Manejo de características categóricas</b>	Requiere pre-procesamiento	Requiere pre-procesamiento	Excelente (soporte nativo)	Bueno (soporte nativo)
<b>Rendimiento con configuración por defecto</b>	Bueno	Bueno	Excelente	Bueno



Característica / Algoritmo	Gradient Boosted Trees (sklearn)	XGBoost	CatBoost	LightGBM
Prevención de sobreajuste	Bueno (con ajuste de hiperparámetros)	Bueno (con ajuste de hiperparámetros)	Excelente	Bueno
Soporte para optimización de hiperparámetros	Amplio	Amplio	Amplio	Amplio
Facilidad de uso e integración	Alta (especialmente con el ecosistema de sklearn)	Alta	Alta	Alta
Popularidad y comunidad	Alta	Muy alta	Alta	Alta

**Nota:** Esta tabla proporciona una visión general y las decisiones deben basarse en pruebas empíricas y requisitos específicos del proyecto.

**Lecturas recomendadas:**

[When to Choose CatBoost Over XGBoost or LightGBM](#)

[XGBoost vs LightGBM: How Are They Different](#)

[XGBoost: Everything You Need to Know](#)

[Understanding LightGBM Parameters \(and How to Tune Them\)](#)