



# Machine Learning – Ensembling

# Concurso de la tele

En un concurso de la tele tenemos varios participantes. Se trata de acertar preguntas. Cuantas más aciertos, más dinero ganas. Veamos cómo lo hacen los participantes:

								VOTACIÓN
Pregunta 1	1	1	0	1	1	0	0	1
Pregunta 2	0	0	1	1	1	1	0	1
Pregunta 3	0	0	0	0	0	0	0	0
Pregunta 4	0	1	1	0	1	1	1	1
Pregunta 5	1	0	1	0	0	1	1	1
% ACIERTOS	40%	40%	60%	40%	60%	60%	40%	80%

¿Resultado? 7 concursantes trabajan mejor en equipo que de manera individual.  
Este mismo comportamiento lo podemos extrapolar a los ensemblings

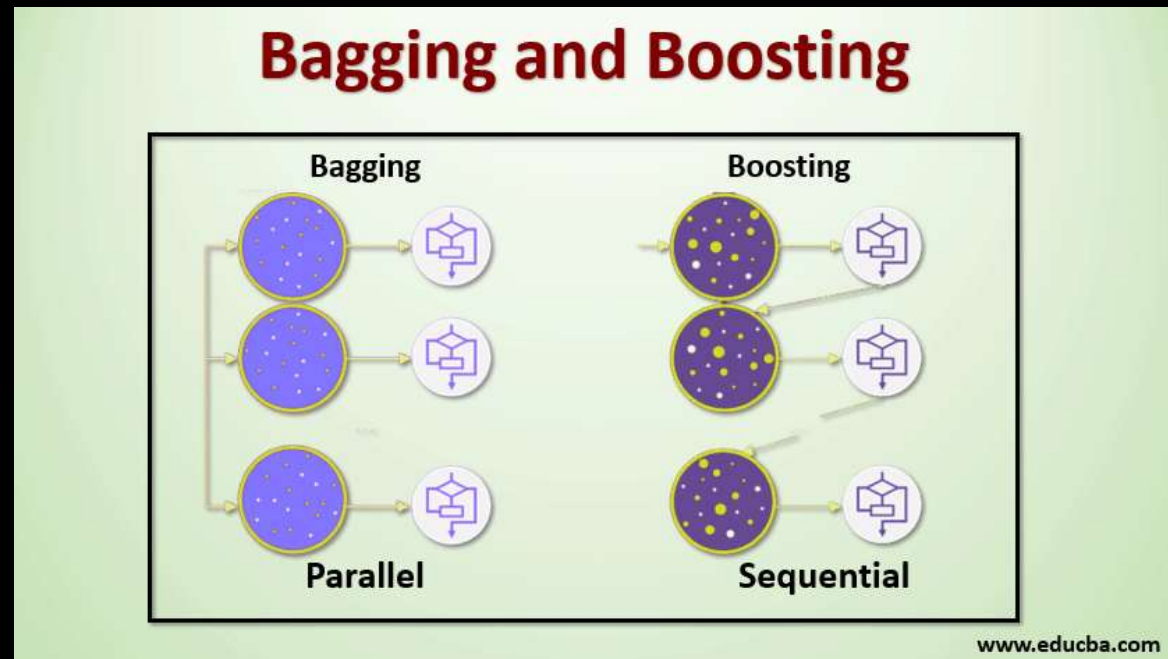
# Definición

Los modelos ensamblados (ensemble models) combinan las decisiones de múltiples modelos para mejorar su precisión y estabilidad.

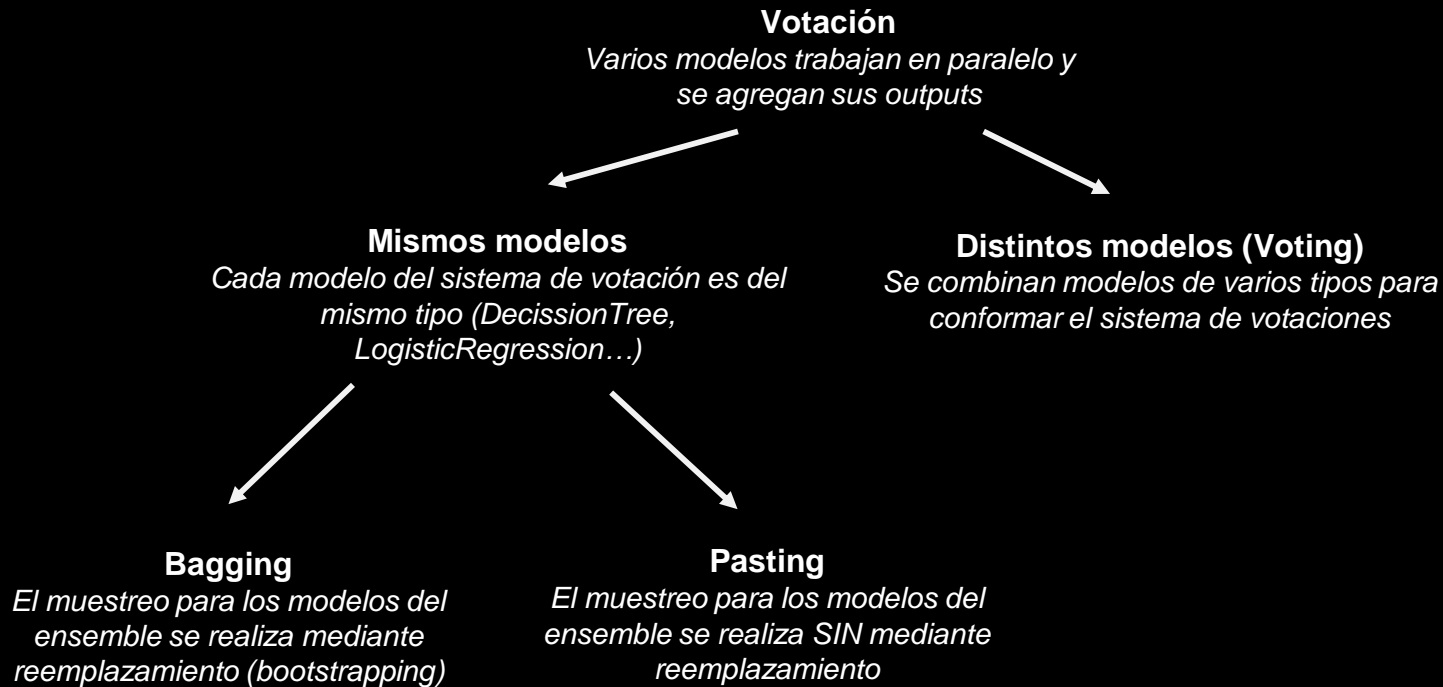
Se trata de modelos que se comportan muy bien y reducen bastante el variance. Este tipo de modelos son los que se suelen utilizar para ganar competiciones de Kaggle

Tipos de ensembles:

1. Bagging
  - a. Random Forest
2. Boosting
  - a. AdaBoost
  - b. GradientBoost
  - c. XGBoost



# Tipos de Ensembles

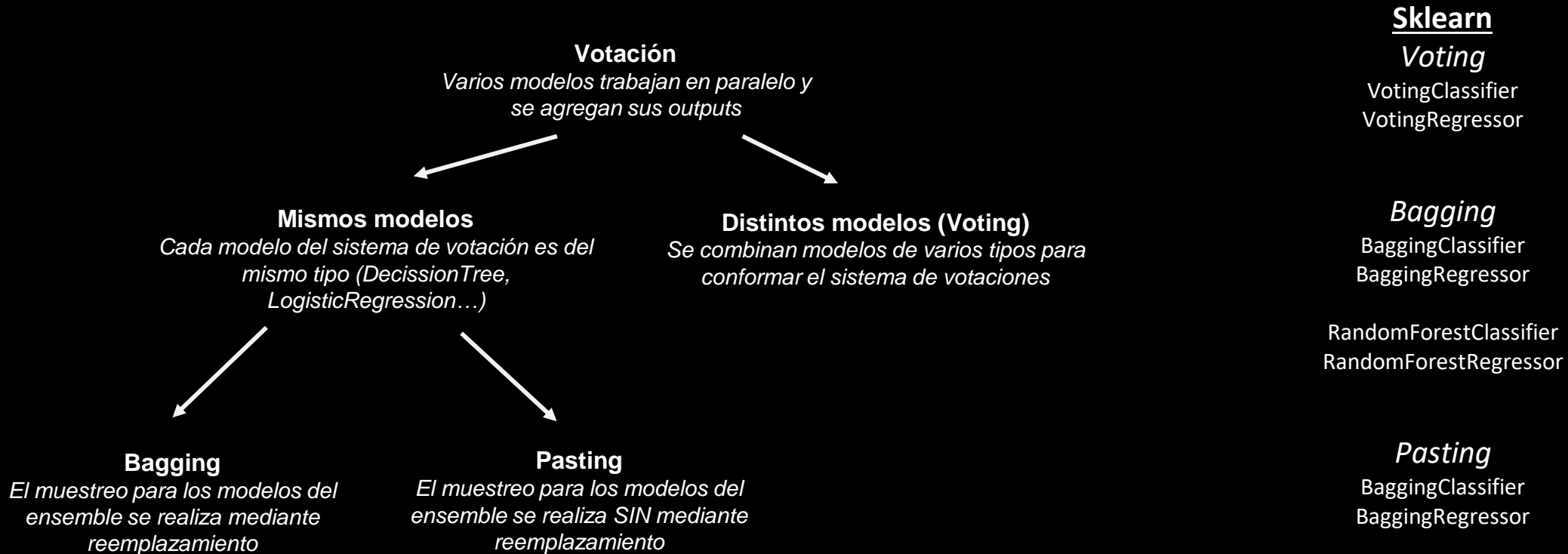


**Secuenciales o Boosting**  
*El output del primer modelo sirve de input para el siguiente, que corregirá los errores del anterior.*

Para todos los modelos vistos en esta presentación tenemos las versiones de clasificación y regresión en sklearn

Votación

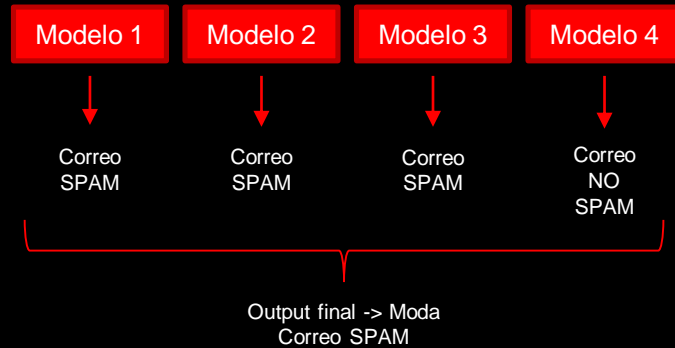
# Algoritmos de votación



# ¿Cómo se realizan las predicciones?

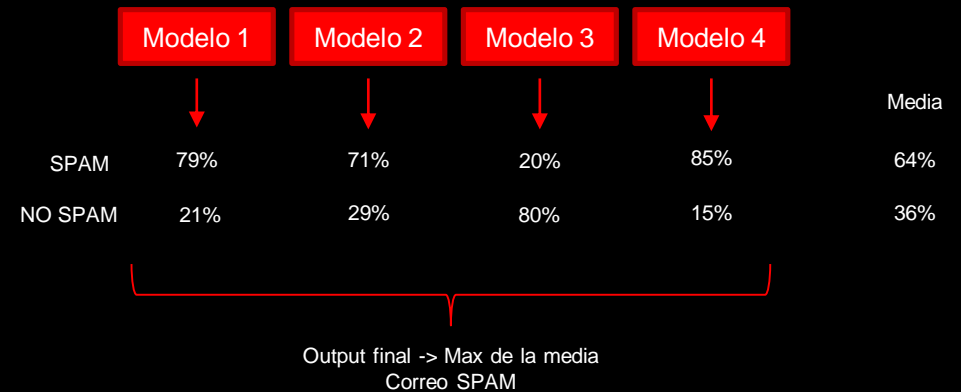
## Clasificación – Hard voting

Se calcula la moda



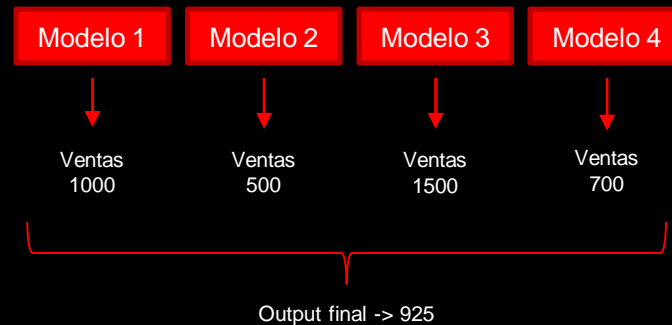
## Clasificación – Soft Voting

La clase que tenga la probabilidad más alta dentro de los outputs de todos los modelos

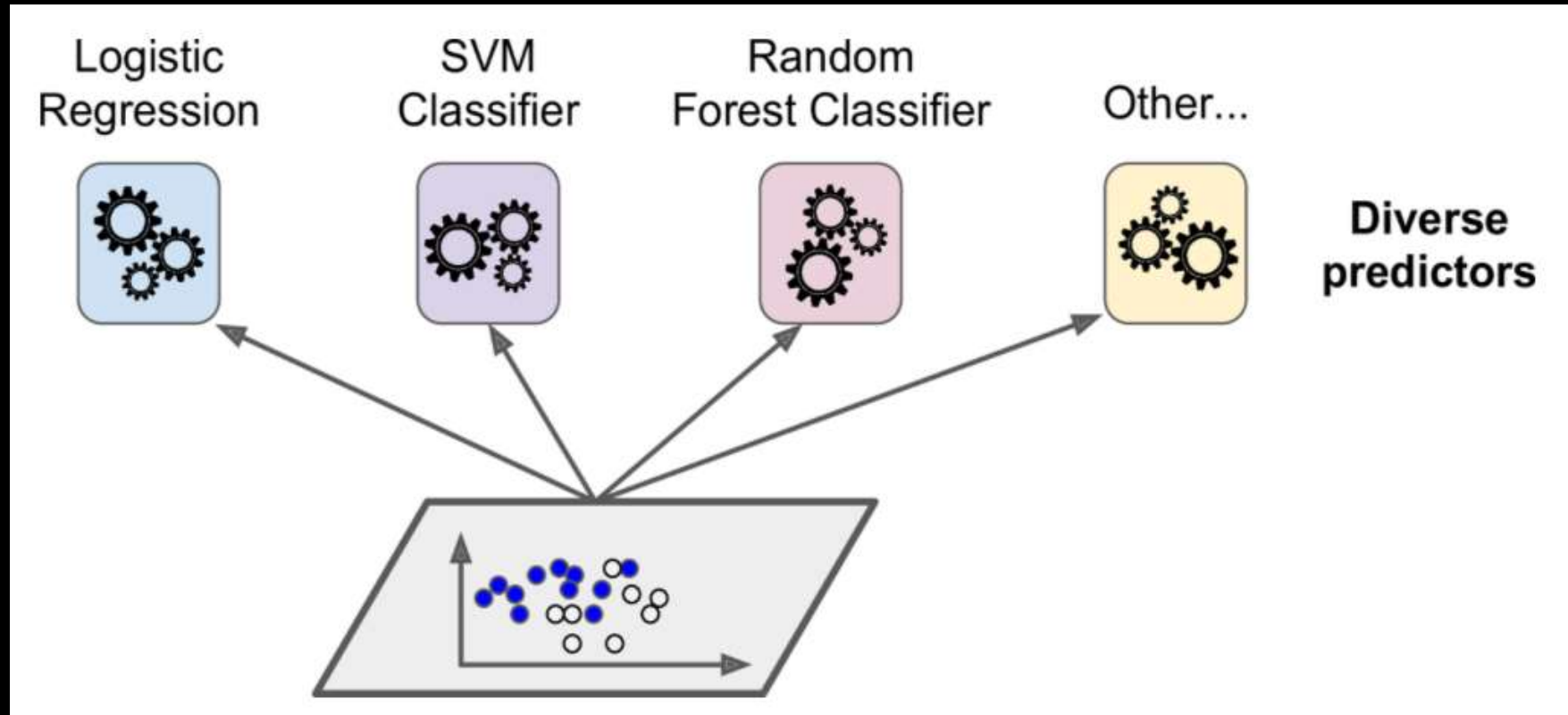


## Regresión

Se calcula la media de todos los outputs



# Voting





# Bagging y Pasting

# Bagging (Bootstrap Aggregating)

Con esta técnica se entrenan un conjunto de modelos, mediante muestras **con reemplazamiento**. Para cada predicción todos los modelos dan un output, y como si de un sistema de votación se tratase, se escoge como output final el más frecuente.

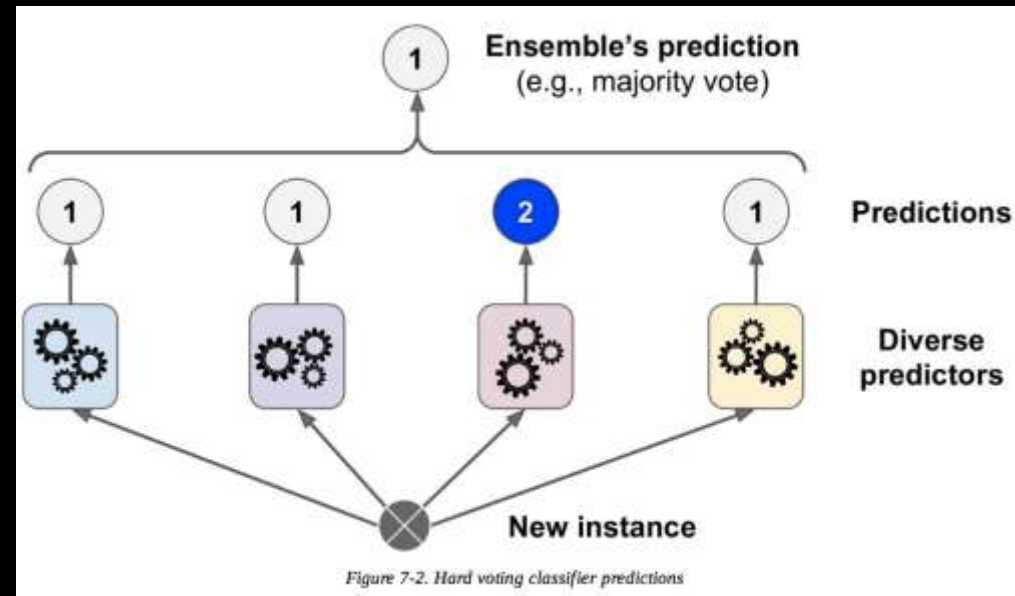
Esta técnica se usa tanto en **clasificación**, como en **regresión**. Si en clasificación utiliza la moda para elegir output, en regresión es la media de los outputs de todos los modelos.

Se trabaja con el mismo modelo

## Pasting

Simplemente escogemos el argumento  
“bootstrap=False”

Suele funcionar mejor bagging



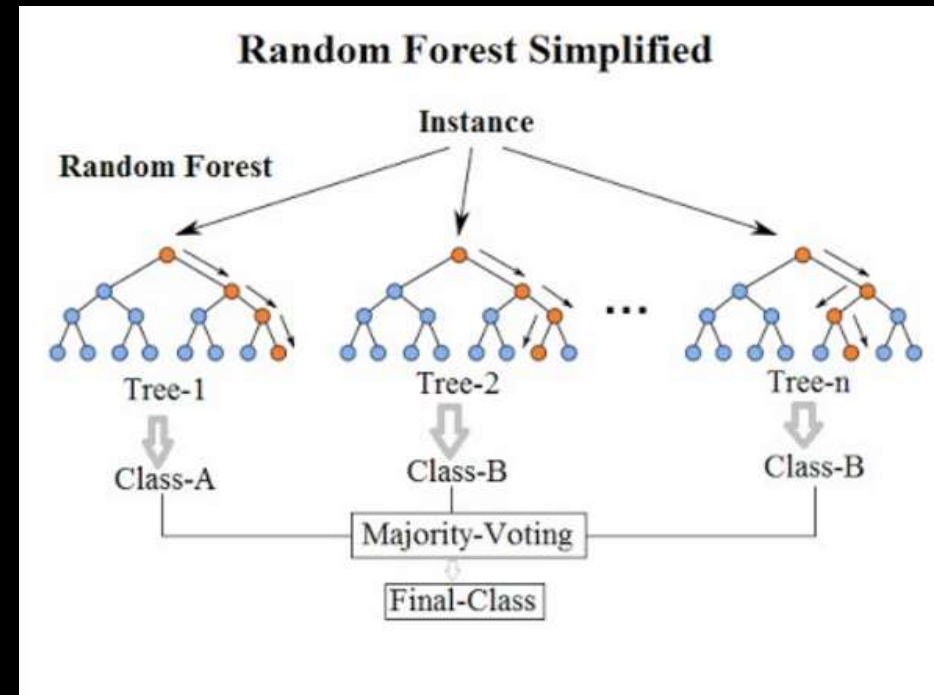
# Random Forest Demo

<https://aternova.github.io/random-forest-viz/>

# Random Forest

El algoritmo de bagging que más se utiliza es el random forest. Implementa el sistema de votación de bagging mediante árboles de decisión. **RandomForestClassifier** y **RandomForestRegressor**. Funciona de la siguiente manera:

1. Escogemos una cantidad de árboles que entrenaremos.
2. Cada árbol escoge un conjunto aleatorio de features para realizar cada split. Este número lo podemos configurar en sklearn.
3. Aplicamos bootstrapping, es decir, cada árbol entrena con una muestra aleatoria con reemplazamiento del conjunto de train.
4. Una vez entrenados los árboles, aplicamos el sistema de votación de bagging para las predicciones.

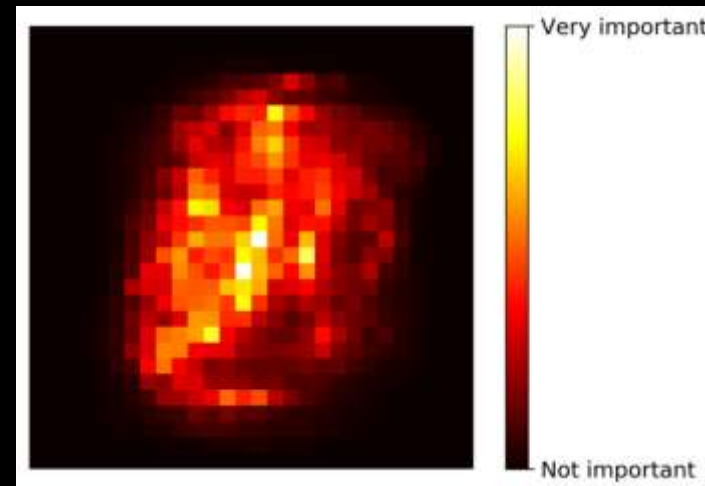


# Feature importance

Una característica interesante que tiene Random Forest es el feature importance. Nos da una medida de cuánto aporta cada feature a las predicciones. Se calcula para cada feature la aportación que ha hecho al gini o entropía. Para todos los árboles. Calculamos la media de la aportación de cada feature.

Por suerte sklearn ya realiza esta operación por nosotros, y lo normaliza a 1, de tal manera que las features más importantes estarán cercanas a 1.

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```



Feature importance para predicción de números

# Boosting

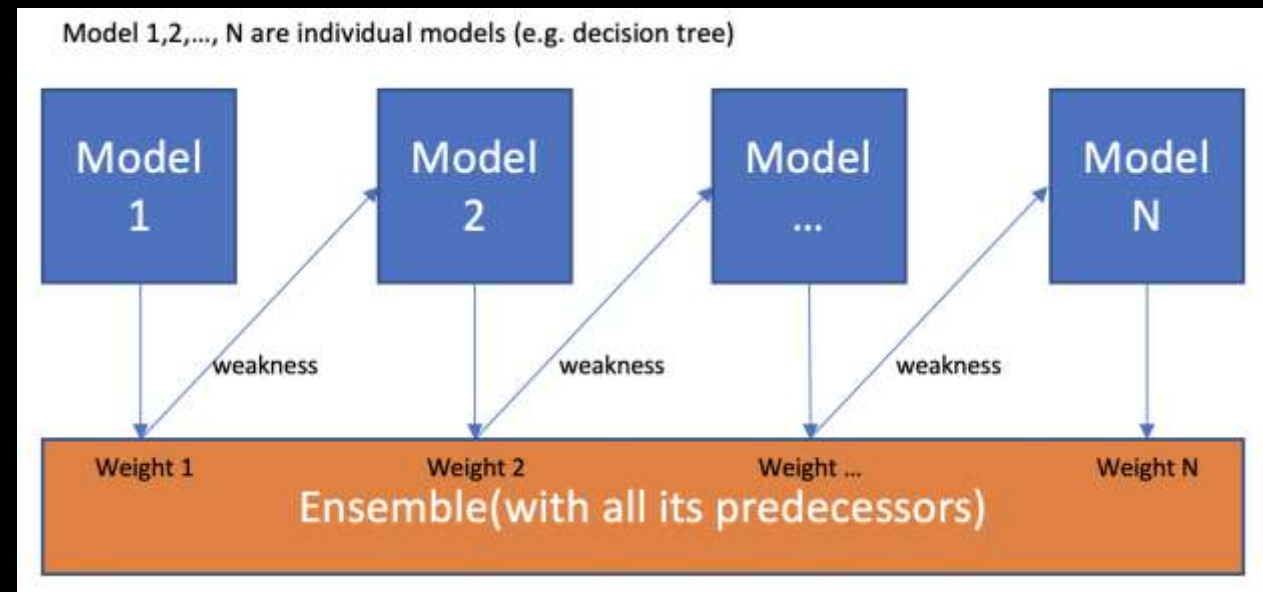
# Boosting

En el caso del bagging, teníamos un conjunto de modelos independientes, cuyos outputs agregados servían para el output final. En boosting los modelos se entrenan secuencialmente y por tanto existe una dependencia entre ellos.

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

Los algoritmos más utilizados son:

1. AdaBoost
2. Gradient Boosting
3. XGBoost

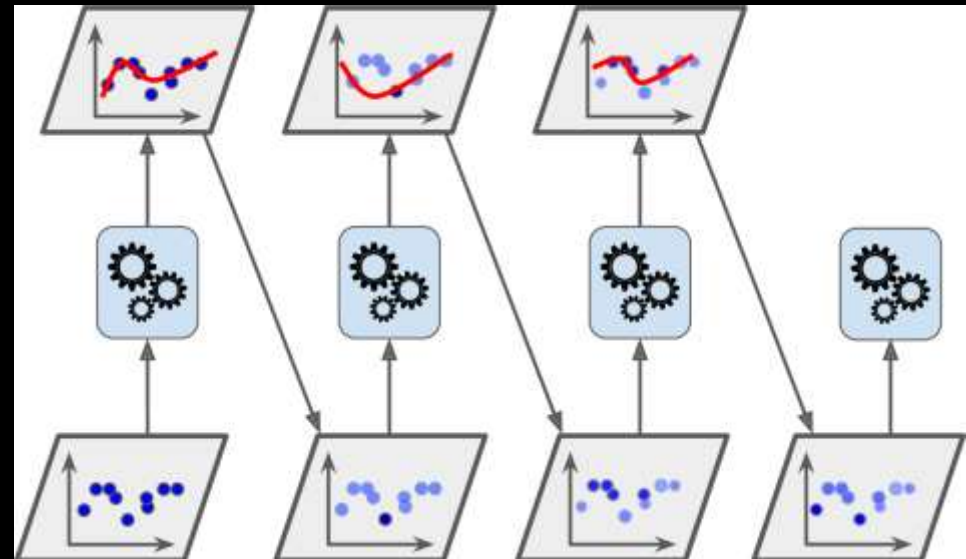
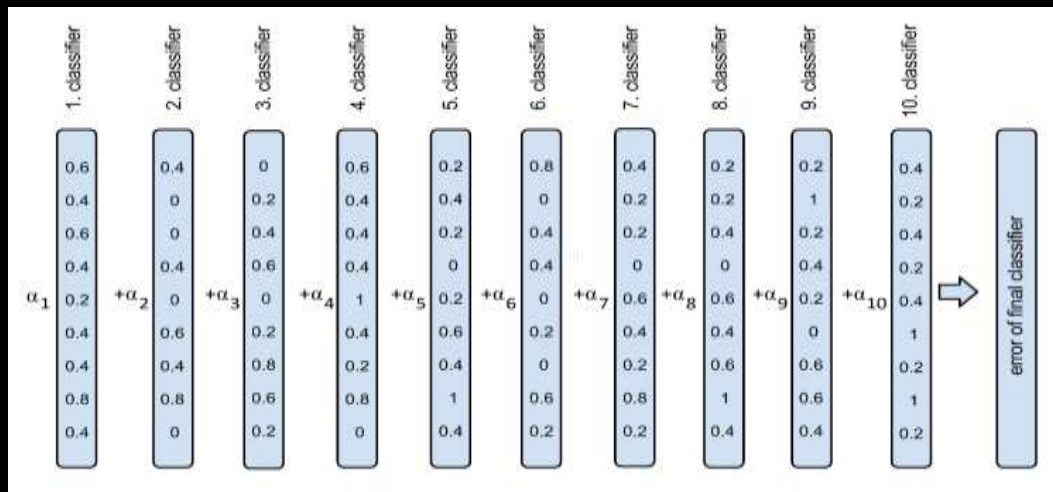


# AdaBoost (Adaptive Boosting)

Se trata de un conjunto de modelos iguales (árboles de decisión normalmente) que actúan de manera secuencial. Las predicciones (junto con sus errores) del modelo predecesor sirven de input para el siguiente, de tal manera que se intenta corregir el error del modelo. Se pone foco en los peores errores.

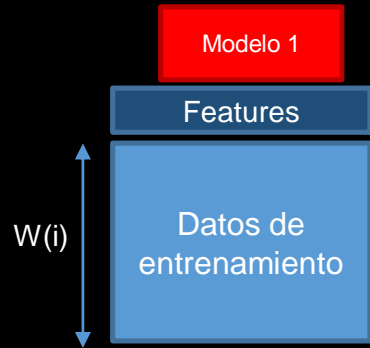
Según lo bien que lo haga cada modelo intentando corregir los errores, se le aplicará un parámetro  $\alpha$  diferente.

Una vez entrenados, el output del modelo final será una combinación lineal de todos los estimadores, teniendo en cuenta el peso de cada uno,  $\alpha$ .





# Entrenamiento



1. Ponderamos todas las observaciones. Se irá actualizando con cada modelo. En este punto inicial, todas las observaciones valen por igual:  $1/m$ , siendo  $m$  el número de observaciones.

La ponderación de observaciones se usa en el entrenamiento:  
`.fit(sample_weight)`

$$r_j = \frac{\sum_{i=1}^m w^{(i)} \mathbb{1}_{\hat{y}_j^{(i)} \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

2. Calculamos los errores
3. Calculamos  $r$  (weighted error rate).

*Pesos de fallos/pesos*

*Cuanto más cercano a 0, mejor será. Cercano a 1, peores los errores.*

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

4. Calculamos  $\alpha_1$   
 $r=0,5$  (random)  $\rightarrow \alpha=0$   
 $r=0,9$  (mal estim)  $\rightarrow \alpha=\eta * (-0.95)$   
 $r=0,1$  (buen estim)  $\rightarrow \alpha=\eta * 0.95$

$\alpha$  es el peso que tendrá este estimador. Cuanto más preciso, mayor será su peso en la decisión final.

$\eta$  es un hiperparámetro (learning rate). Lo que aporta cada árbol.

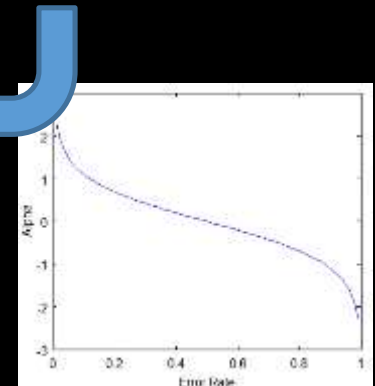
Modelo 2

7. Mismo procedimiento para el segundo estimador

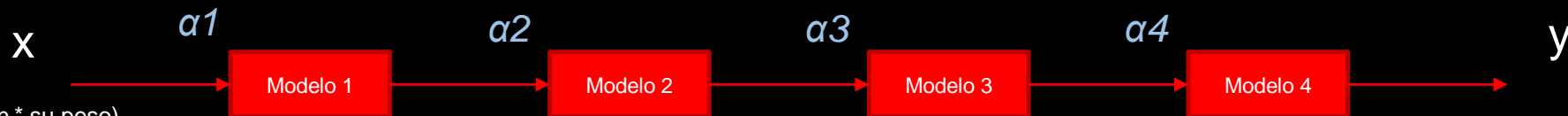
$$\text{for } i = 1, 2, \dots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

5. Se actualizan los pesos de cada observación, si fueron error.
6. Normalizamos el vector de ponderaciones para que su suma sea 1



# Predicción



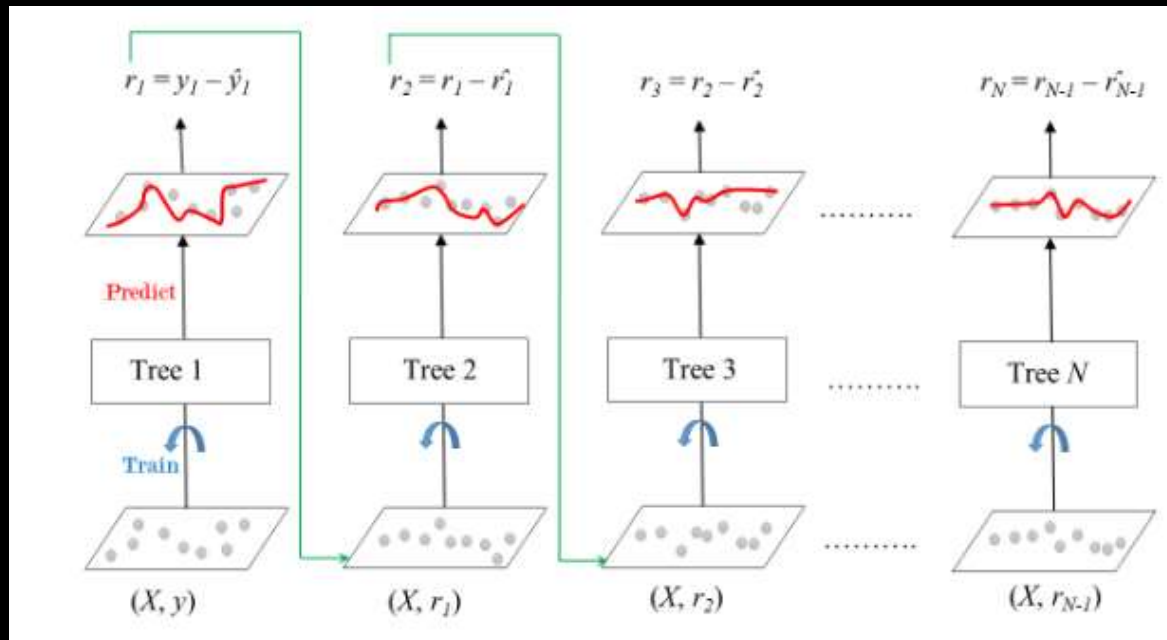
**Class:** suma de (clasificación \* su peso).  
 Clase que esté más cerca de ese valor  
**Regr:** output total.

$$y = \alpha_1 * (\text{Modelo\_1}) + \alpha_2 * (\text{Modelo\_2}) + \alpha_3 * (\text{Modelo\_3}) + \alpha_4 * (\text{Modelo\_4})$$

# GradientBoost

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, minimizar los errores (residuos) del modelo predecesor.

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



Veamos cómo funciona este algoritmo en:

[Hands On Machine Learning](#)

# Bibliografía

Hands on Machine Learning

[https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch07.html#ensembles\\_chapter](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch07.html#ensembles_chapter)

<https://towardsdatascience.com/simple-guide-for-ensemble-learning-methods-d87cc68705a2>

<https://towardsdatascience.com/how-does-xgboost-work-748bc75c58aa>