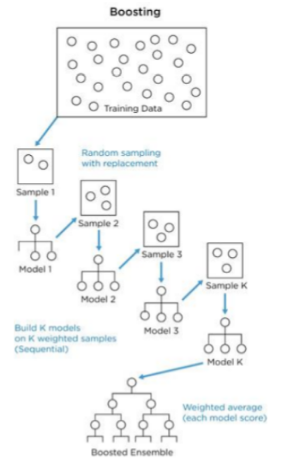
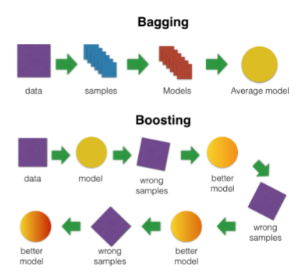
**Boosting**

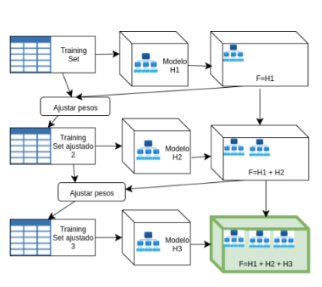
A diferencia de Bagging y Random Forest, donde lo que se hace es entrenar modelos en forma paralela con distintos subsets de Entrenamiento, **Boosting** es una **técnica de ensamble** donde lo que se hace es un **entrenamiento secuencial**.



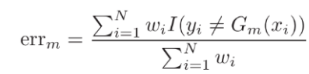
Se trata de un **procedimiento iterativo** donde se va construyendo un **modelo final en pasos.** En cada paso se intentará **aprender de los errores de los pasos previos**, dándole **más peso** a las **observaciones mal clasificadas** de **iteraciones previas**. La forma que tiene esta técnica de **trabajar con los errores previos** es o bien **usándolos para cambiar la ponderación**, o bien **entrenando un modelo que los prediga.** Análogamente a como es con bagging, en **modelos de clasificación** se decide por **mayoría ponderada de votos**; en **modelos de regresión** se decide por **suma ponderada**.



**Adaboost:** Se usan **pesos uniformes** en la **primera iteración**. En las **iteraciones subsiguientes**, se **ajustan los pesos** para **enfatizar los errores** de la iteración anterior. La **predicción final** se hace con **votos ponderados** de los distintos modelos base. La **ponderación de estos votos** depende del **error de entrenamiento** de cada modelo. La **lógica de Adaboost** consiste en **tomar** un **modelo de base débil** **y** en **fortalecerlo reentrenándolo** con **muestras mal clasificadas**.



1. Se **empieza** con un **peso wi uniforme igual a 1/N**, siendo N la cantidad de observaciones del subset de entrenamiento.
2. El **algoritmo entrena M clasificadores**.
3. El clasificador **Gm** se **entrena considerando el set de entrenamiento y el peso wi** de cada observación del mismo.
4. Ahora pasa a calcularse el **error de clasificación ponderado de Gm**. Esto se logra multiplicando el **peso ponderado de los ejemplos mal clasificados** y dividiéndolo por la **suma de los pesos mínimos de 0 cuando no haya errores y máximos de 1 cuando sean todos errores**:



1. Se pasa a calcular el **coeficiente de aporte de este Clasificador Gm en el ensamble**. **Cuanto más preciso** sea este clasificador Gm, **mayor será este valor** y **más pesará el voto** de este clasificador **en la predicción final**.



1. **Se recalculan los pesos de las observaciones** del Set de Entrenamiento, de forma tal que aquellas que fueron **mal clasificadas pesen más**:



1. **En algunas variaciones al algoritmo**, además de incrementar las ponderaciones de las observaciones mal clasificadas, es posible también **reducir las ponderaciones** de las **observaciones bien clasificadas** y **normalizar los pesos** una vez hecho todo esto.
2. El **resultado final** es **G(x) donde cada Gm(x) hace su aporte** con su **voto ponderado** por su coeficiente **alpha\_m** (Calculado en el paso 5).

**En Python:**

Usaremos el dataset Hitters.csv, para entrenar un **modelo de regresión** con **AdaboostRegressor** para **predecir** el valor de **log(Salary).** Como **modelo base**, usaremos una **regresión linear.**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

form sklearn.ensemble import AdaBoostRegressor

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

data\_raw = pd.read\_csv(’../Data/Hitters.csv’)

print(data\_raw.shape)

data\_complete = data\_raw.dropna()

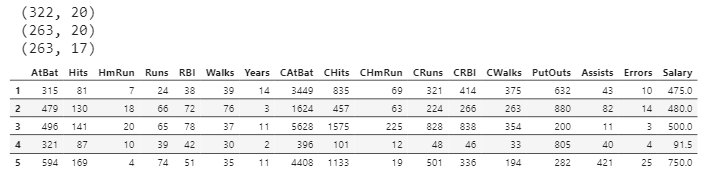
print(data\_complete.shape)

data\_columns = [‘AtBat’, ‘Hits’, ‘HmRun’, ‘Runs’, ‘RBI’, ‘Walks’, ‘Years’, ‘CAtBat’, ‘CHits’, ‘CHmRun’, ‘CRuns’, ‘CRBI’, ‘CWalks’, ‘PutOuts’, ‘Assists’, ‘Errors’, ‘Salary’]

data = data\_complete.loc[:, data\_columns]

print(data.shape)

data.head()



X = data.drop(‘Salary’, axis = 1)

print(X.shape)

y = np.log(data.Salary)

print(y.shape)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state = 127)



scaler = StandardScaler()

X\_train\_scl = scaler.fit\_transform(X\_train)

X\_test\_scl = scaler.transform(X\_test)

base\_regressor = LinearRegression()

bost\_linreg = AdaBoostRegressor(base\_estimator = base\_regressor, n\_estimators = 200, learning\_rate = 0.8, loss = ‘linear’, random\_state = 127)

bost\_linreg.fit(X\_train\_scl, y\_train)



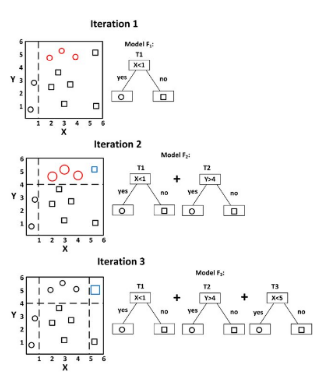
prediction = bost\_linreg.predict(X\_test\_scl)

performance = mean\_squared\_error(y\_test, prediction)

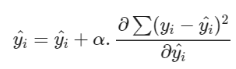
performance

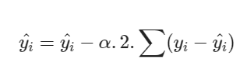


**Gradient Boosting:** Se trata de un **algoritmo greedy** que puede **sobreajustar rápidamente** al **set de entrenamiento.** Es posible mejorarlo con **regularización.** Se trata de una técnica de **aprendizaje supervisado** que interpreta el boosting como un **problema de optimización** a partir de una **función de pérdida.** Se entrenan **modelos débiles en forma secuencial** para ir optimizando dicha función de pérdida. Se distingue de Adaboost en que **no le da más peso a las observaciones con resultados incorrectos**¸ sino que **minimiza la función de pérdida** de los modelos débiles al **promediar las predicciones**.



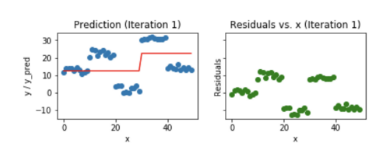
A partir de un **Descenso por el Gradiente**, es posible actualizar las predicciones con una tasa de aprendizaje α de forma tal que la suma de los residuos tienda a cero:



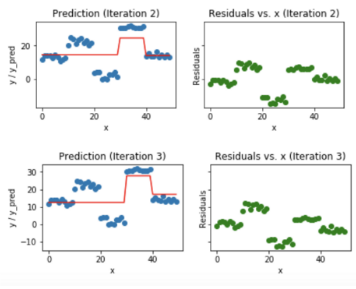


1. Se ajusta un modelo a partir de X features y un vector target objetivo y. Se obtienen las predicciones del modelo y predicted.
2. Se calculan los residuos: 
3. Se ajusta **otro modelo** para los residuos e1. Se usan X como features y e1 como vector objetivo. Se obtienen las predicciones e1 predicted.
4. Se suman los modelos: y predicted2 = y predicted + α \* e1 predicted, siendo α el ***learning rate***.
5. Se ajusta **un tercer modelo** para los residuos e2: 
6. Repetir pasos 2 a 5 hasta que la suma de los residuos sea constante o el modelo empiece a sobreajustar. Con los valores de accuracy en el conjunto de validación es posible controlar el overfitting.

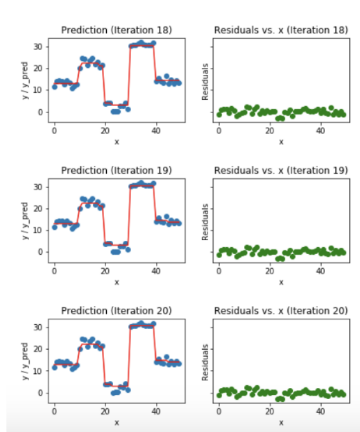
Ahora vamos a ir desmenuzando este proceso paso a paso: Primero se usa un modelo simple para ajustar a los datos y registramos los residuos en un gráfico aparte. Nuestra función de costo buscará minimizar estos residuos:



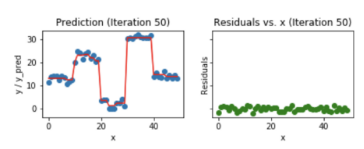
Se van agregando modelos débiles para que se concentren en las áreas donde los modelos que tenemos hasta ahora andan mal. A partir de la 3er iteración, podemos ver que ensamblar modelos débiles empieza a funcionar mejor:



Después de 20 iteraciones, el modelo ajusta tan bien a los datos que los residuos prácticamente tienden a cero:



Siguiendo hasta a 50 iteraciones, los residuos son prácticamente cero:



**En Python:**

from sklearn.ensemble import GradientBoostingRegressor

gb\_reg = GradientBoostingRegressor(los = ‘ls’, learning\_rate = 0.5, n\_estimators = 200, subsample = 1, criterion = ‘mse’, max\_depth = 4, random\_state = 127)

gb\_reg.fit(X\_train\_scl, y\_train)



prediction = gb\_reg.predict(X\_test\_scl)

performance = mean\_squared\_error(y\_test, prediction)



Obtuvimos una performance mucho mejor con GradientBoosting que con Adaboost.

**XGBoost:** **Extreme Gradient Boosting.** Se trata de una **implementación Específica de Gradient Boosting**, donde se usan **aproximaciones más precisas** para **encontrar** el **mejor modelo de árbol.**

Las **mejoras más relevantes** que implementa son:

* Calcula **gradientes de segundo orden**: Segundas derivadas parciales de la Función de Pérdida (similar a Método de Newton). **Esto da más información** sobre la **dirección de los gradientes** y **cómo llegar al mínimo** de la **función de pérdida**.
* **Regularización L1** y **L2**, para mejorar la generalización del Modelo.

**Otras ventajas adicionales** son que el **entrenamiento** **es muy rápido** y **se puede paralelizar / distribuir** entre **clusters**.

Empezó en 2014. En 2015 empieza a aparecer en competencias de Kaggle. Se convierte en el **algoritmo más popular de problemas con datos estructurados.** Lo diseñó **Tianqi Chen**, Universidad de Washington.

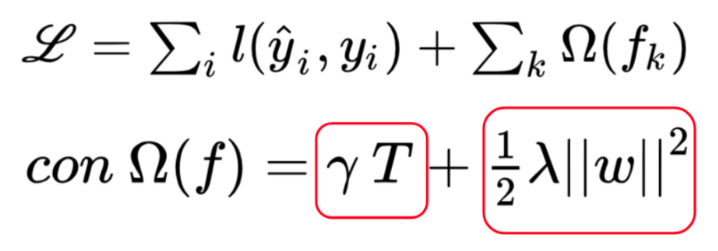
Ventajas con Respecto a Gradient Boosting:

* **Velocidad:** Logró ajustar modelos mucho más rápido que los que tenían las librerías antes de su llegada (Scikit Learn de Python, gbm de R, H2O y Spark MLLib).
* **Escalabilidad**: Como es muy eficiente en el uso de RAM, es posible entrenar usando mucha más información.
* **Rendimiento**: Es posible **buscar hiperparámetros de una manera más amplia** en la **misma cantidad de tiempo** y también **se puede entrenar con más información**. Con el agregado de una **mejora clave en la función objetivo**, como consecuencia este **algoritmo da mejores resultados**.

XGBoost además soporta las 3 principales implementaciones de Gradient Boosting:

1. **Gradient Boosting**
2. **Stochastic Gradient Boosting** (con sub-muestras en filas y columnas por split)
3. **Gradient Boosting con Regularización** (normas L1 y L2)

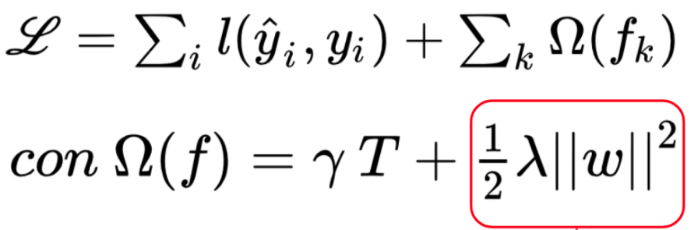
**Gradient Boosting con Regularización:**

****

T es la cantidad de hojas; *Gamma* es cuánto debe reducir la función de pérdida un Split para que se produzca: A mayor gamma, más difícil es realizar un Split, entonces tendremos árboles más simples.

½ λ ||w||2 es la norma L2 de **Ridge.** Los **w** son los **Scores de las hojas**. Acá lo que se está haciendo es penalizar los scores grandes. Esto se hace buscando que ninguna hoja sea particularmente importante.

Es posible implementar la norma L1 en vez de la L2 (**Lasso):**



Y además, es posible incluir funciones de pérdida personalizadas (Σl(i,yi))

XGBoost usa **tres tipos de algoritmos** para **encontrar los splits**:

1. **Exact Greedy:** Prueba todos los posibles puntos de Split y se queda con el mejor. Es un método muy lento.
2. **Aproximado:** Sólo toma los percentiles de cada variable en vez de todos los valores posibles. Recalcula los bins en cada iteración.
3. **Histograma:** Se almacenan los valores de las variables continuas en bins de un histograma. Se reutilizan a lo largo de los cómputos. El Split se hace entre los bins del histograma. Permite entre 2 y 256 bins.

**Mejoras adicionales:**

* **Shrinkage (eta):** Le saca peso a cada nuevo árbol que se agrega al ensamble, multiplicando su influencia por una constante.
* **Sampling de Columnas**:SimilRandom Forest. Usa sampling de columnas para reducir el overfitting y aumentar la variabilidad de los árboles.
* **Sampling de Filas (subsampling)**: Permite definir qué proporción de filas se usa para entrenar cada árbol.
* **Ausencia de datos**:Les asigna una dirección por default en cada nodo. Los va a agrupar a la izquierda o derecha de cada Split en función de cuál sea la dirección que de mejores resultados.

En 2016 **Microsoft** lanza **LightGBM**, que performa igual o mejor que XGBoost ocupando menos memoria. Usa un algoritmo para buscar los splits llamado **GOSS (Gradient-Based One-Side Sampling).** Este algoritmo calcula el gradiente para cada observación, selecciona aquellas observaciones con un gradiente elevado y muestrea un porcentaje de observaciones con gradiente bajo. Esto lo hace presuponiendo que como tienen gradiente bajo, estas observaciones ya pueden ser predichas correctamente. Entrena el árbol con estos datos y multiplica por una constante las observaciones muestreadas para evaluar los splits.

En 2017 **Yandex** libera **Catboost**. Se trata de un algoritmo rápido y con excelentes resultados sin necesidad de definirle hiperparámetros. Puede transformar a **one hot encoder** las variables con valores menores a una determinada cantidad. Maneja a las variables categóricas de una manera especial, usando distintas métricas que se aplican en cada Split para analizar la relación entre categoría y label. Detecta el overfitting rápido y permite detener el proceso cuando esto ocurre.

**Conclusiones:**

* En general con **Boosting** se obtienen **mejores resultados** que con **Bagging.**
* Se necesita **más tiempo de entrenamiento** con **Boosting** que con **Bagging.**
* Hay **varias implementaciones de Gradient Boosting** que **mejoran los tiempos de entrenamientos** y **disminuyen el costo computacional**.
* Uno de los **principales** **motivos para usar Gradient Boosting**, es que **permite optimizar una función de costo personalizada.** Dandole un sentido más práctico en el mundo real que una función de pérdida.