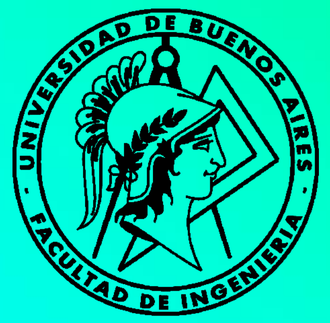


MÉTODOS DE ENSAMBLE

APRENDIZAJE DE MAQUINA I - CEIA - FIUBA

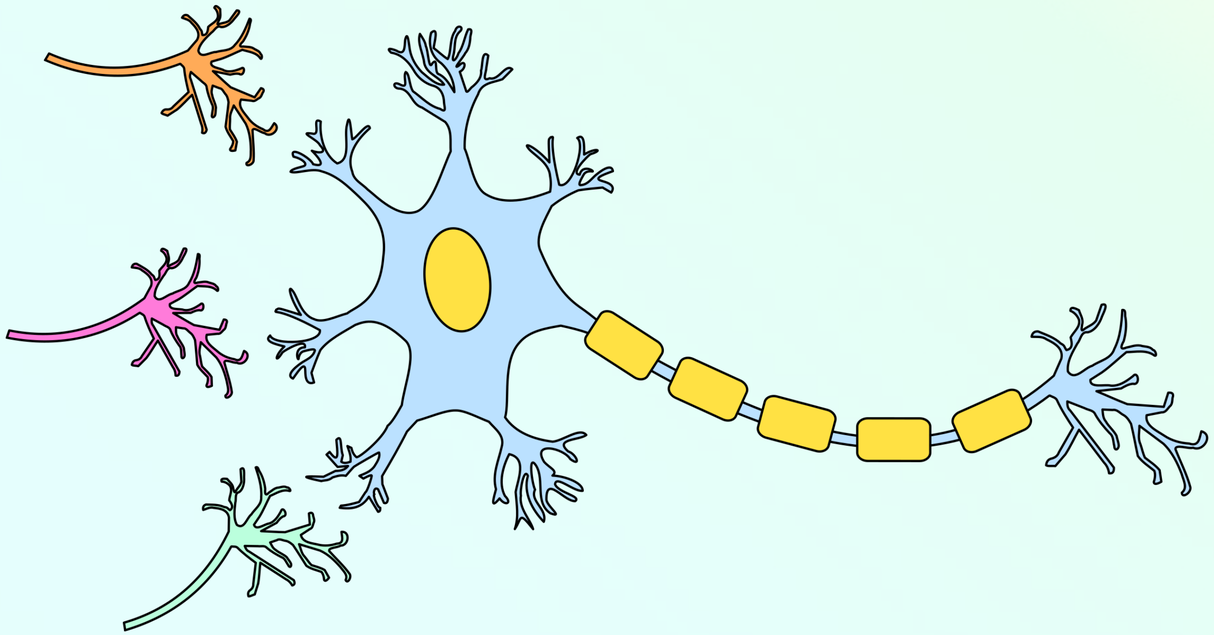
Antonio Zarauz Moreno

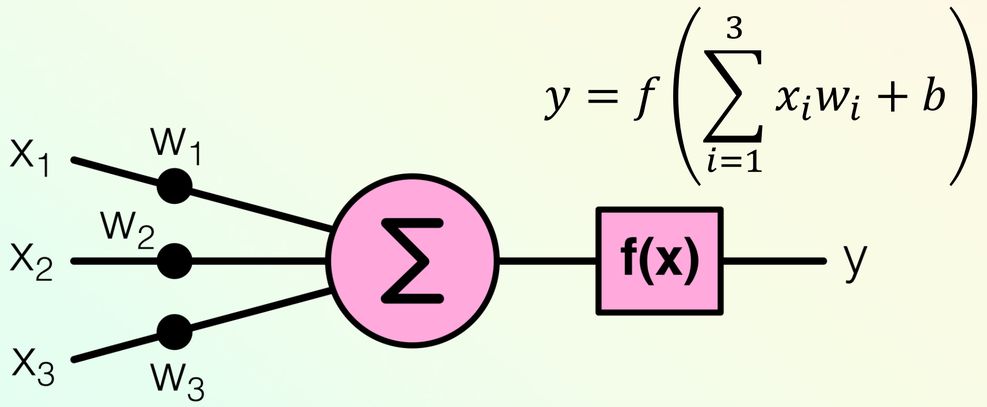
REPASO CLASE ANTERIOR

- Perceptrón y neuronas sigmoideas
- Funciones de activación
- Redes feed-foward
- Breve introducción a Backpropagation
- Pytorch

PERCEPTRÓN

Con esto en mente, armemos el modelo de neurona



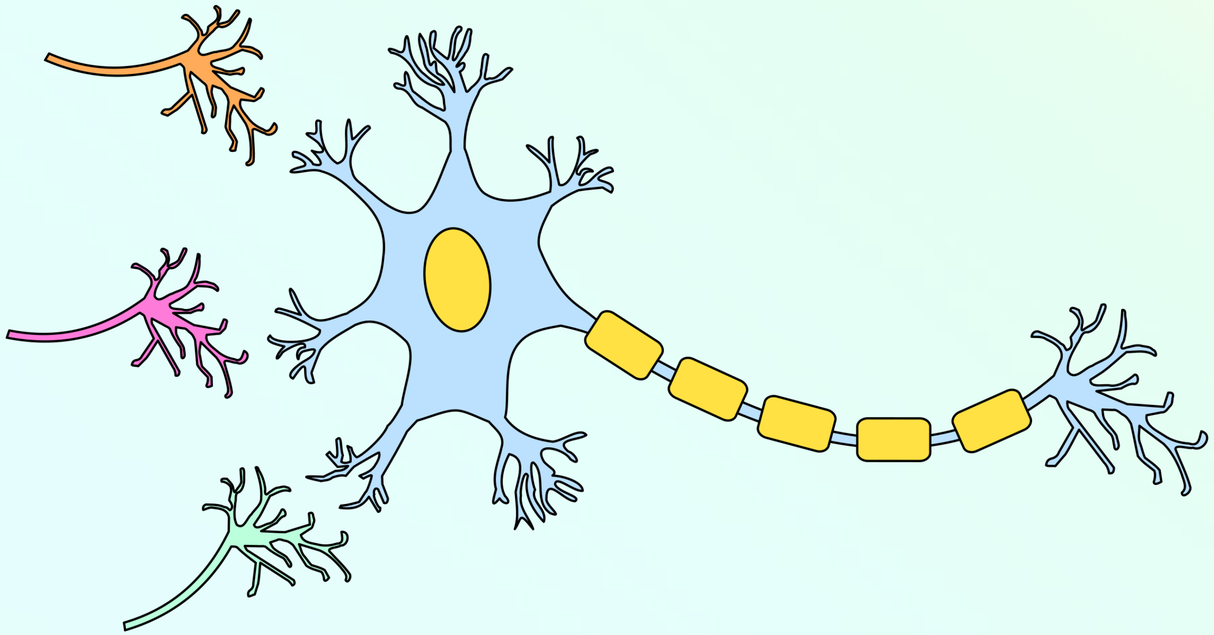


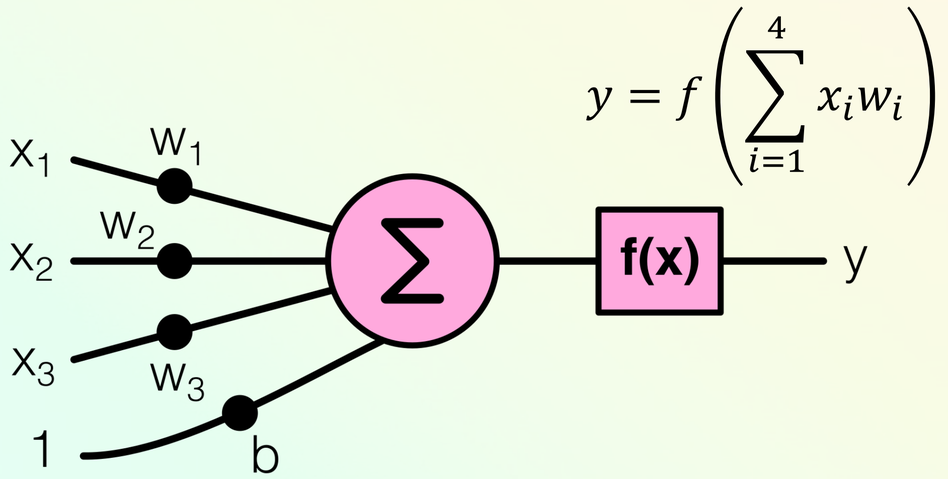
$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

$$\sum_{i=1}^{3} x_i w_i + b$$

PERCEPTRÓN

Con esto en mente, armemos el modelo de neurona





$$x_1 w_1 + x_2 w_2 + x_3 w_3 + 1.b$$

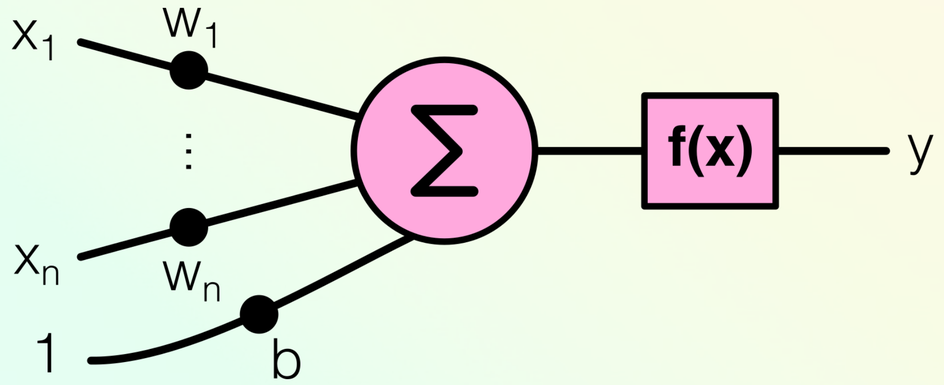
$$\sum_{i=1}^{4} x_i w_i$$

PERCEPTRÓN

Si tenemos una observación de n atributos, la neurona tendrá n entradas con n+1 pesos sinápticos.

Por lo que la parte lineal es:

$$\sum_{i=1}^{n} x_i w_i + b$$

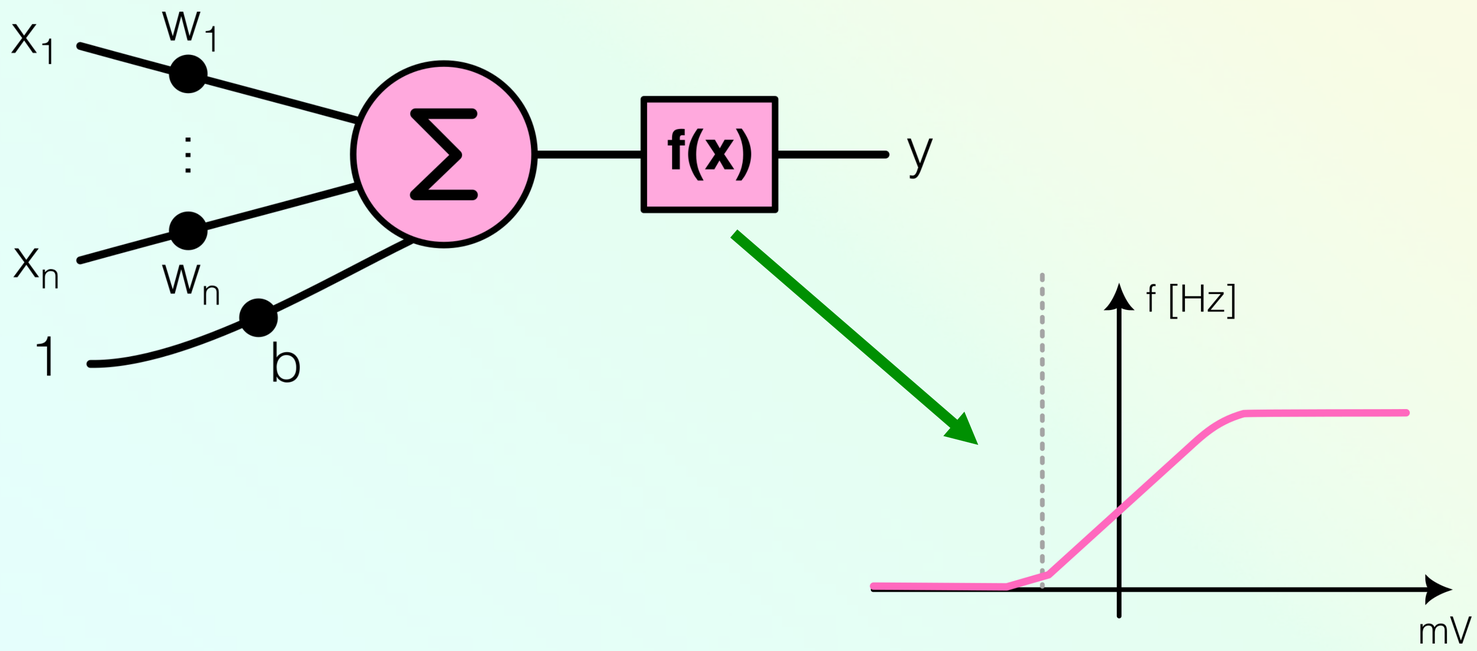


Donde w_i, b son los pesos sinápticos, que representan si la conexión es excitatoria ($w_i > 0$) o inhibitoria ($w_i < 0$), o que no haya conexión sináptica ($w_i = 0$)

PERCEPTRÓN

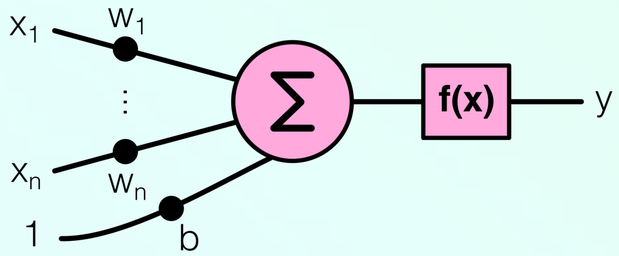
Por lo que la función de activación es una función no lineal que modela la variación de la tasa de disparo de la neurona.

Y con esto último, tenemos la expresión mínima de una neurona, el cual es una unidad de cálculo no lineal.



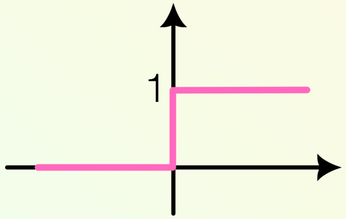
PERCEPTRÓN

La función de activación que se usa en redes neuronales es una decisión de diseño:

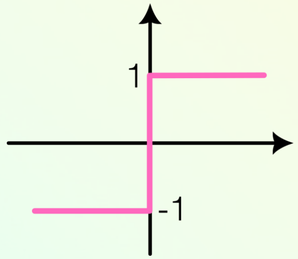


$$y = f\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

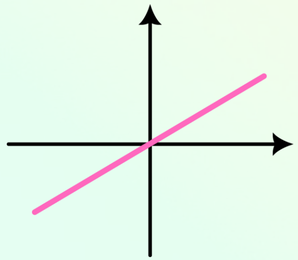
Función escalón $$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



Función signo $$f(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

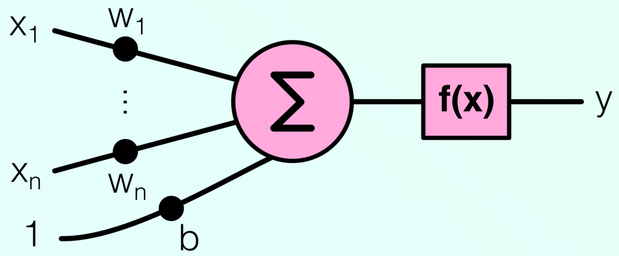


Función lineal $$f(x) = x$$



PERCEPTRÓN

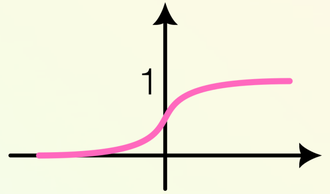
La función de activación que se usa en redes neuronales es una decisión de diseño:



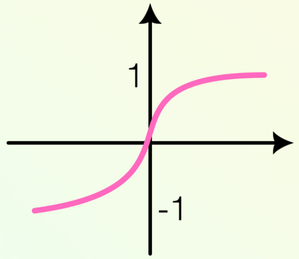
$$y = f\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

Funciones sigmoideas

$$f(x) = \frac{e^x}{1 + e^x}$$

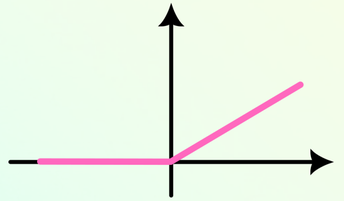


$$f(x) = \tanh(x)$$



Función ReLu

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

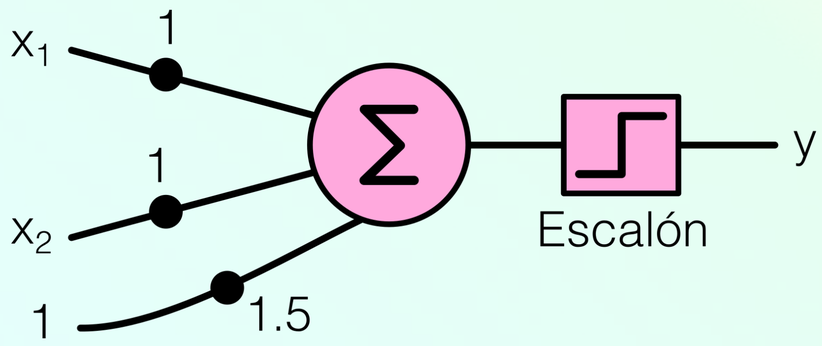


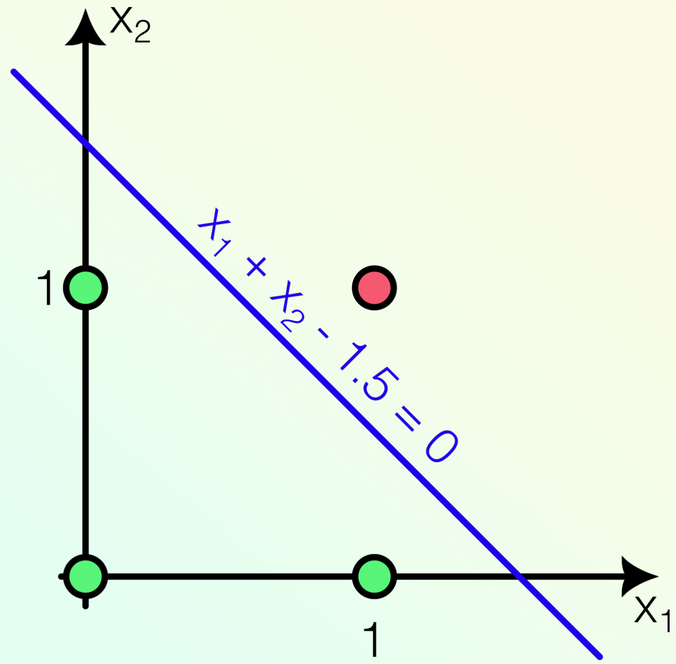
PERCEPTRÓN

Veamos el ejemplo de funciones lógicas:

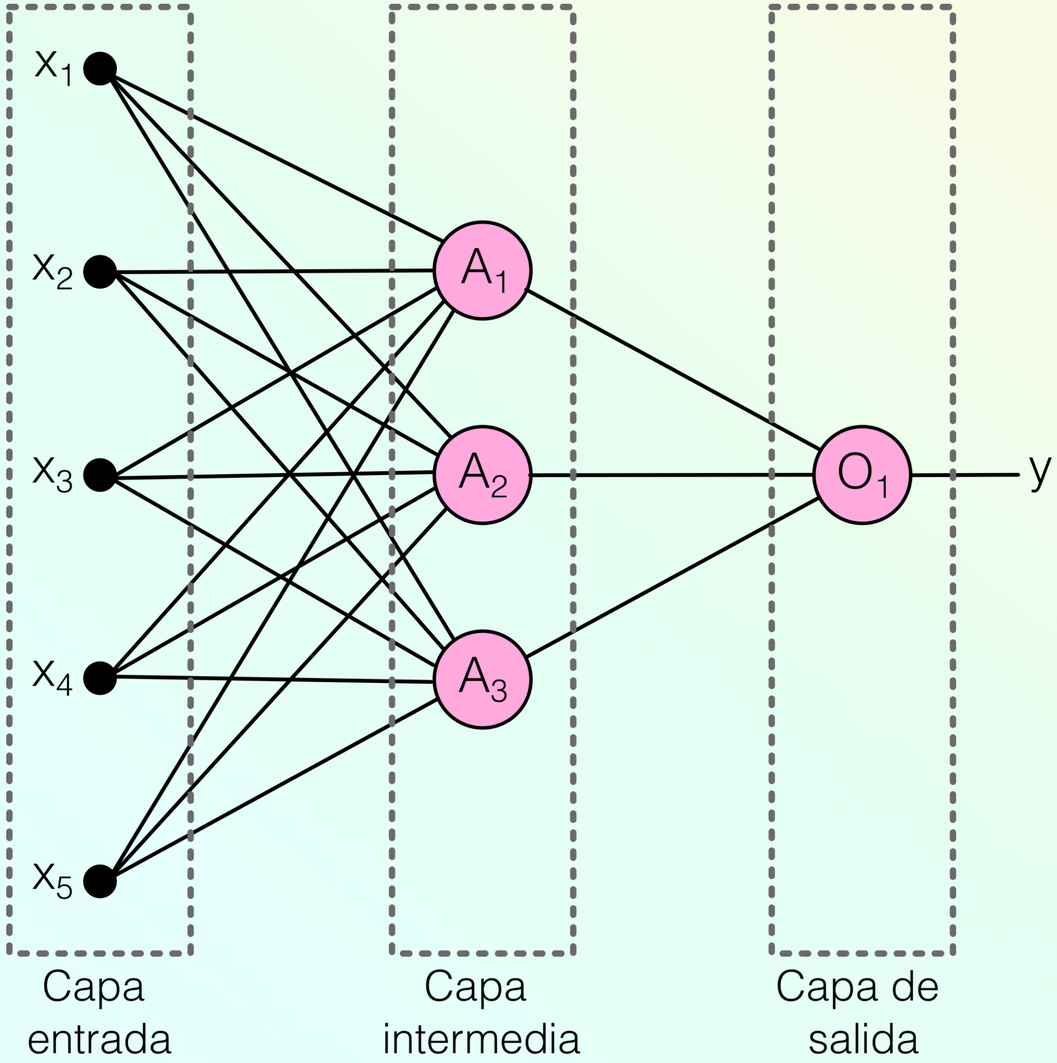
AND (2 entradas)

X_1	X_2	Y
0	0	0
0	1	0
1	0	0
1	1	1

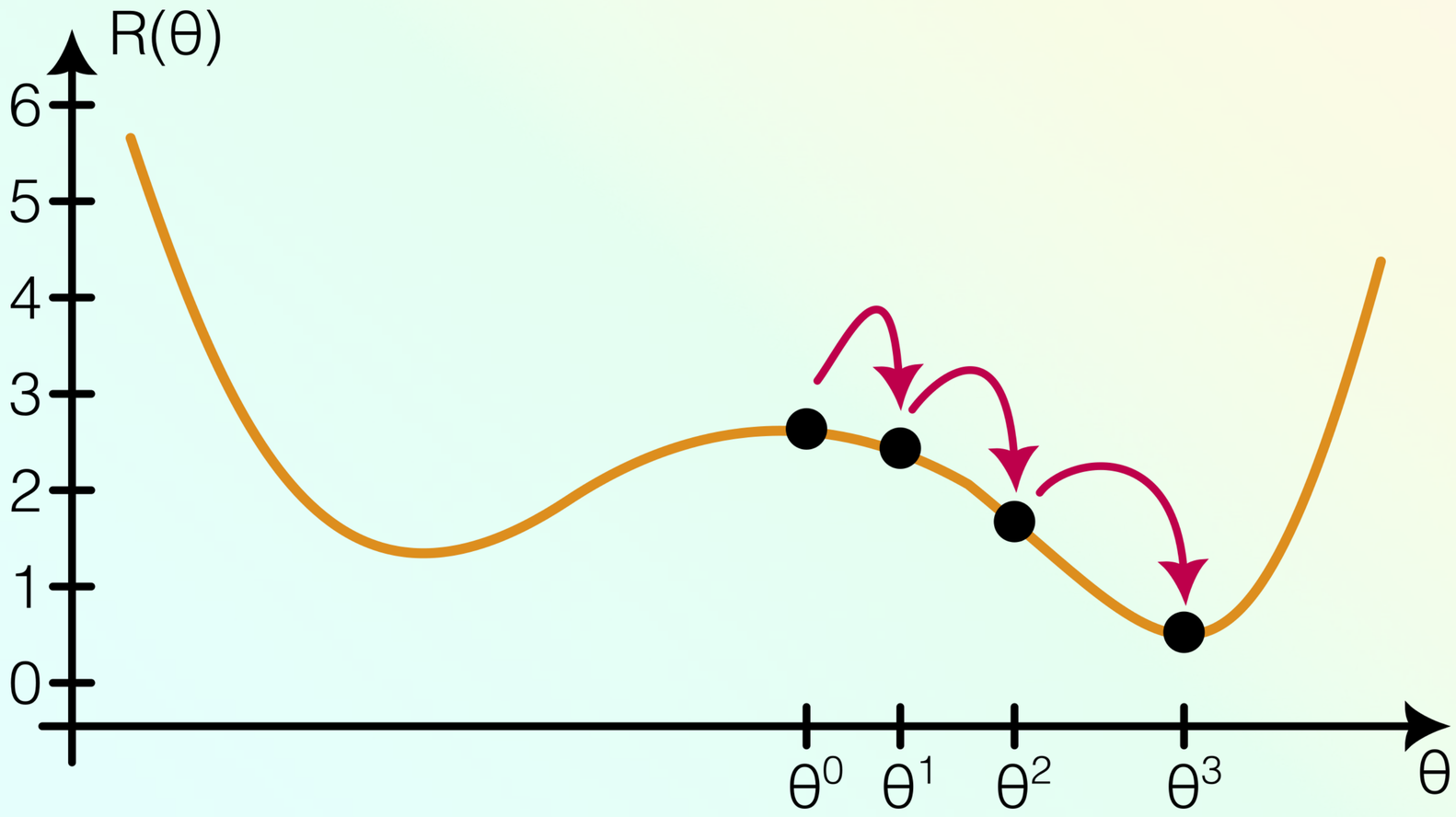




REDES FEED-FORWARD



ENTRENAMIENTO



PYTORCH

PyTorch es una librería de Deep Learning. Es mantenida por Meta y es la principal competencia de otra famosa librería (tensorflow de Google).

Tiene un API para Python, como también para C++. PyTorch nos da toda una infraestructura de:

- Computación de tensores. Es similar a los arrays de Numpy pero con posibilidad de usarlos en GPU.
- Redes neuronales profundas.

MÉTODOS DE ENSAMBLE

MÉTODOS DE ENSAMBLE

¿Qué son?

En 1906, el estadístico Sir Francis Galton estaba visitando una feria del condado en Inglaterra, en la que se celebraba un concurso para adivinar el peso de un buey que se encontraba en exhibición. Hubo 800 conjeturas y, si bien las conjeturas individuales variaron ampliamente, tanto la media como la mediana estuvieron dentro del 1% del peso real del buey.

→ **Wisdom of the crowd (Sabiduría de la multitud)**

Si llevamos esta idea al campo del aprendizaje automático, al promediar la salida de un conjunto de predictores es altamente probable que obtengamos mejores métricas que considerando la salida de uno solo de ellos.

Dado que un conjunto de predictores se conoce como **ensamble**, estos métodos comúnmente se llaman **métodos de ensamble.**

MÉTODOS DE ENSAMBLE

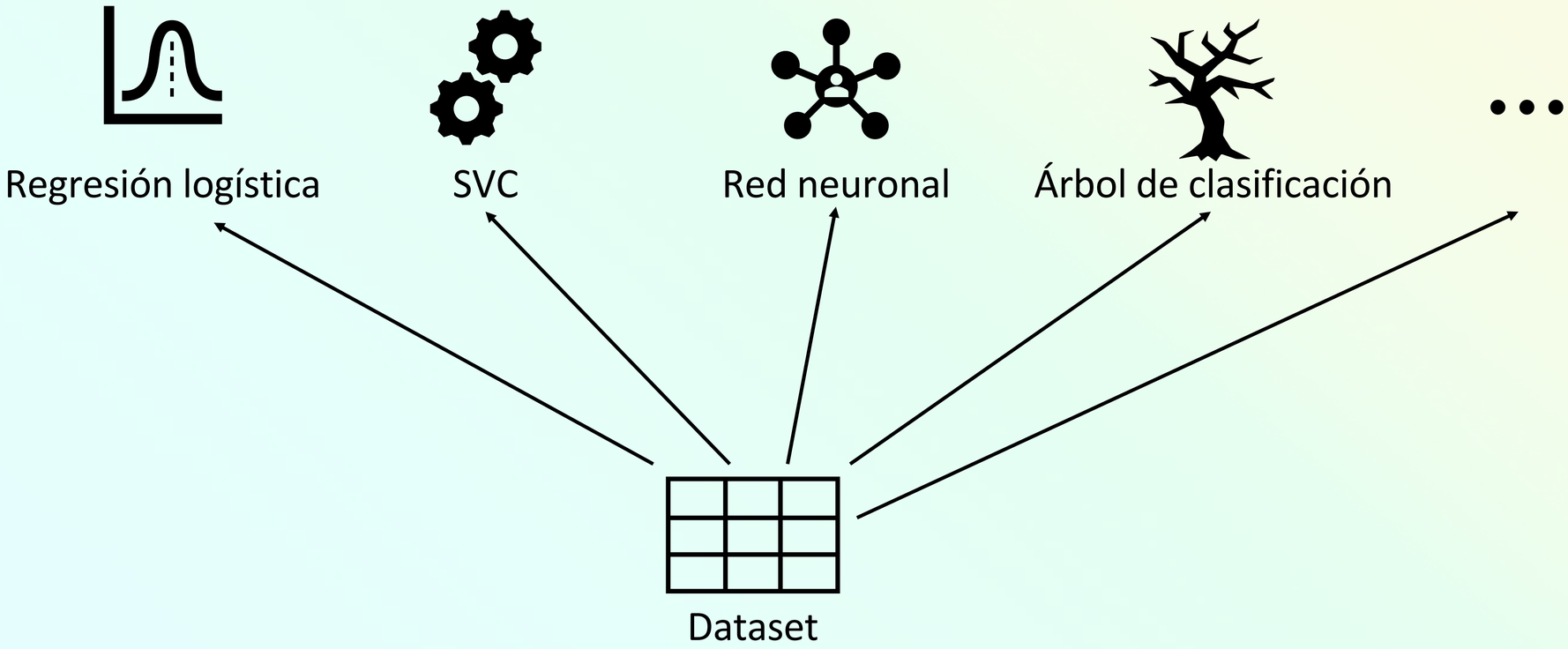
Por ejemplo, podríamos entrenar un grupo de árboles de decisión, cada uno en una parte diferente del conjunto de entrenamiento, generar predicciones con cada uno de los árboles individuales y luego predecir la clase que tenga mayor cantidad de votos.

¿Cuándo usarlos?

Es más frecuente utilizar métodos de ensamble cuando ya probamos algunas arquitecturas más simples (*weak learners*) y para poder mejorar las métricas obtenidas combinamos los predictores ya desarrollados en un ensamble.

MODELOS QUE VOTAN

MODELOS QUE VOTAN



MODELOS QUE VOTAN

Hard voting classifiers: Por ejemplo, podríamos entrenar un grupo de árboles de decisión, cada uno en una parte diferente del conjunto de entrenamiento, generar predicciones con cada árbol individual y luego predecir la clase que tenga mayor cantidad de votos.

Soft voting classifiers: En este caso, si los modelos entrenados tienen la posibilidad de estimar la probabilidad por clase, lo que se hace es calcular la posibilidad promedio de cada clase sobre la cantidad total de modelos.

Voting regressor: En este caso, dado que los modelso pueden dar una salida continua, se puede obtener la salida promedio (u otra medida de posición central) de los modelos.

MODELOS QUE VOTAN

Hard voting classifiers: Por ejemplo, podríamos entrenar un grupo de árboles de decisión, cada uno en una parte diferente del conjunto de entrenamiento, generar predicciones con cada árbol individual y luego predecir la clase que tenga mayor cantidad de votos.

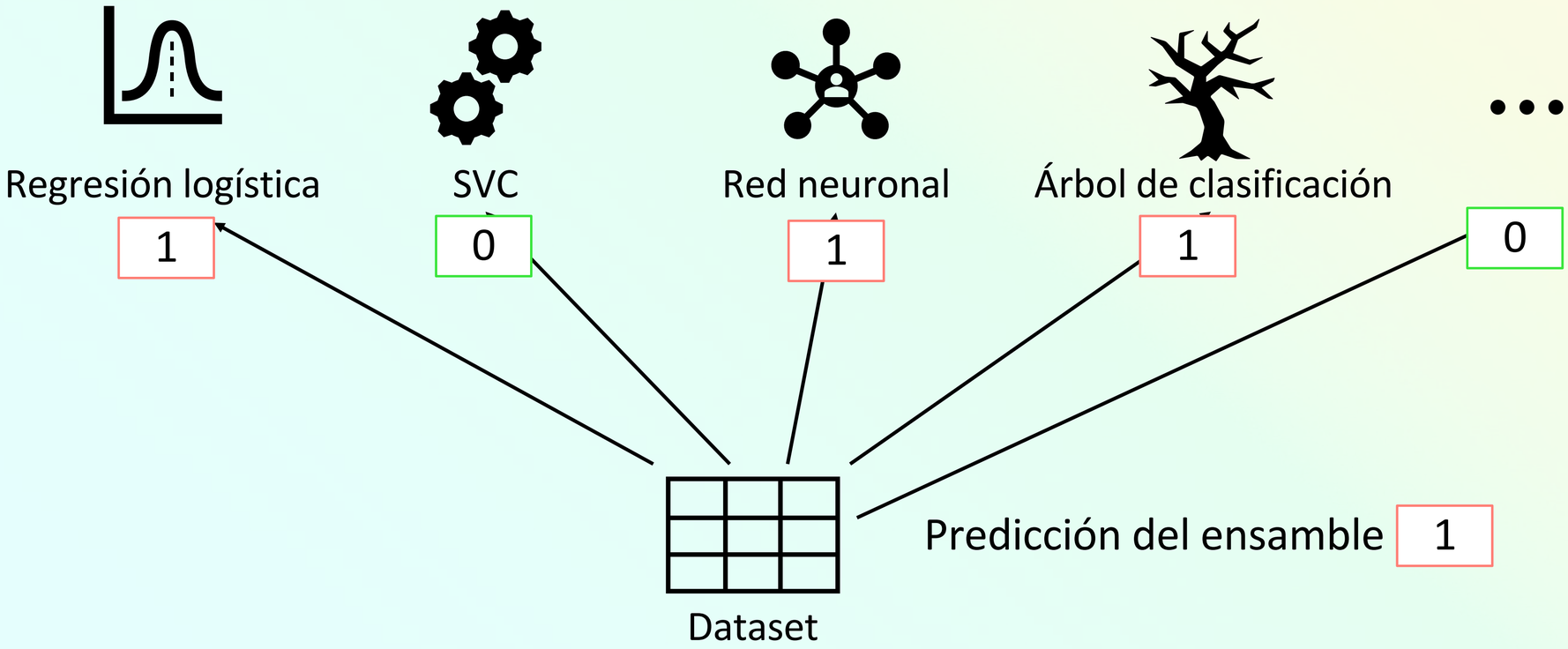
Soft voting classifiers: En este caso, si los modelos entrenados tienen la posibilidad de estimar la probabilidad por clase, lo que se hace es calcular la posibilidad prom
total de modelos.

Una ventaja de esta manera de estimar las clases es que se le da más peso a los votos que tienen más confianza en su decisión.

Voting regressor: En este caso, dado que los modelso pueden dar una salida continua, se puede obtener la salida promedio (u otra medida de posición central) de los modelos.

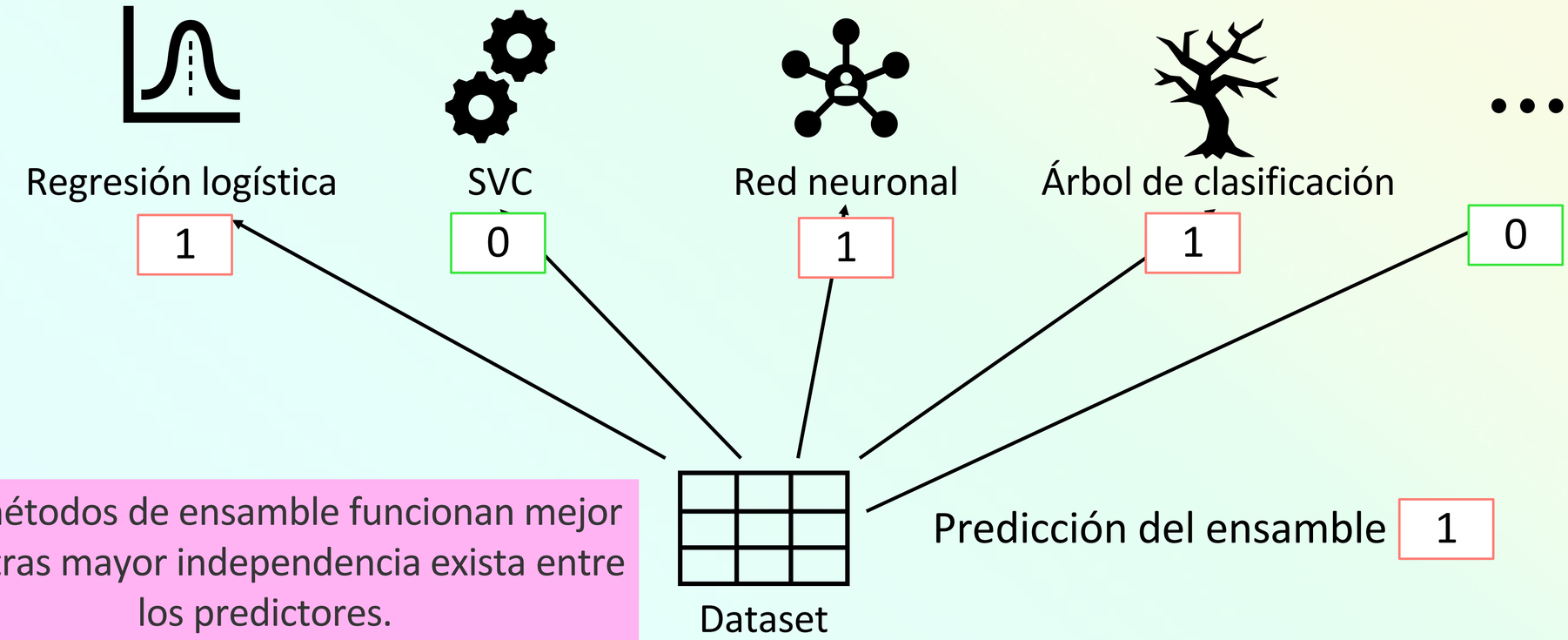
MODELOS QUE VOTAN

Hard voting classifiers



MODELOS QUE VOTAN

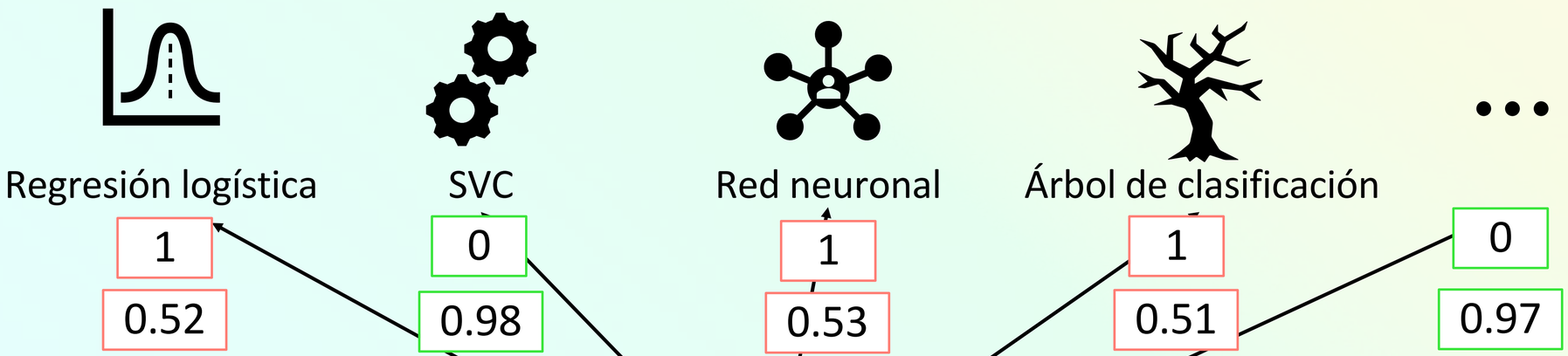
Hard voting classifiers

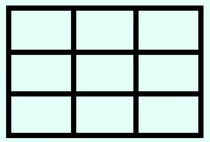


Los métodos de ensamble funcionan mejor mientras mayor independencia exista entre los predictores.
Por ej.: utilizando arquitecturas distintas para cada predictor.

MODELOS QUE VOTAN

Soft voting classifiers





Dataset

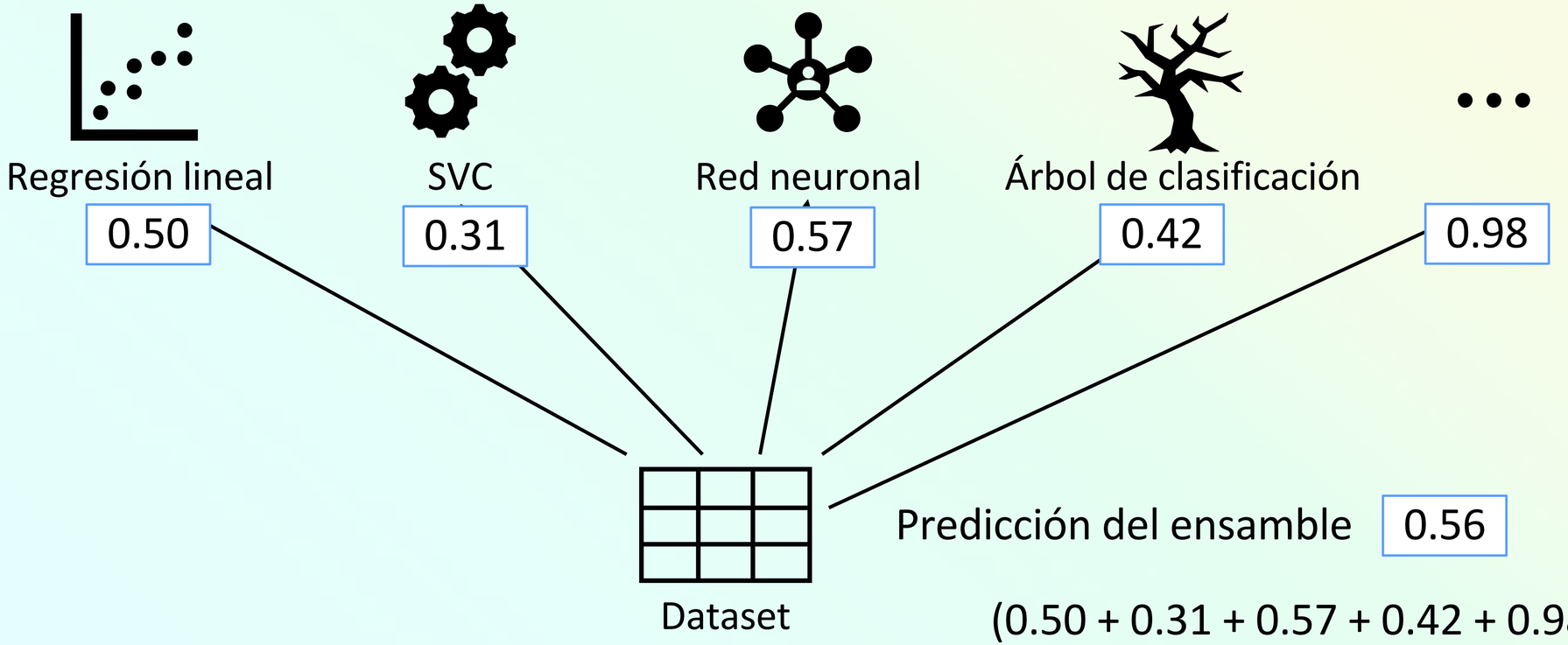
Predicción del ensamble 0

1 (0.52 + 0.02 + 0.53 + 0.51 + 0.03) / 5 = 0.312

0 (0.48 + 0.98 + 0.47 + 0.49 + 0.97) / 5 = 0.678

MODELOS QUE VOTAN

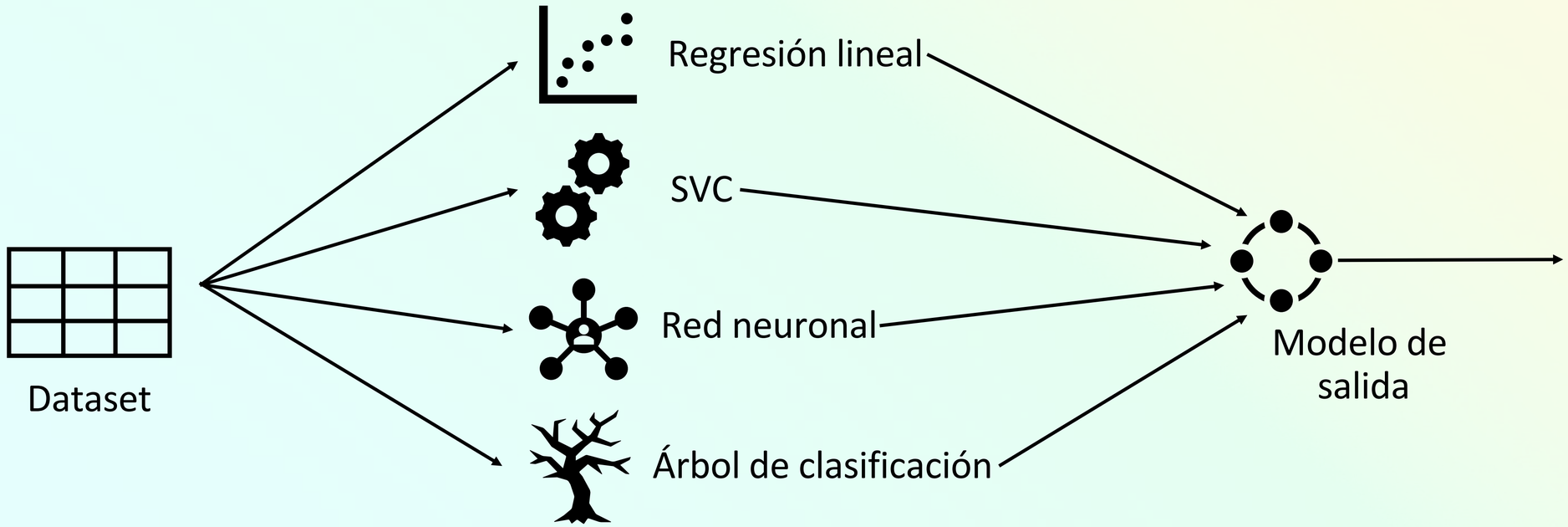
Voting regression



MODELOS QUE VOTAN

¿Y si reemplazamos el voto por algo más inteligente?

Una técnica un poco más avanzada es reemplazar el voto de los modelos, por un nuevo modelo que se entrena con la salida de los modelos.



VAMOS A PRÁCTICAR UN POCO…

BAGGING

BAGGING

¿Cómo logramos diversidad en nuestros predictores?

Una manera de lograr diversidad en nuestro ensamble es utilizando arquitecturas diferentes para cada predictor.

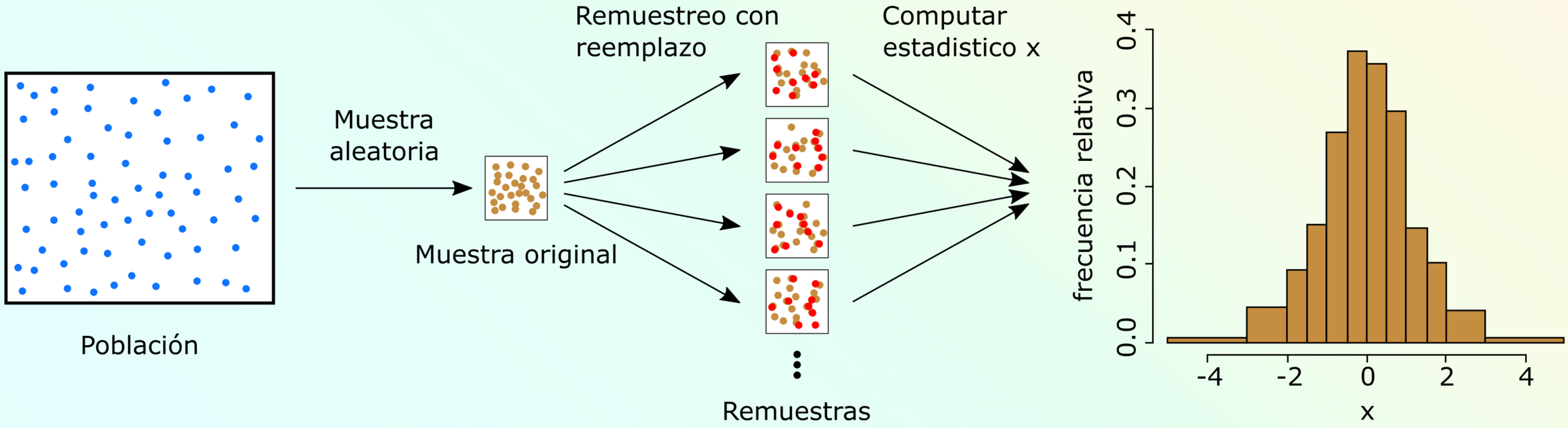
Otra manera de lograr esto es utilizando la misma arquitectura, pero entrenándola en distintos subconjuntos de nuestro set de entrenamiento.

Cuando el muestreo es **con reposición**, el método se llama **Bagging**. Cuando es **sin reposición**, se llama **Pasting**.

Cada una de las filas del dataset puede ser muestreada por múltiples predictores. Pero solo **Bagging** permite que una fila pueda ser muestreada más de una vez por el mismo predictor.

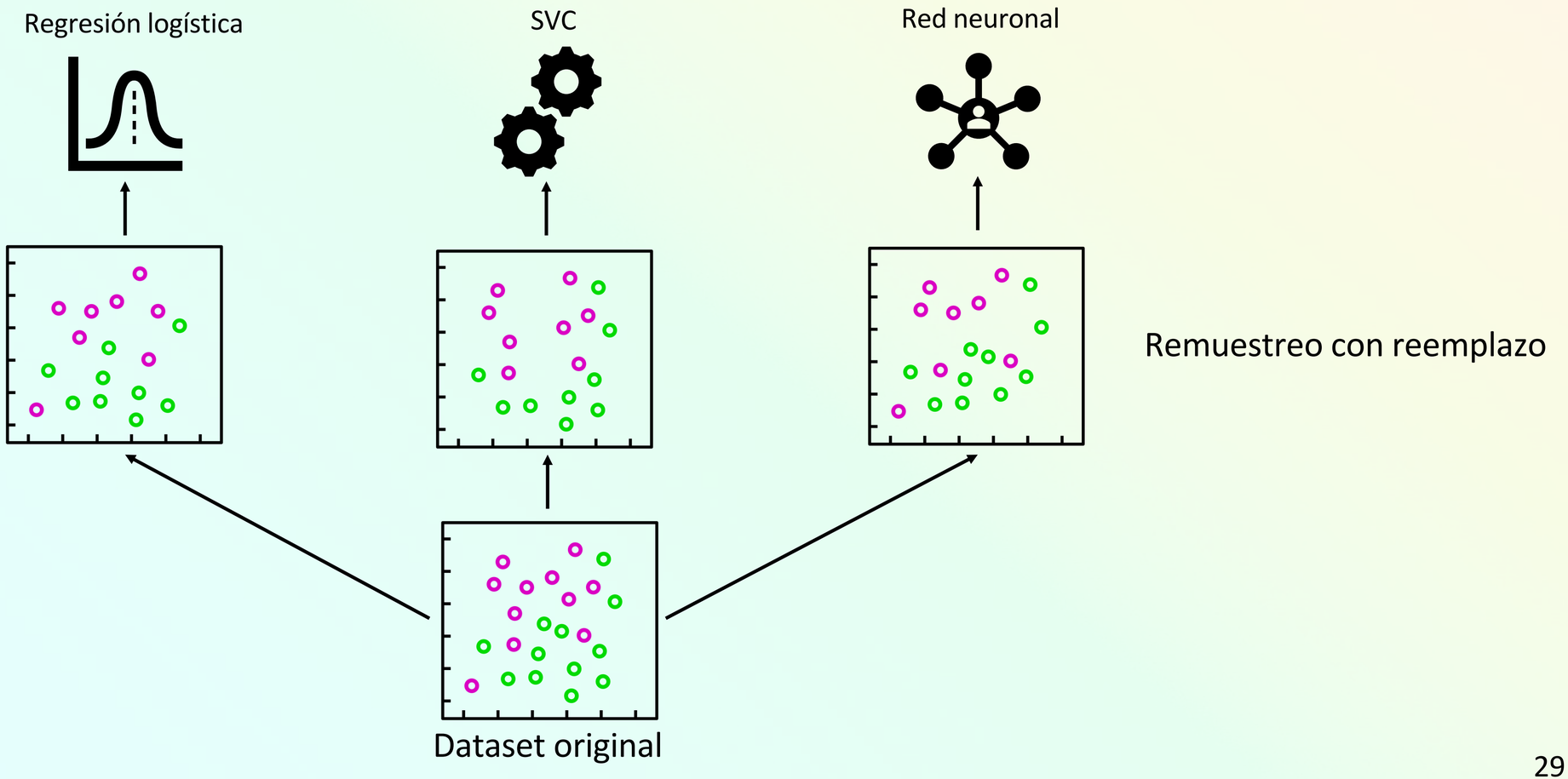
BAGGING

En estadística el muestreo con reemplazo se llama **Bootstrapping**



BAGGING

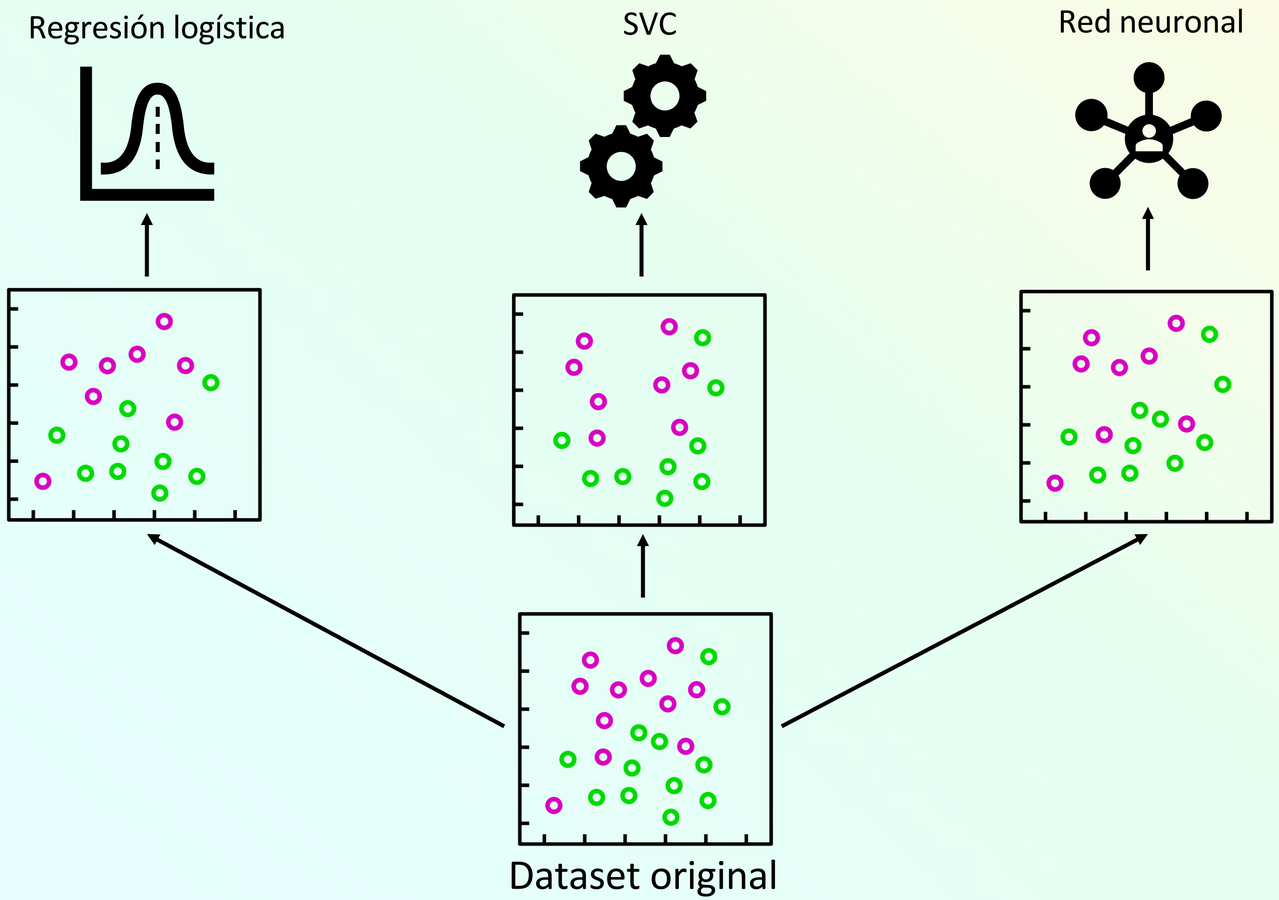
Para realizar **Bagging (bootstrap aggregating),** buscamos modelos que tengan mucha varianza y poco sesgo.



BAGGING

Para realizar **Bagging (bootstrap aggregating),** buscamos modelos que tengan mucha varianza y poco sesgo.

Cada uno de los modelos del ensamble puede ser entrenado en paralelo, lo que permite escalar la solución.



BAGGING

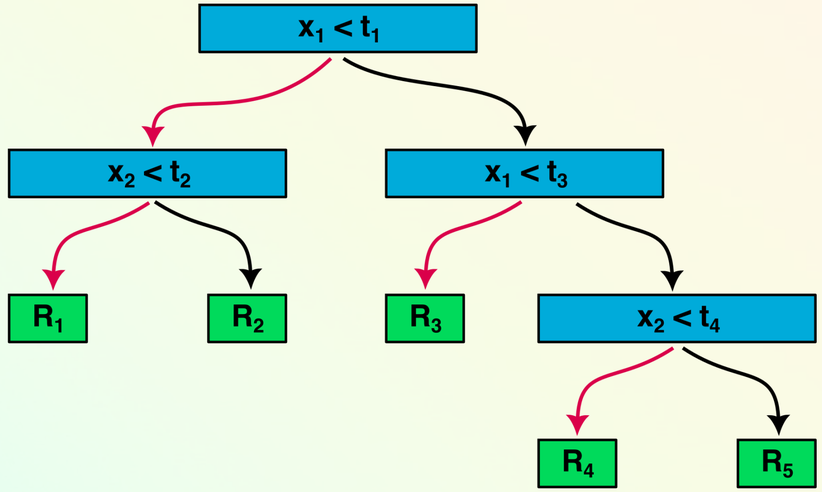
Lo más común es usar un solo tipo de modelo, particularmente el **árbol de decisión**.

- Si son de clasificación, estos árboles votan la clase.
- Si son de regresión, se promedia las predicciones.

Dado que cada árbol es entrenado diferente, va a ser diferente de los demás árboles.

En general estos **árboles son profundos**, es decir gran **varianza**, pero poco **sesgo**. Al promediar las salidas reducimos la varianza.

Bagging mejora sustancialmente los resultados cuando se lleva a cientos o miles de árboles.



BOSQUES ALEATORIOS

BOSQUES ALEATORIOS

Los **bosques aleatorios** son una mejora de los obtenidos mediante **bagging** únicamente.

Los bosques aleatorios hacen lo mismo que mediante bagging, pero además se usa una **cantidad aleatoria de atributos**. El valor de cuantos atributos a usar se elige aproximadamente la raíz cuadrada de la cantidad de atributos totales.

Por ejemplo, si tenemos p=13 atributos, se usarán m=4, y en cada árbol será una combinación al azar diferente.

BOSQUES ALEATORIOS

El razonamiento de esto es:

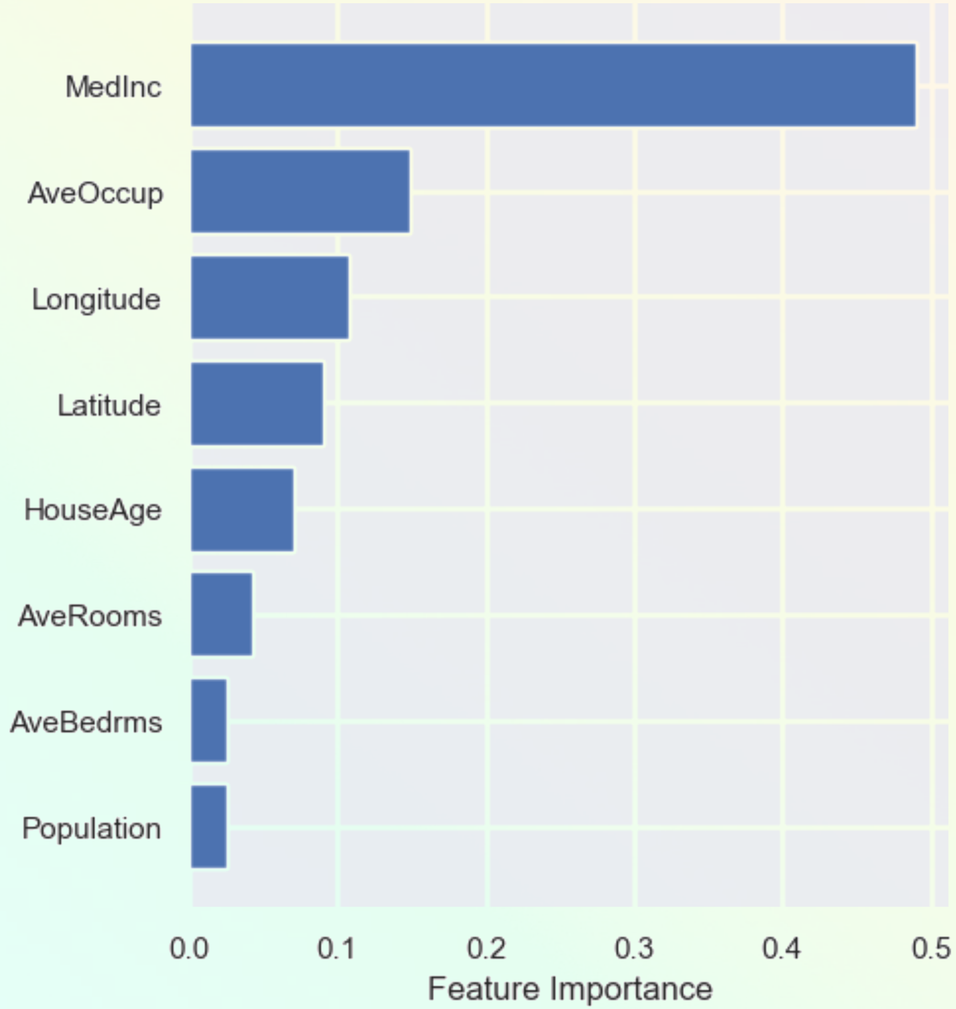
Supongamos que hay un atributo que es muy **fuerte predictor**, junto a otros atributos moderadamente fuertes. Si aplicamos **bagging**, todos estos árboles siempre tenderán a usar el **atributo fuerte** en la bifurcación inicial. Por consiguiente, todos serán parecidos y habrá una fuerte correlación entre árboles, **quitando la posibilidad de reducir la varianza**.

Bosques aleatorios, al separar los atributos, en promedio (p-m)/p de las particiones no van a considerar al atributo fuerte, quitando la correlación.

BOSQUES ALEATORIOS

El poder de bosques aleatorios se muestra cuando se crean modelos predictivos para datos con muchos atributos.

Este tiene la capacidad de determinar automáticamente qué predictores son importantes y descubrir relaciones complejas entre los predictores correspondientes a términos de interacción.



BOSQUES ALEATORIOS

Hay dos formas de medir la importancia de un atributo:

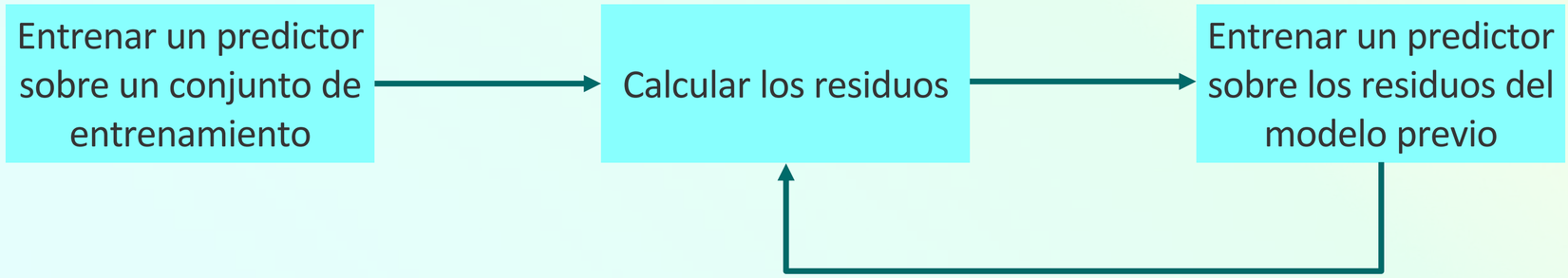
- Por la disminución de la precisión del modelo si los valores de un atributo se permutan aleatoriamente. Permutar aleatoriamente los valores tiene el efecto de eliminar todo poder predictivo de esa variable.
- Por la disminución media en la puntuación de impureza (de Gini, entropía, etc.) para todos los nodos que fueron divididos por el atributo. Esto mide cuánto mejora la pureza de los nodos al incluir ese atributo.

BOOSTING

BOOSTING

Este tipo de arquitecturas son ensambles que al igual que los vistos anteriormente, combinan varios predictores para hacer una mejor predicción. La principal diferencia es que en las arquitecturas que aplican **Boosting**, el entrenamiento de los predictores se da de forma secuencial.

Cada nuevo predictor que se agrega al ensamble intenta corregir a su predecesor.



BOOSTING

Hay varios algoritmos de complejidad variable. El más sencillo de entender es **Adaboost**. El algoritmo de entrenamiento se puede simplificar a:

1. Se comienza con un conjunto de datos de entrenamiento y se asigna pesos iguales a todas las muestras de entrenamiento.

2. Se entrena un modelo (weak learner) utilizando el conjunto de datos de entrenamiento.

3. Se calcula el error del modelo débil al predecir el conjunto de entrenamiento. Las observaciones que fueron mal clasificadas por el modelo débil se ponderan más en el siguiente paso.

4. Se calcula un peso para el modelo débil en función de su precisión. Los modelos que hacen predicciones más precisas tienen un mayor peso en la predicción final.

5. Se actualizan los pesos de las instancias de entrenamiento. Las observaciones que fueron mal clasificadas por el modelo débil anterior reciben un peso mayor para que el siguiente modelo débil se enfoque más en ellas.

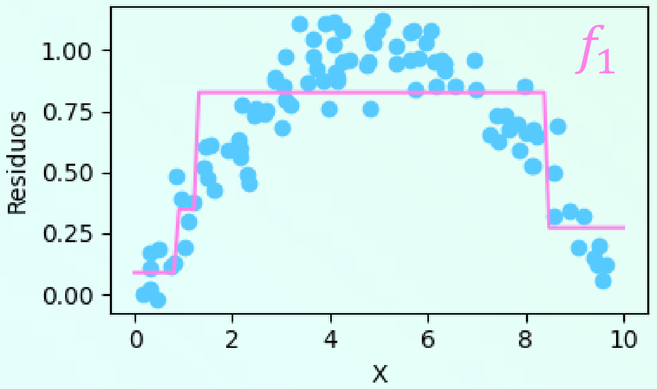
6. Se repiten los pasos 2-5 varias veces (generalmente de 50 a 100 iteraciones), cada vez construyendo un nuevo modelo débil y ajustando los pesos de las instancias.

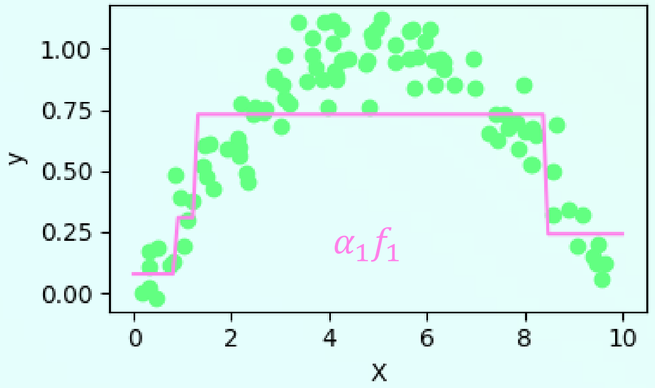
7. Finalmente, se combinan las predicciones de todos los modelos débiles, ponderadas por sus pesos respectivos, para obtener la predicción final del modelo:

$$\hat{F} = \alpha_1 \hat{f}_1 + \alpha_2 \hat{f}_2 + \ \ldots + \alpha_M \hat{f}_M$$

BOOSTING

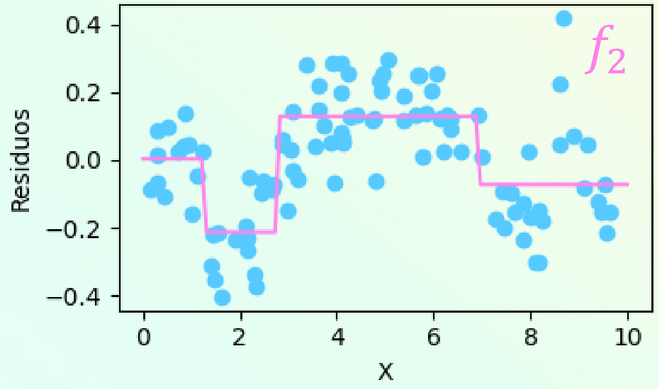
Iteración 1

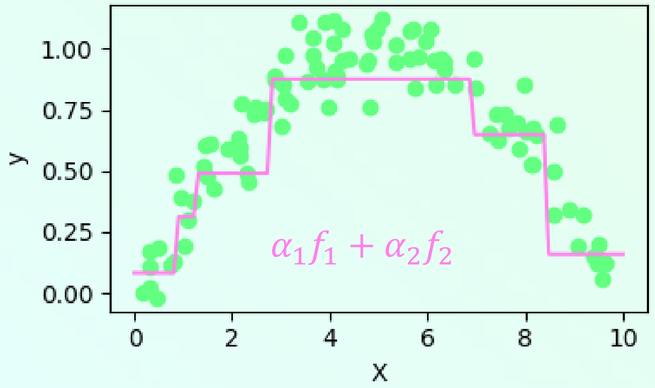




Salida de ensamble: $F = \alpha_1 f_1$

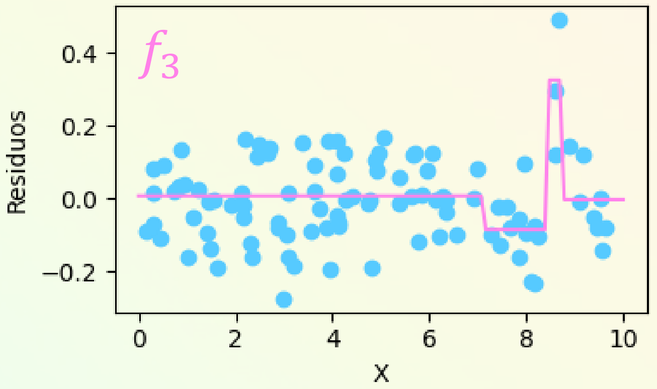
Iteración 2

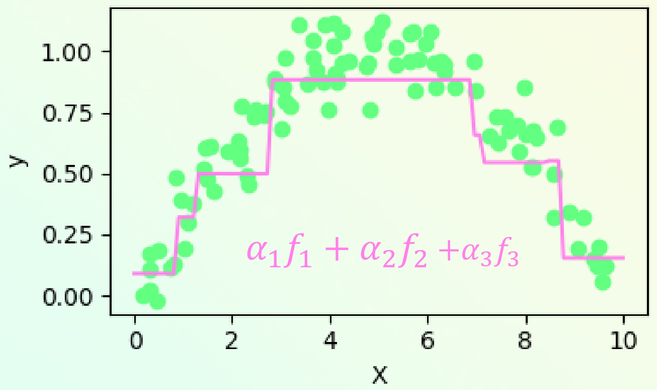




Salida de ensamble: $F = \alpha_1 f_1 + \alpha_2 f_2$

Iteración 3





Salida de ensamble: $F = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3$

BOOSTING

La principal diferencia es que en las arquitecturas que aplican **Boosting**, el entrenamiento de los predictores se da de forma secuencial.

Lo más común es usar árboles. En este caso se utilizan arboles con pocos nodos.

Esto es un proceso de aprendizaje lento que depende de árboles anteriores. Un proceso lento de aprendizaje estadísticamente lleva a buenos entrenamiento.

BOOSTING VS. BAGGING

	Bagging	Boosting
Método de construcción del conjunto de modelos	Múltiples modelos de aprendizaje automático independientes entrenados en subconjuntos aleatorios de datos de entrenamiento	Construye modelos secuencialmente, centrándose en corregir los errores del modelo anterior
Tipo de modelos	Adecuado para cualquier modelo de aprendizaje automático	Se enfoca en modelos débiles y mejora su precisión al combinar modelos débiles
Proceso de predicción	Promedio de las predicciones de todos los modelos del conjunto	Asigna diferentes pesos a cada modelo y los combina para hacer una predicción final
Sensibilidad al ruido	Menos sensible al ruido y a los valores atípicos	Puede ser más sensible al ruido y a los valores atípicos
Tiempo de entrenamiento	Puede ser más rápido ya que los modelos se construyen de forma independiente	Puede requerir más tiempo de entrenamiento ya que cada modelo se entrena secuencialmente
Ventajas	Reduce la varianza, mejora la precisión y puede manejar grandes conjuntos de datos	Mejora la precisión y el rendimiento del modelo mediante la corrección de errores y la creación de modelos más precisos
Desventajas	Puede aumentar el sesgo y no mejorar la precisión si los modelos del conjunto son similares	Puede ser más propenso al sobreajuste y al ruido si se usa un modelo débil o ruidoso

XGBOOST

XGBOOST

El software más utilizado en Boosting es **XGBoost**, una implementación del Boosting de gradiente estocástico desarrollado por Tianqi Chen y Carlos Guestrin en la Universidad de Washington.

En AdaBoost se usa una función exponencial para ponderar cada residuo, con XGBoost se utiliza al gradiente de la función de costo y con ello se pondera a los residuos y los weak learners.

XGBOOST

El modelo XGBoost tiene muchísimos hiperparámetros. Dos parámetros muy importantes son:

- **subsample**: Controla la fracción de observaciones que se deben muestrear en cada iteración. El uso de submuestra hace que actúe como un bosque aleatorio excepto que el muestreo se realiza sin reemplazo (**Pasting**).
- **eta**: Un factor de contracción aplicado a cada α_m de cada nuevo árbol que se va a agregando, el cual reduce el peso a las muestras mal clasificadas o con más error. El parámetro de contracción eta es útil para evitar el sobreajuste.

XGBOOST

En Python, **XGBoost** tiene dos interfaces:

- Una propia más cercana al paradigma funcional (cercana a R).
- Una que imita al API de Scikit-Learn. Para ser consistente con otros métodos de Scikit-Learn, algunos parámetros son renombrados, como por ejemplo eta se llama *learning_rate*.

Otra capacidad que presenta **XGBoost**, dado su facilidad de tener overfitting es **regularización**. Para ello se presenta dos parámetros que controlan dos partes:

- reg_alpha: El cual corresponde a una regularización L1 (Lasso)
- reg_lambda: El cual corresponde a una regularización L2 (Ridge)

Incrementar estos parámetros, penalizarán modelos y reducirán el tamaño de los árboles.

Otra implementación de este tipo de modelos muy popular es LightGBM.

VAMOS A PRÁCTICAR UN POCO…