



# 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 modelos

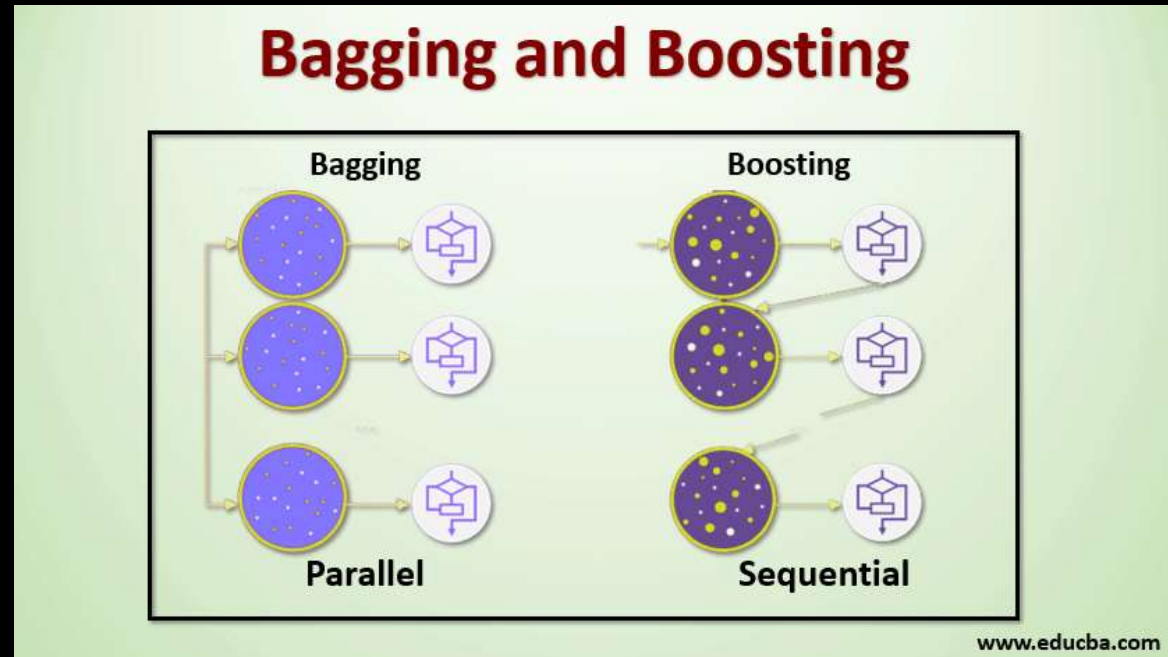
# Definición

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



# Ejemplo

Imagina que quieres comprarte un móvil. ¿Vas a la tienda y simplemente compras el que te recomienda el vendedor?  
NO! Buscas, comparas, ves reviews, preguntas a tus amigos... y dependiendo del output que te den todas esas fuentes, tomas una decisión.

No la tomas a la ligera, sino que tienes en cuenta diferentes fuentes.

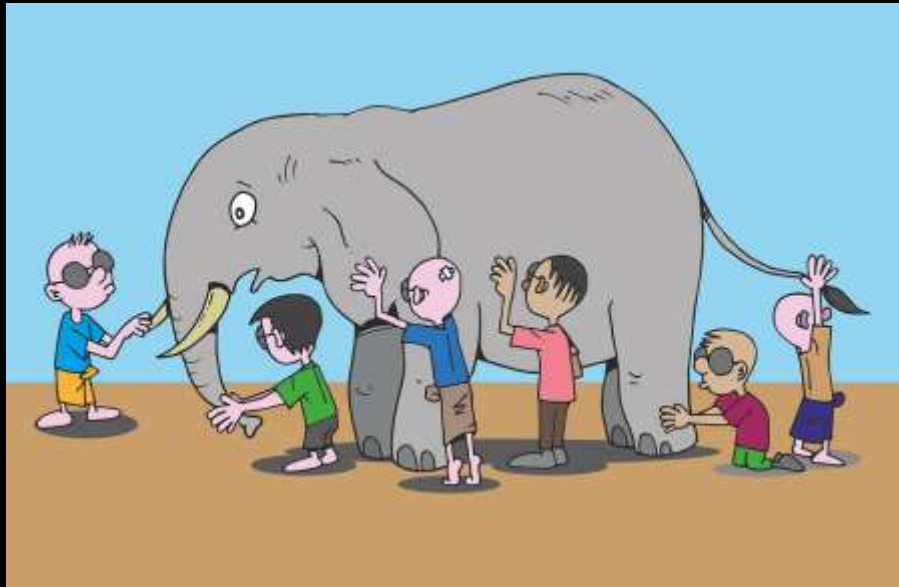


[Fuente](#)

# Ejemplo

Imagina un conjunto de personas ciegas que están intentando describir un elefante. Cada uno tocará una parte del elefante y por tanto una versión diferente.

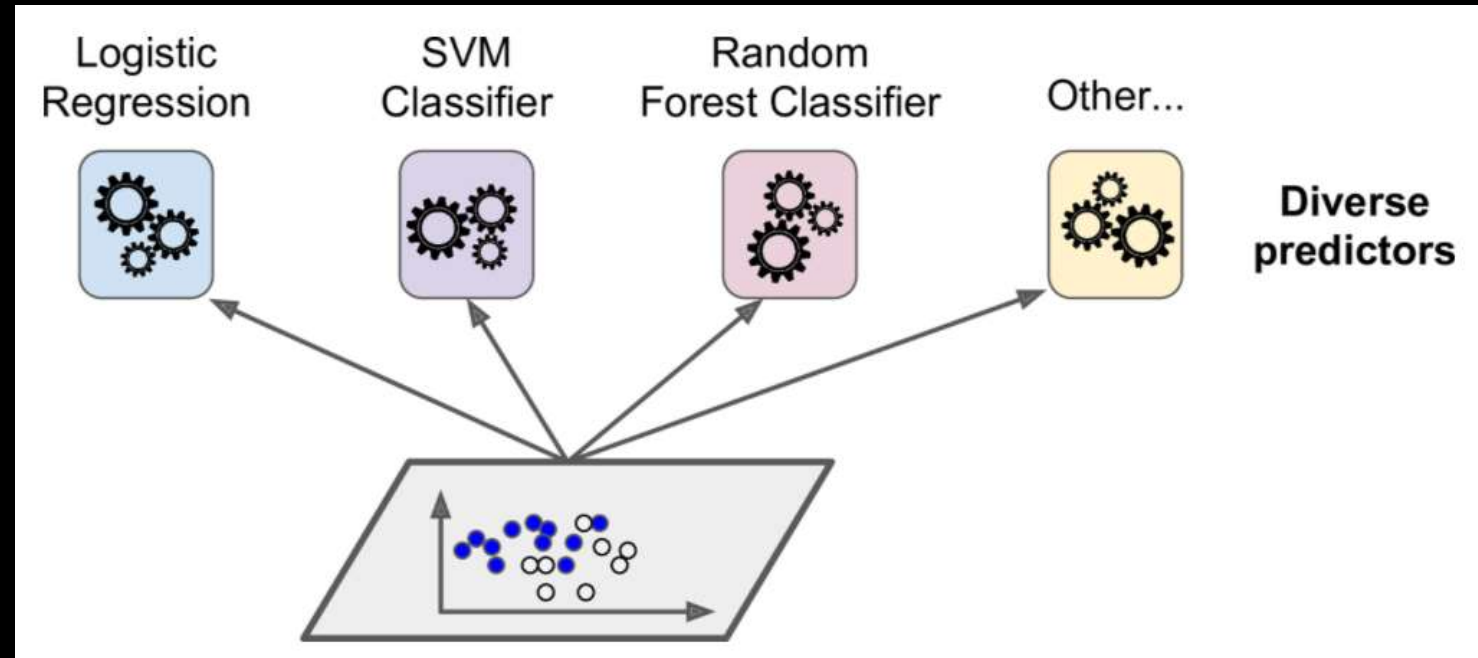
Por tanto, sus versiones individuales describirán partes del elefante, pero su versión colectiva tendrá una información mucho más rica y precisa de cómo es un elefante.



[Fuente](#)

# Voting

Hard voting: los clasificadores votan una respuesta, y la decisión final lo determinará la respuesta más votada.  
Soft voting: está basado en las probabilidades de las respuestas de los clasificadores. Suele funcionar mejor.





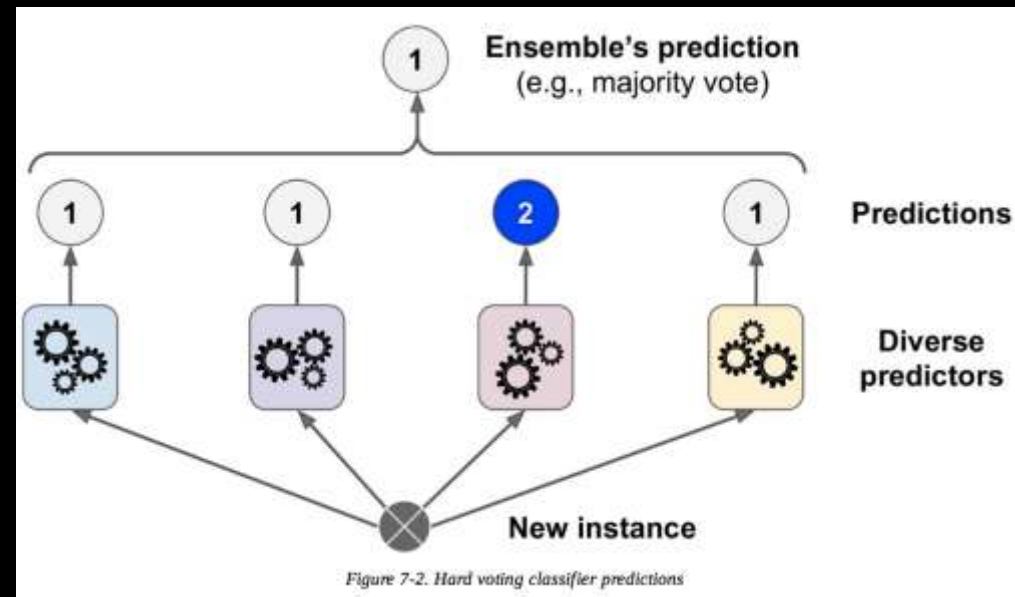
# Bagging (Bootstrap Aggregating)

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. **Utiliza soft voting**

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

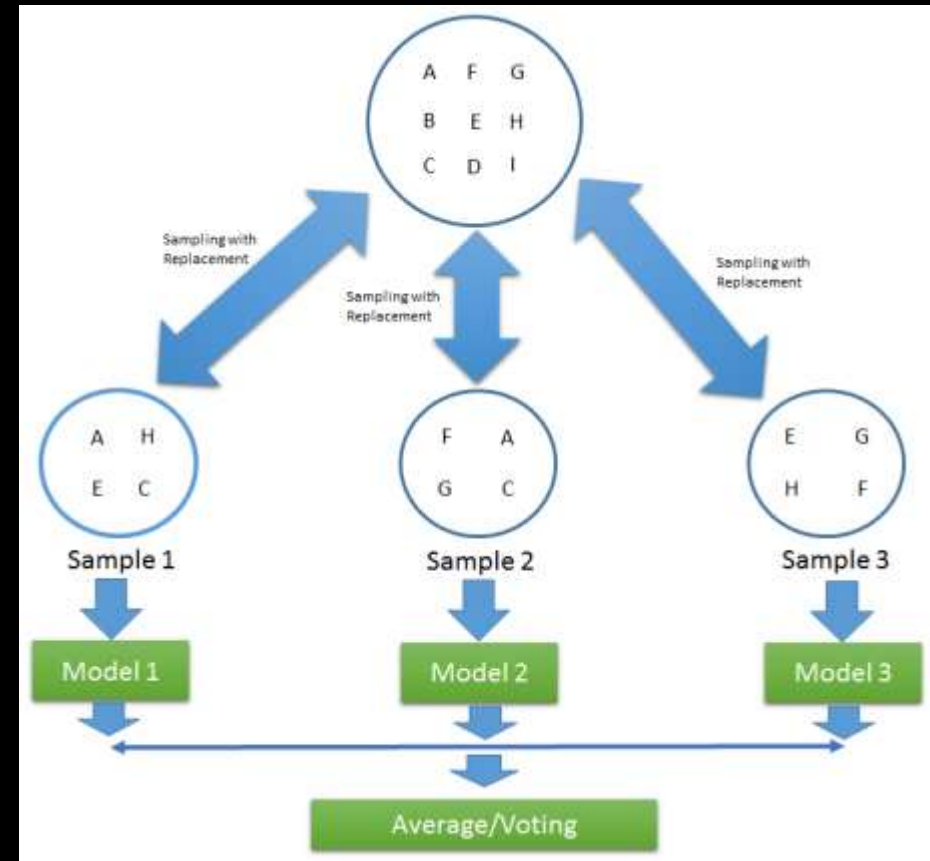
Se puede utilizar esta técnica con muestreos sin reemplazamiento. En este caso estaríamos ante un **pasting**





# Cómo funciona el Bagging

1. Se escoge un modelo. Normalmente árboles de decisión, aunque podría ser un KNN, SVM...
2. Elegimos cuántos modelos queremos entrenar, por ejemplo 10
3. Entrenamos cada modelo con una muestra con reemplazamiento del conjunto de training (bootstrapping).
4. Una vez entrenados los modelos, cuando haya que hacer una predicción, cada uno dará un output. Si es un problema de clasificación, el output final será el valor más frecuente, mientras que si es de regresión se calculará la media de todos los outputs.



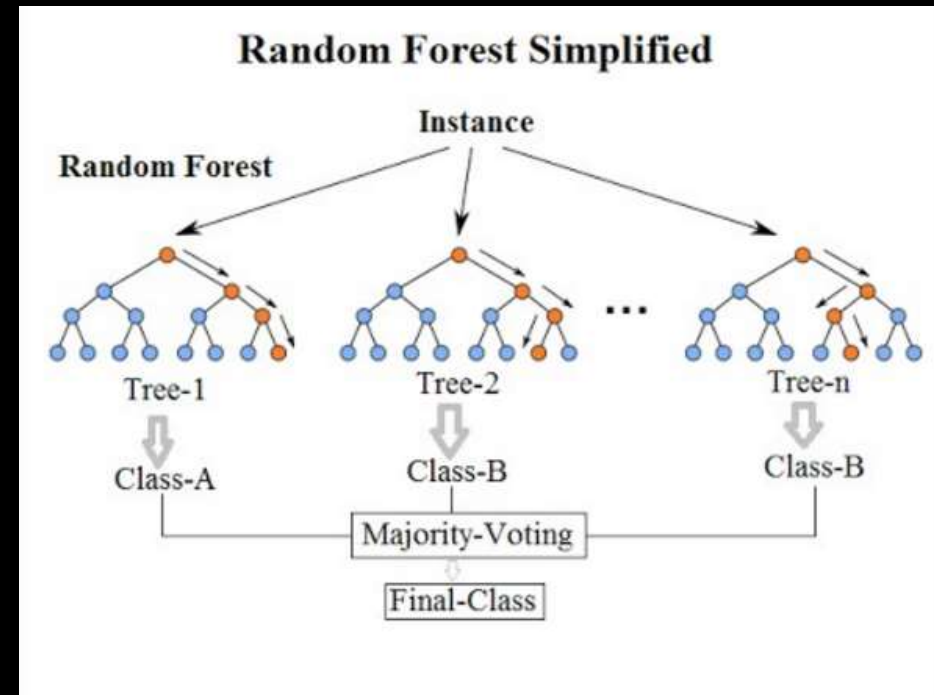
# Random Forest Demo

<https://waternova.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.

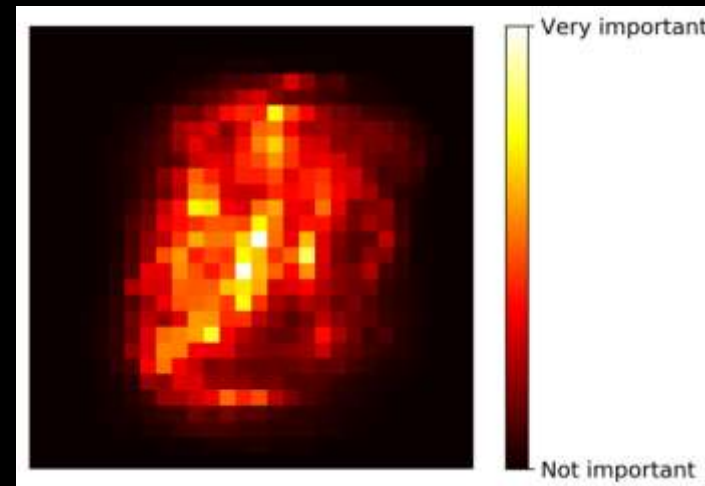


# Feature importance

Una característica interesante que tiene Random Forest es el feature importance. Nos da una medida de cuánto aporta cada feature a las predicciones. Se realiza un cálculo en función de los pesos de cada nodo, y de en cuántas muestras divide el set de train.

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 (el sumatorio de todas no es 1, no es un %).

```
>>> 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

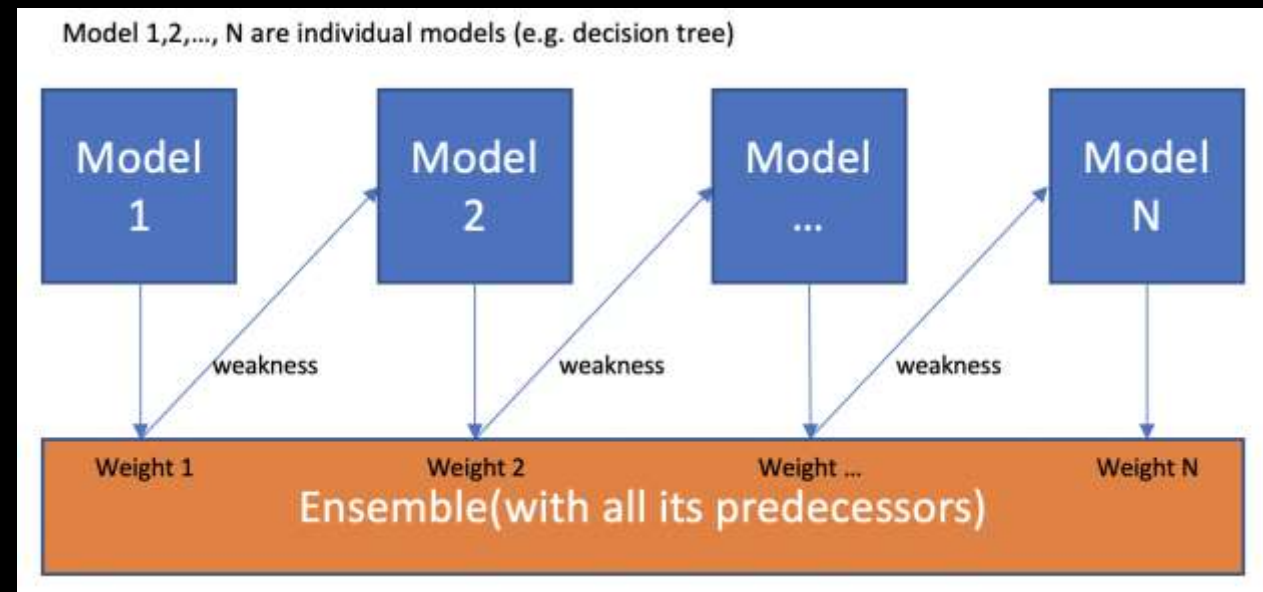
# Boosting

En el caso del bagging teníamos un conjunto de modelos independientes, cuyos outputs servían para el output final. En este caso de 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

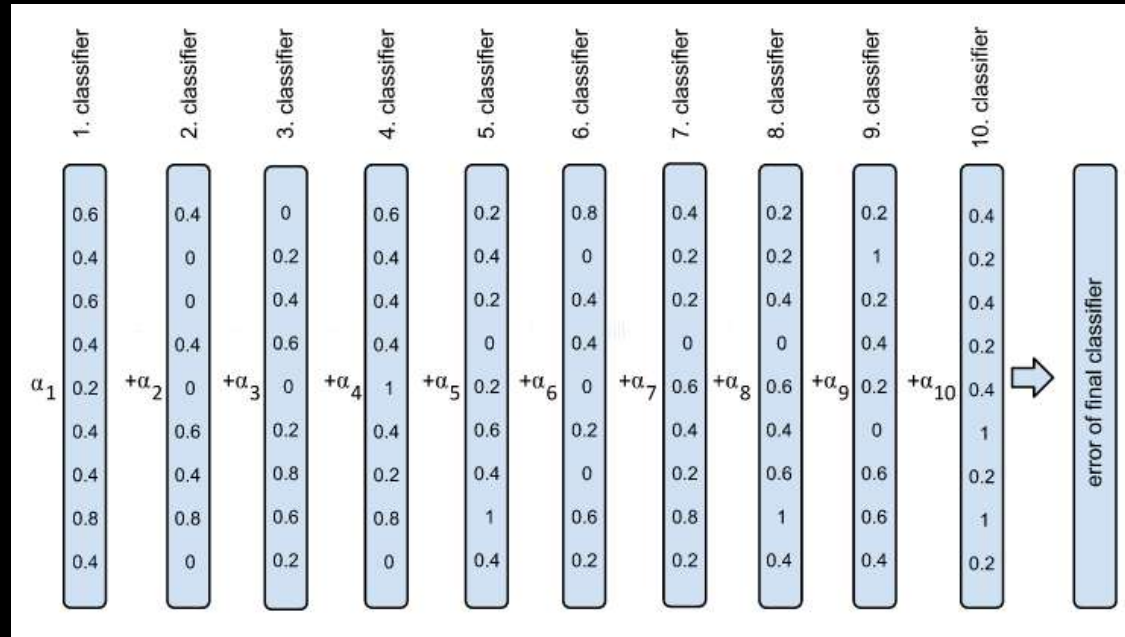


# AdaBoost (Adaptive Boosting)

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,  $\alpha$ . Éste último punto sí se parece más a un bagging que a un boosting.



# Cómo funciona el AdaBoost

1. Ponderamos todas las observaciones a 1. Este vector de ponderaciones se irá actualizando con cada modelo. En este punto inicial, todas las observaciones valen por igual.
2. Entrenamos el modelo.
3. Obtenemos su error de entrenamiento.
4. Calculamos el coeficiente  $\alpha$  en función de sus errores.
5. Actualizamos las ponderaciones (que inicialmente valían 1). Ahora el siguiente modelo no tendrá en cuenta todas las observaciones por igual, sino que hará foco en los mayores errores.
6. Se normaliza el vector de ponderaciones.
7. Continuamos con el siguiente predictor.
8. Acabamos cuando alcanzamos un número máximo de estimadores o el error sea suficientemente bajo.
9. Finalmente tendremos una combinación lineal de todos los modelos:

$$y = \alpha_1(\text{modelo 1}) + \alpha_2(\text{modelo 2}) + \dots + \alpha_n(\text{modelo } n)$$

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

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

$$\begin{aligned} &\text{for } i = 1, 2, \dots, m \\ w^{(i)} &\leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases} \end{aligned}$$

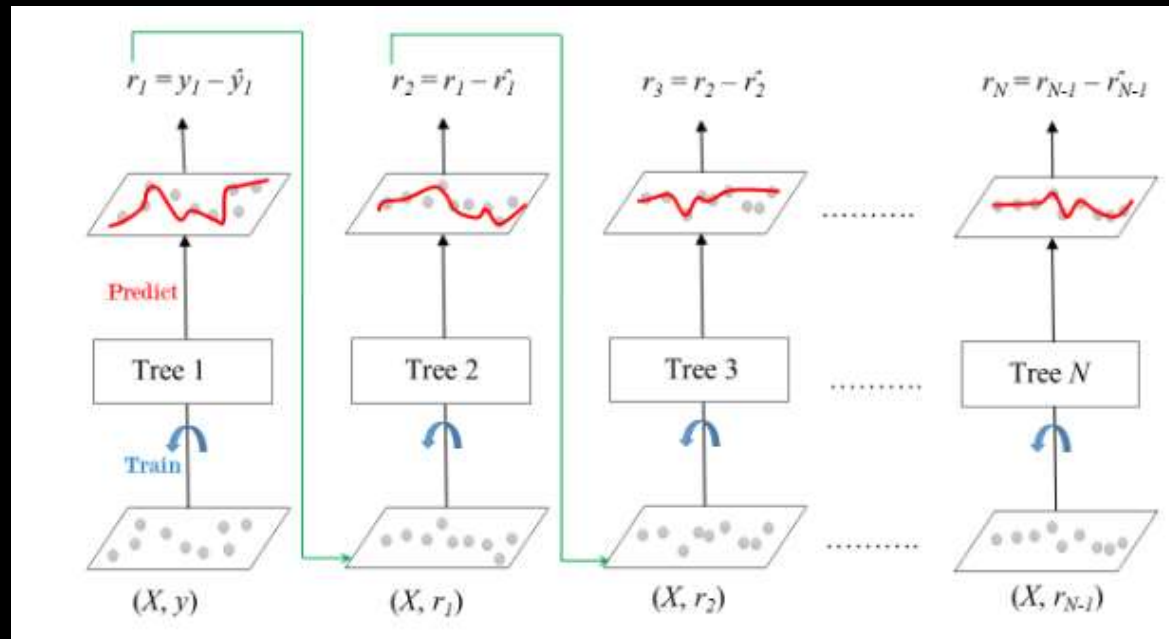
$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^N \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$



# GradientBoost

Al igual que el AdaBoost, el GradientBoost trabaja sobre un conjunto secuencial de modelos, 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](#)