Machine Learning Supervisado

Ing. Remigio Hurtado Ortiz, PhD.

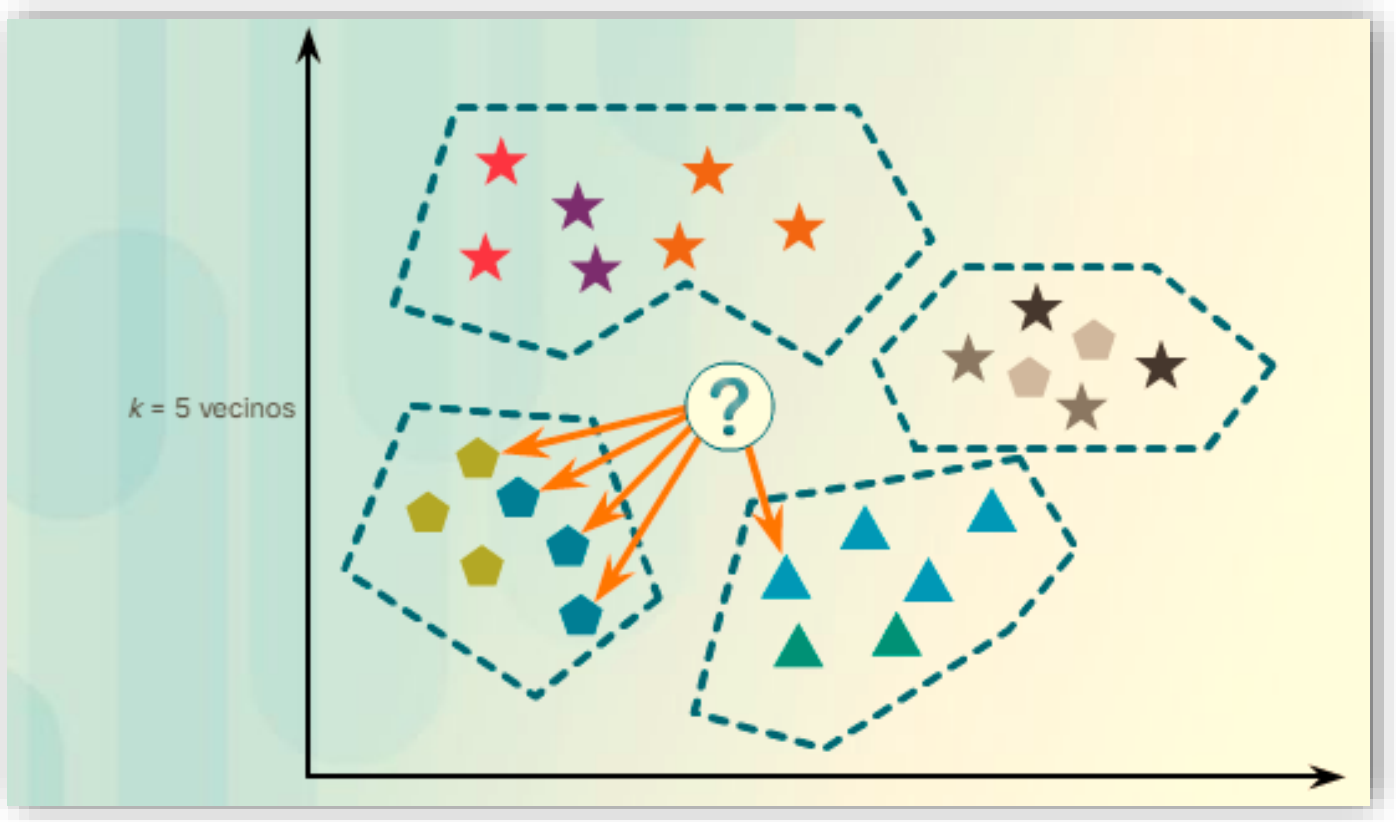
2023

Supervised Machine Learning (introducción y técnicas elementales)

Supervised Machine Learning (introducción)

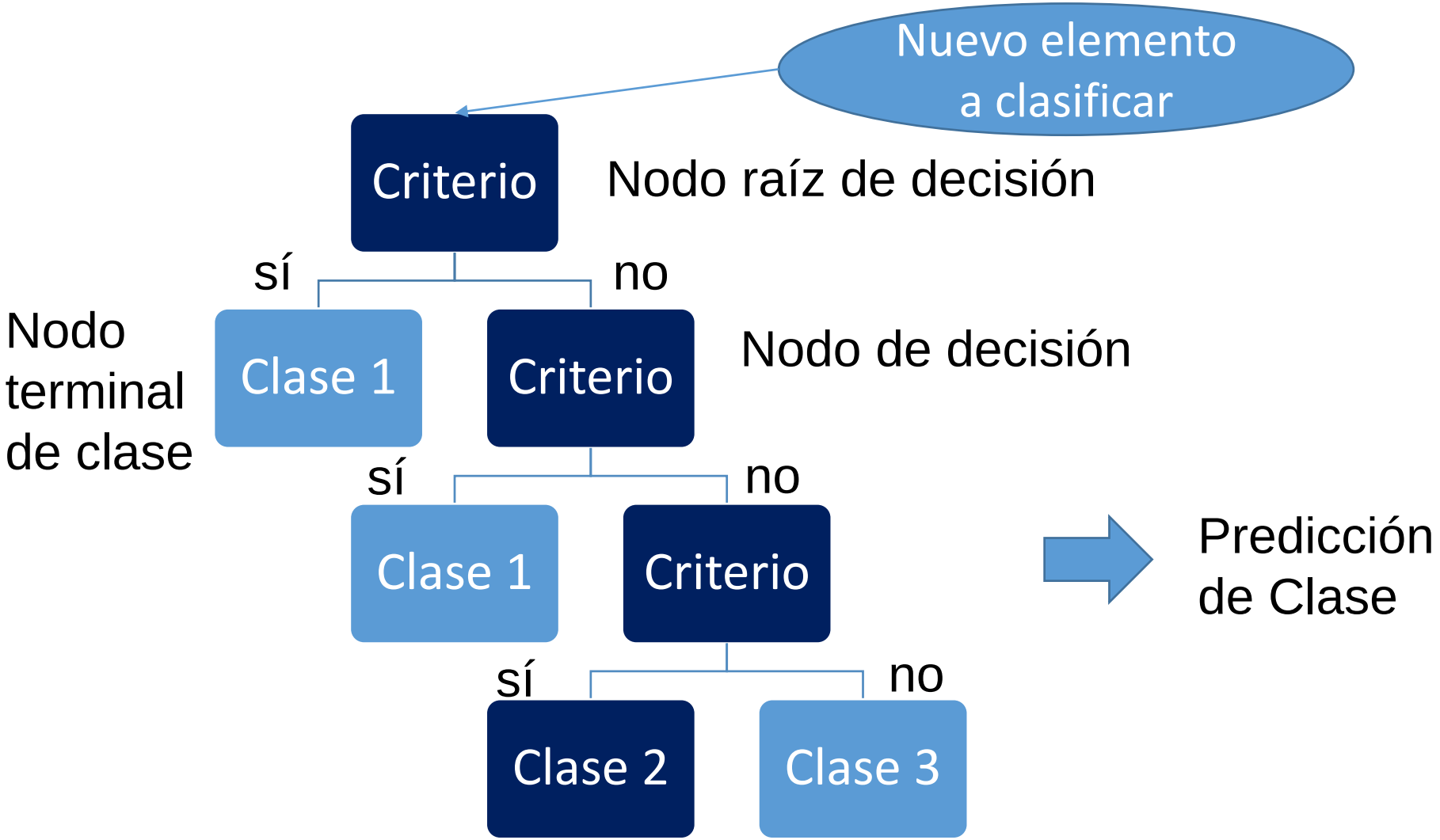
Algoritmos para clasificación: K-nearest neighbor (K vecinos)

K-nearest neighbor (kNN): KNN es posiblemente el **clasificador más intuitivo**, que utiliza la distancia entre los ejemplos de entrenamiento como **medida de similitud**. La distancia entre los puntos representa la diferencia entre los valores de sus funciones. Dado un nuevo punto de datos, un clasificador KNN debe ver los puntos de entrenamiento más cercanos. La **clase predicha** para el nuevo punto será la clase más común entre los K neighbors.



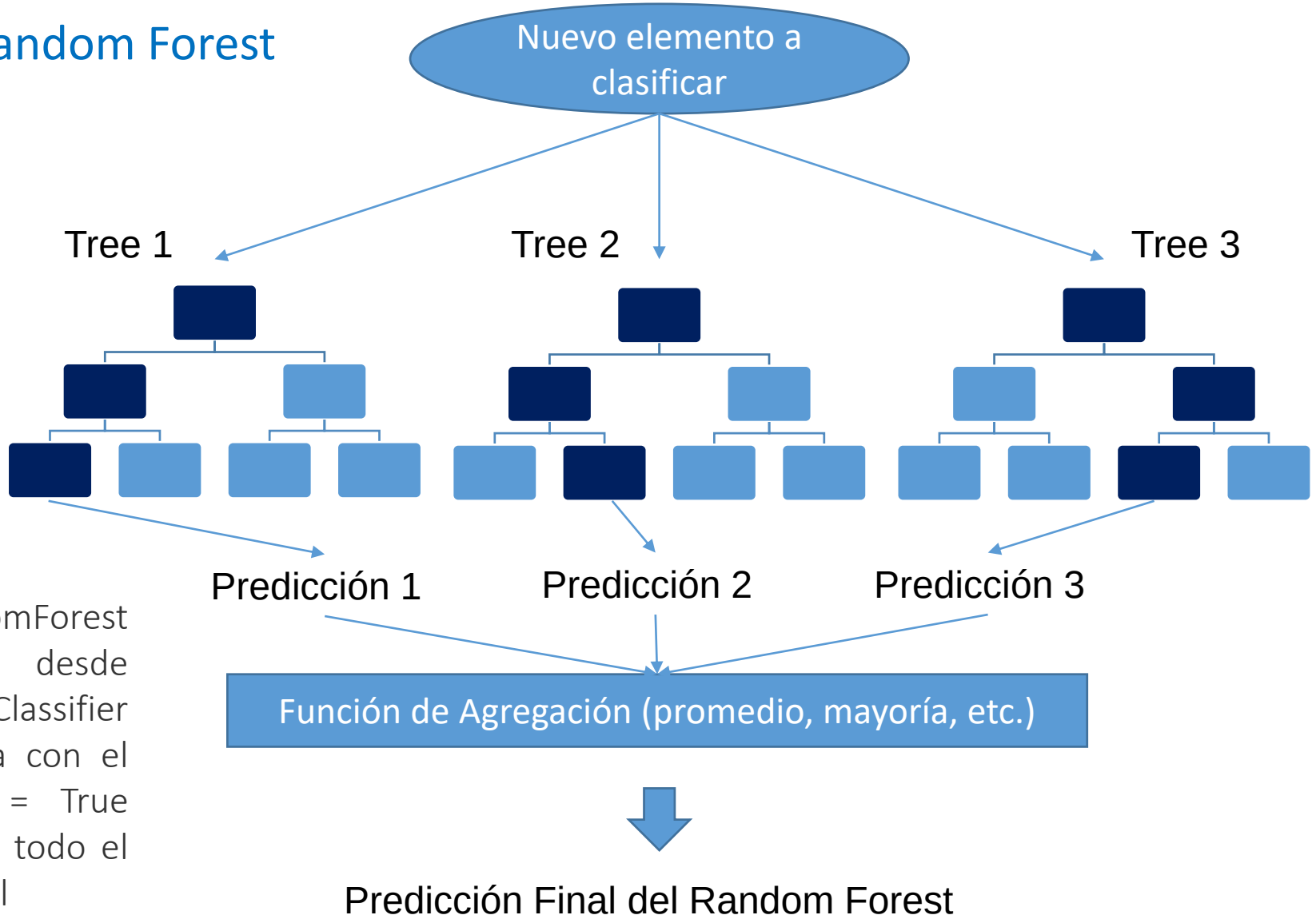
Algoritmos para clasificación: Decision Tree

Decision Tree (o Árboles de decisión): los árboles de decisión representan un problema de clasificación como un **conjunto de decisiones basadas en los valores de las funciones**. Cada nodo del árbol representa un umbral sobre el valor de una función, y parte los ejemplos de entrenamiento en dos grupos más pequeños. **Entrenamiento: El proceso de decisión se repite sobre todas las características, con lo que el árbol crece hasta que una manera óptima de dividir los ejemplos se computa**. **Predicción:** la clasificación de un nuevo ejemplo luego puede obtenerse siguiendo las ramas del árbol según los valores de sus funciones.



Algoritmos para clasificación: Random Forest

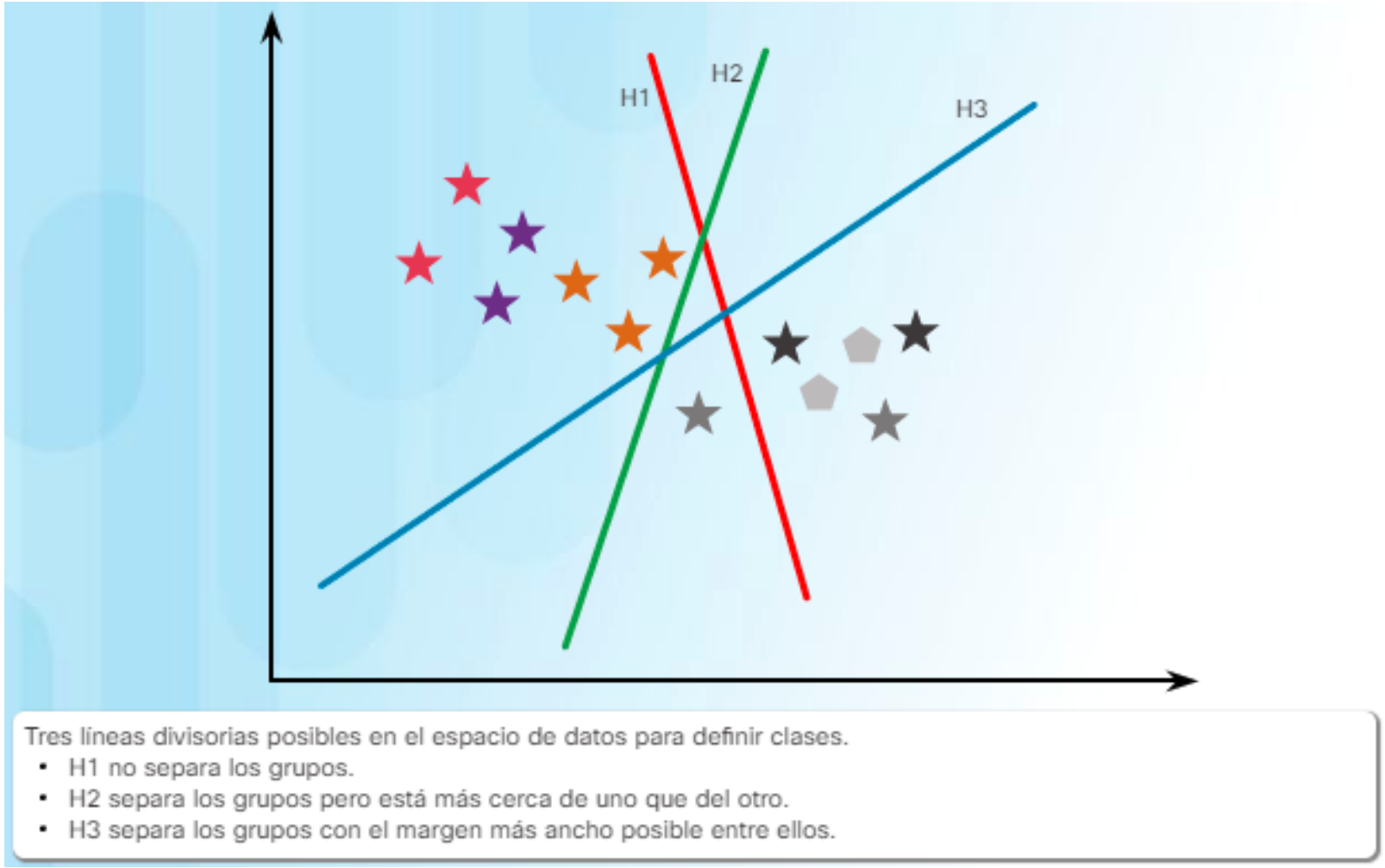
Un **Random Forest** es un **metaestimador** que se ajusta a una serie de **clasificadores** de árboles de decisión "**Decision Tree**" en varias submuestras del conjunto de datos y utiliza una **función de agregación** (ejemplo: promedios) para mejorar la precisión predictiva y controlar el sobreajuste.



En Python se puede desarrollar un RandomForest con RandomForestClassifier desde sklearn.ensemble. Con un RandomForestClassifier el tamaño de la submuestra se controla con el parámetro max_samples si bootstrap = True (predeterminado); de lo contrario, se usa todo el conjunto de datos para construir cada árbol

Algoritmos para clasificación o regresión: SVM

Máquinas de vectores de soporte (SVM): las máquinas de vector de soporte (SVM), que se muestran en la figura, son ejemplos de **clasificadores** de aprendizaje automático supervisados. En lugar de basar la asignación de membresía de la categoría en distancias de otros puntos, las máquinas de vector de soporte **computan la frontera, o el hiperplano, que mejor separa los grupos**. En la figura, el H3 es el hiperplano que maximiza la distancia entre puntos de entrenamiento de las dos clases, visibles en color o en blanco y negro. Cuando se presenta un nuevo punto de datos, se clasifica según si se encuentra en un lado o en el otro de H3. Una SVM también se puede diseñar para **regresión**.



Tres líneas divisorias posibles en el espacio de datos para definir clases.

- H1 no separa los grupos.
- H2 separa los grupos pero está más cerca de uno que del otro.
- H3 separa los grupos con el margen más ancho posible entre ellos.

Algoritmos para regresión: Multiple Linear Regression

Definiciones:

Dos tipos de algoritmos de machine learning supervisado: Regresión y Clasificación.

Regresión: predice valores continuos. (el precio de una casa en dólares).

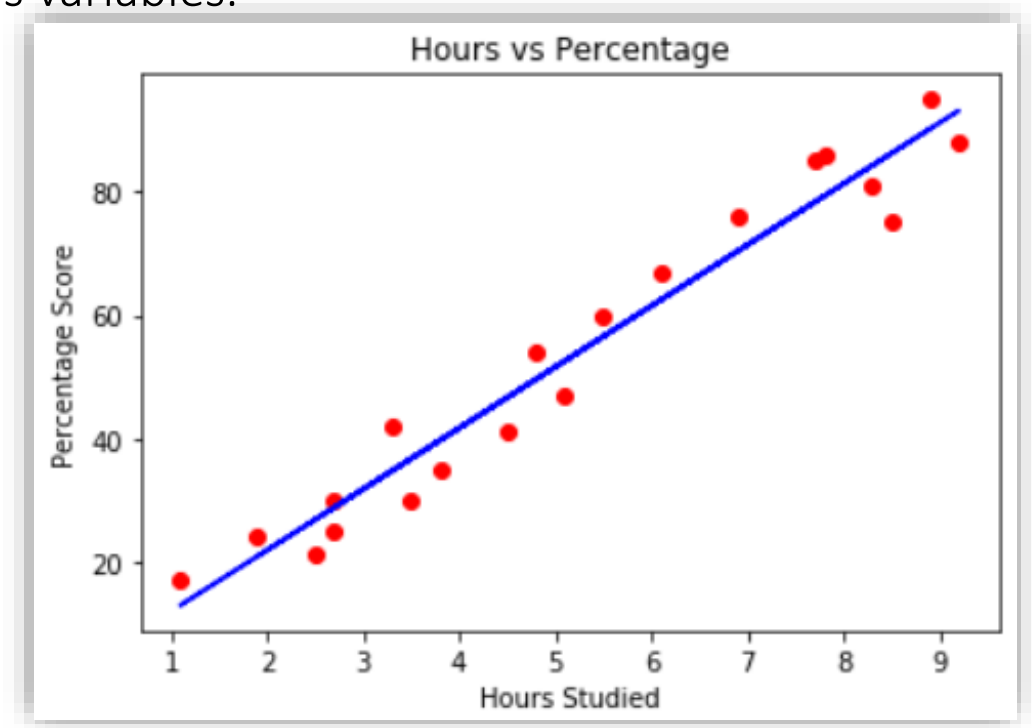
Clasificación: predice salidas discretas (clasificar un tumor como maligno o benigno).

Linealidad (Linearity) se refiere a la relación lineal entre dos o más variables.

Variable independiente (X): variables explicativas.

Variable dependiente (Y): la que deseamos predecir.

Ejemplo: determinar la **relación lineal** entre el número de horas (variable independiente) que estudia un estudiante y el porcentaje de calificaciones (variable dependiente) que el estudiante obtiene en un examen. Queremos descubrir que dado la cantidad de horas que un estudiante se prepara para un examen, ¿qué tan alto es el puntaje que puede alcanzar el estudiante?



Algoritmos para regresión: Multiple Linear Regression

La ecuación de una recta es básicamente:

```
y = mx + b
```

$$y = b_0 + b_1 \cdot x + u$$

Donde **b es la intersección (constante)** y **m es la pendiente de la recta**. Entonces, básicamente, el algoritmo de regresión lineal nos da el valor más óptimo para la intersección y la pendiente (en dos dimensiones). Las **variables y** y **x** siguen siendo las mismas, ya que son las características de los datos y no se pueden cambiar. Los valores que podemos controlar son la intersección y la pendiente. Puede haber múltiples líneas rectas dependiendo de los valores de intercepción y pendiente. Básicamente, **lo que hace el algoritmo de regresión lineal es que se ajusta a varias líneas en los puntos de datos y devuelve la línea que produce el menor error**.

Este mismo concepto se puede extender a los **casos donde hay más de dos variables**. Esto se llama **regresión lineal múltiple**. Por ejemplo, considere un escenario en el que tiene que predecir el precio de la casa según su área, el número de habitaciones, el ingreso promedio de las personas en el área, la edad de la casa, etc. En este caso, la variable dependiente depende de varias variables independientes. Un modelo de regresión que involucra múltiples variables se puede representar como:

```
y = b0 + m1b1 + m2b2 + m3b3 + ... ... mnbn
```

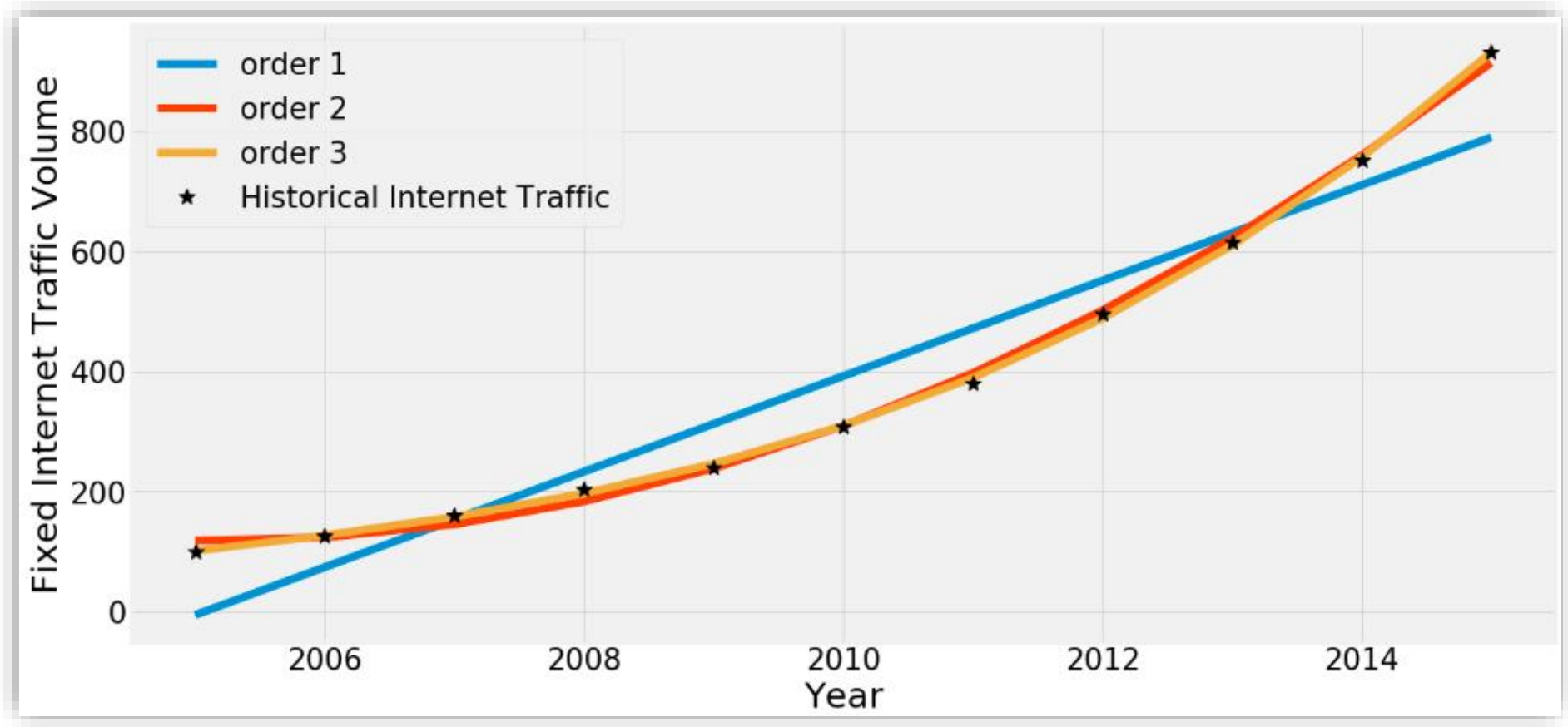
$$y = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 + b_3 \cdot x_3 + \ldots + b_k \cdot x_k + u$$

b_k son los coeficientes de pendiente para cada variable. b_0 es la intersección (constante). Un modelo de regresión lineal en dos dimensiones es una **línea recta**; en tres dimensiones es un **plano**, y en más de tres dimensiones, un **hiper plano**.

Algoritmos para regresión: Polynomial Multiple Regression (polinomios de orden superior)

Segundo orden: $y = a_0 + a_1 x + a_2 x^2$

Tercer orden: $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$

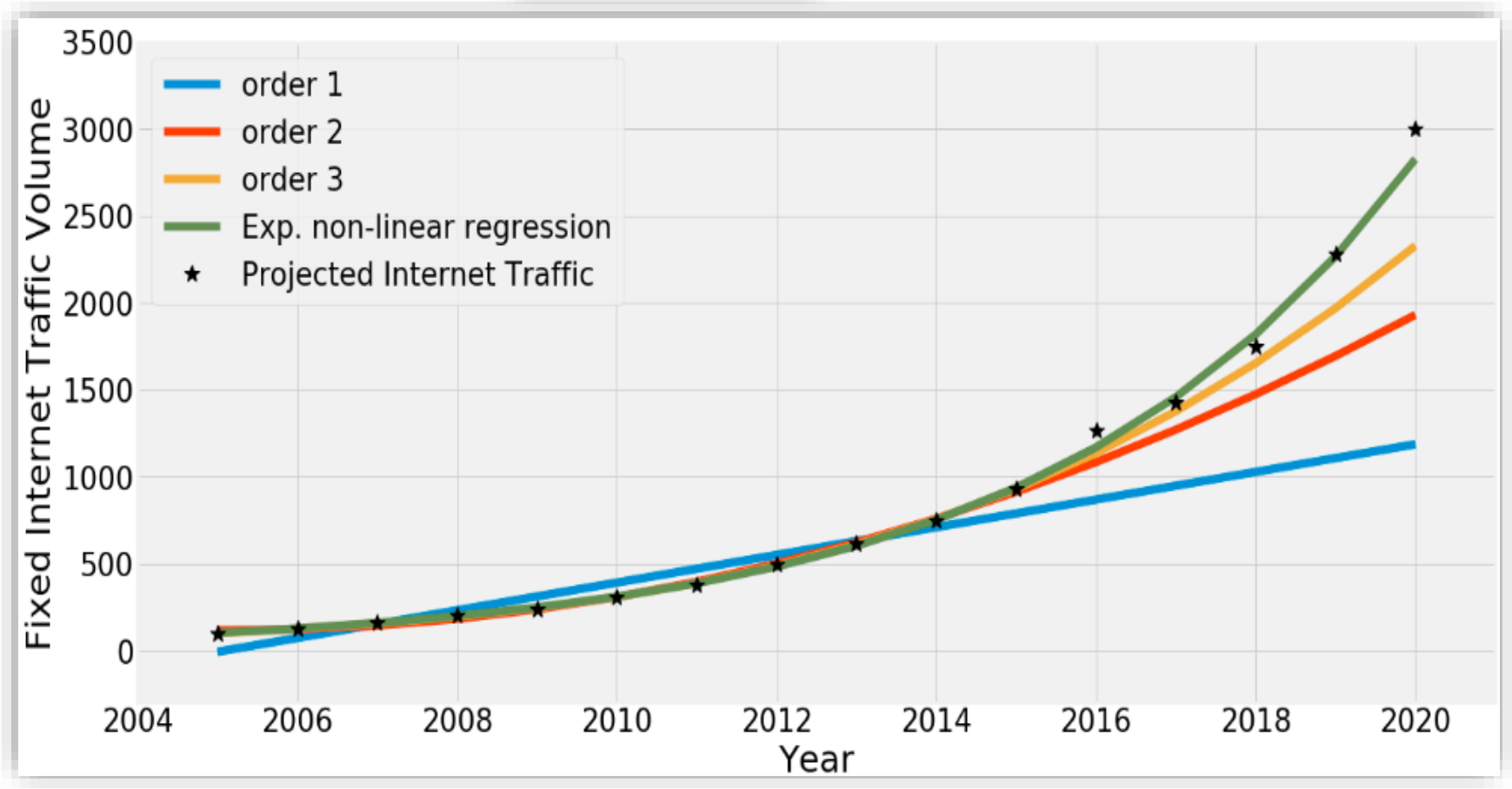


Algoritmos para regresión: Exponential Multiple Regression (crecimiento exponencial)

La fórmula destaca el hecho de que la **relación ente las variables dependientes e independiente implica una exponenciación**. Por este motivo, esta fórmula se llama **crecimiento exponencial**, y puede utilizarse **para describir muchos fenómenos de la naturaleza**

$$y = a * (1 + r)^n$$

a **es el valor inicial del volumen de tráfico**, que es la variable dependiente (a predecir), r es el índice de crecimiento anual (expresado como la relación incremental entre los valores de dos años consecutivos; p. ej.: 0,22) y n **es la cantidad de años** transcurridos desde 2015, que es otra variable dependiente.



Redes Bayesianas

1. Fundamentos
2. Aplicaciones
3. Construcción de redes
4. Naive Bayes

Redes Bayesianas: fundamentos

Las Redes Bayesianas han dado forma a problemas complejos que proporcionan información y recursos limitados. Se está implementando en las tecnologías más avanzadas de la era como la Inteligencia Artificial y el Aprendizaje Automático. Disponer de un sistema de este tipo es una necesidad en el mundo actual centrado en la tecnología.

Una red bayesiana pertenece a la **categoría de técnica de modelado gráfico probabilístico (PGM)** que se utiliza para calcular las incertidumbres mediante el concepto de probabilidad. Conocidas popularmente como Belief Networks (Redes de Creencias), las Redes Bayesianas se utilizan para modelar incertidumbres mediante el uso de **Grafos Acíclicos Dirigidos (DAG)**.

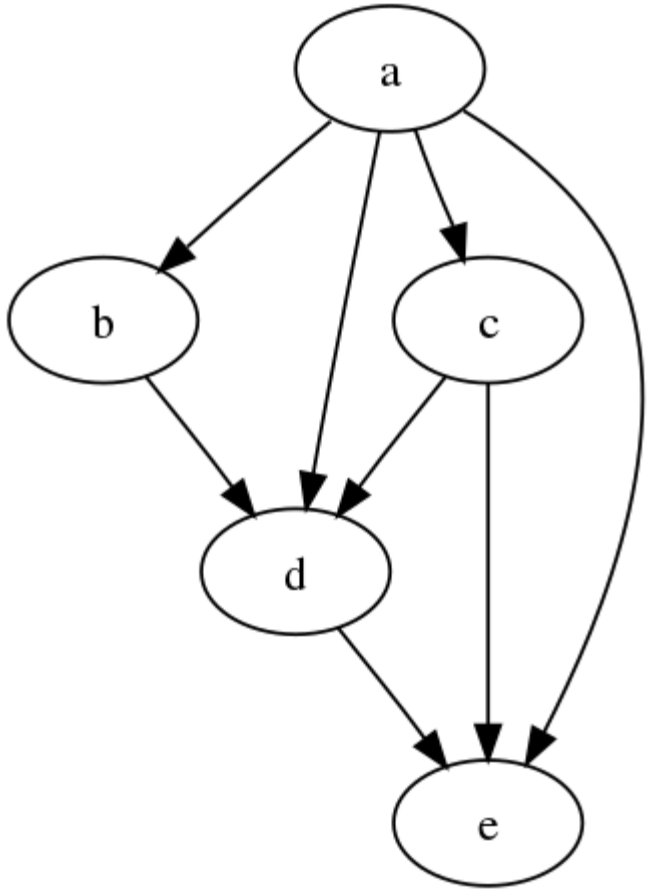
Un DAG modela la incertidumbre de que se produzca un evento basándose en la Distribución de Probabilidad Condicional (CPD) de cada variable aleatoria. Se utiliza una Tabla de Probabilidad Condicional (CPT) para representar la CPD de cada variable en la red.

Redes Bayesianas: fundamentos

Grafos Acíclicos Dirigidos (DAG)

En teoría de grafos e informática, un grafo acíclico dirigido (DAG) es un grafo dirigido sin ciclos dirigidos. En otras palabras, está formado por vértices y aristas (también llamados arcos), con cada arista apuntando de un vértice al siguiente, de tal manera que seguir esas direcciones nunca llevaría a un bucle cerrado como se representa en la figura.

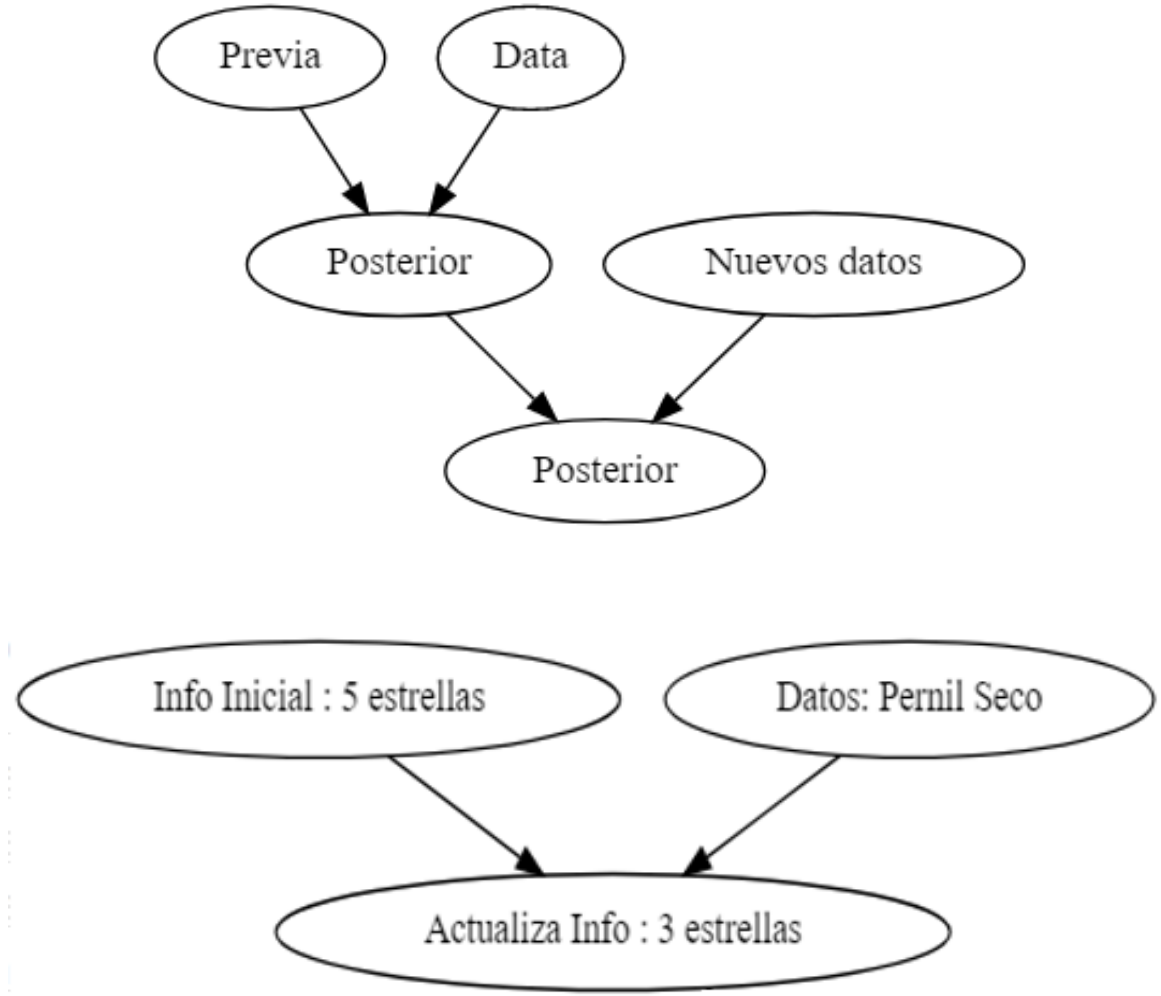
Cada nodo representa variables aleatorias (discretas o continuas) y cada arista representa las conexiones directas entre ellas.



Redes Bayesianas: fundamentos

Los seres humanos vamos actualizando nuestro conocimiento acerca del mundo a medida que pasa el tiempo, es decir, **vamos acumulando experiencias vividas o recolectando datos** [6].

Es importante observar que este proceso Bayesiano se puede aplicar a investigaciones rigurosas en diversos campos científicos. Por ejemplo, si una persona realiza investigación en el ámbito educativo, estará interesada en conocer cómo influyen diferentes factores como la alimentación, las horas de sueño, el tiempo de estudio, etc. en el rendimiento académico del estudiante. Sin embargo, para realizar esta investigación, se parte de conocimiento previo (previa), y **con ayuda de dicho conocimiento se interpretan los nuevos datos**, sopesando la información posterior [6].



Redes Bayesianas: aplicaciones

Las Redes Bayesianas se aplican en una gran cantidad de ámbitos de la ciencia. Algunos de los ejemplos más destacados son los que se indican seguidamente [7]:

- **Diagnóstico médico** (se generan hipótesis para cada enfermedad que tiene el paciente dado un conjunto de observaciones de síntomas).
- **Genética** (se emplean Redes Bayesianas para modelar la base de un rasgo complejo y para modelar fenotipos complejos).
- **Análisis de factores de riesgo delictivo** (se emplean Redes Bayesianas para tratar de explicar aspectos relacionados con los crímenes que cometen las personas).
- **Clasificación de documentos** (identificar a que clase o clases pertenece un documento en base a su contenido).
- **Busca semántica** (realizar búsquedas entendiendo el contexto del contenido).

Redes Bayesianas vs Cadenas de Markov

- Una red de Markov es similar a una red bayesiana en su representación de las dependencias; las **diferencias** son que las redes bayesianas son dirigidas y acíclicas, mientras que las redes de Markov no son dirigidas y pueden ser cíclicas.

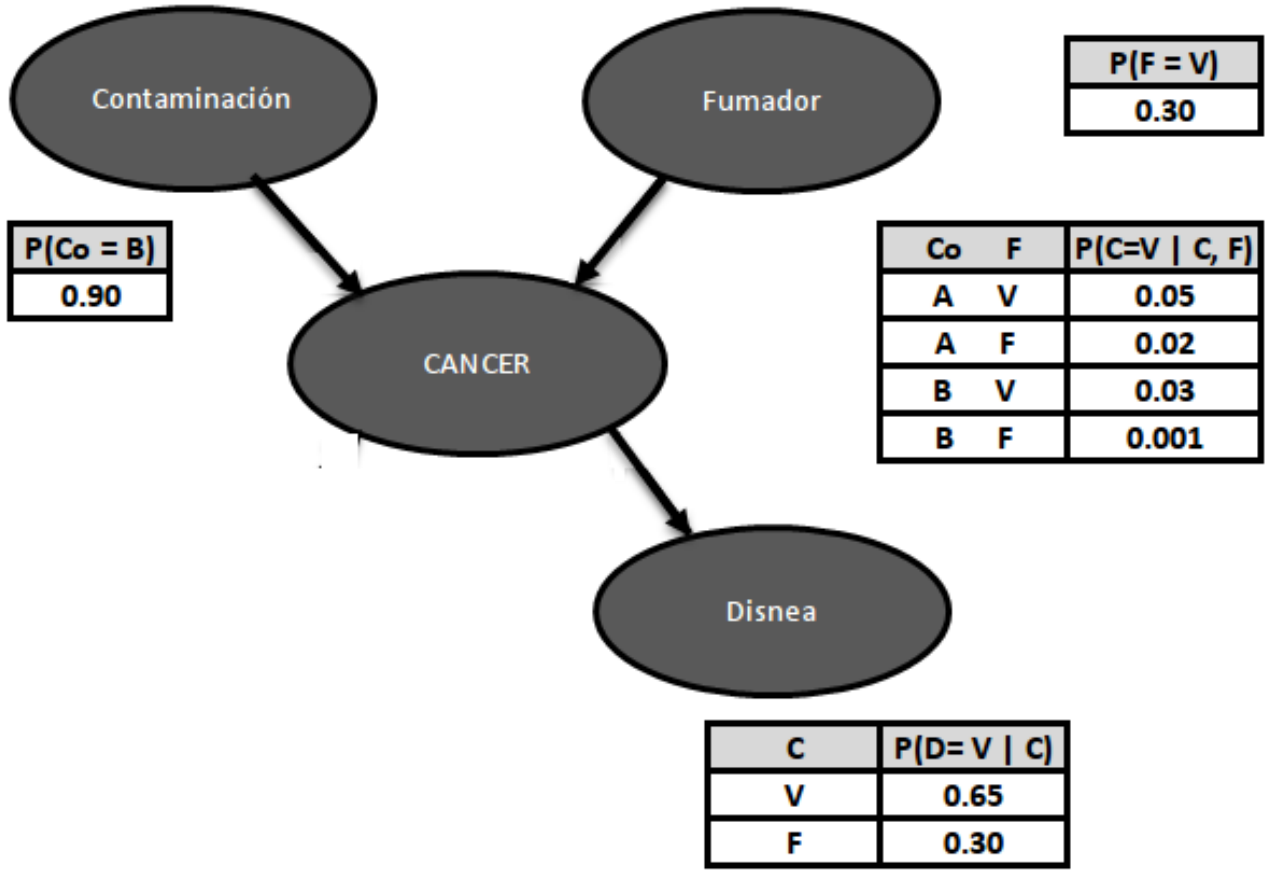
Redes Bayesianas: construcción

- ¿Cuáles son las variables? ¿Cuáles son sus valores / estados?
- ¿Cuál es la estructura del grafo?
- ¿Cuáles son los parámetros (probabilidades)?

- **Nodos:** booleanos, ordinales, enteros
- **Estructura:** dos nodos deben conectarse directamente si uno afecta o causa al otro, con la arista indicando la dirección del efecto.
- Ejemplo: ¿Qué factores afectan la probabilidad de tener cáncer? Si la respuesta es "Contaminación y Fumar", entonces agregar aristas desde "Contaminación" y desde "Fumar" hacia el nodo "Cáncer". Del mismo modo, tener cáncer afectará la respiración del paciente. Por lo tanto, también podemos agregar aristas de "Cáncer" a "Disnea".

Redes Bayesianas: construcción

- Cada nodo es una variable aleatoria discreta o continua. Cada nodo hijo tiene las probabilidades condicionales con respecto a sus nodos padre.
- Contaminación (Co): alta A o baja B. Fumador (F): verdadero V o falso F.
- Cáncer (C): verdadero V o falso F.



Redes Bayesianas: Naive Bayes

- Naive Bayes es una técnica basada en el teorema de Bayes que se utiliza en problemas en el que las variables son independientes (no correlacionadas). Se conoce como un algoritmo ingenuo, es decir, que supone que las variables son independientes.
- **Aprendizaje**: El modelo de Naive Bayes debe aprender las probabilidades condicionales $P(X_i|Y)$ donde X es una característica y Y es la clase. Estas probabilidades se pueden estimar a partir de los datos de entrenamiento.
- **Clasificación**: Una vez que el modelo ha sido entrenado, se puede utilizar para clasificar nuevas instancias. el algoritmo calcula la probabilidad condicional de pertenecer a cada clase "y" utilizando el teorema de Bayes.

$$P(y|x_1, x_2, ..., x_n) \propto P(y) \cdot P(x_1|y) \cdot P(x_2|y) \cdot ... \cdot P(x_n|y)$$

- La instancia se asigna a la clase con la mayor probabilidad condicional.

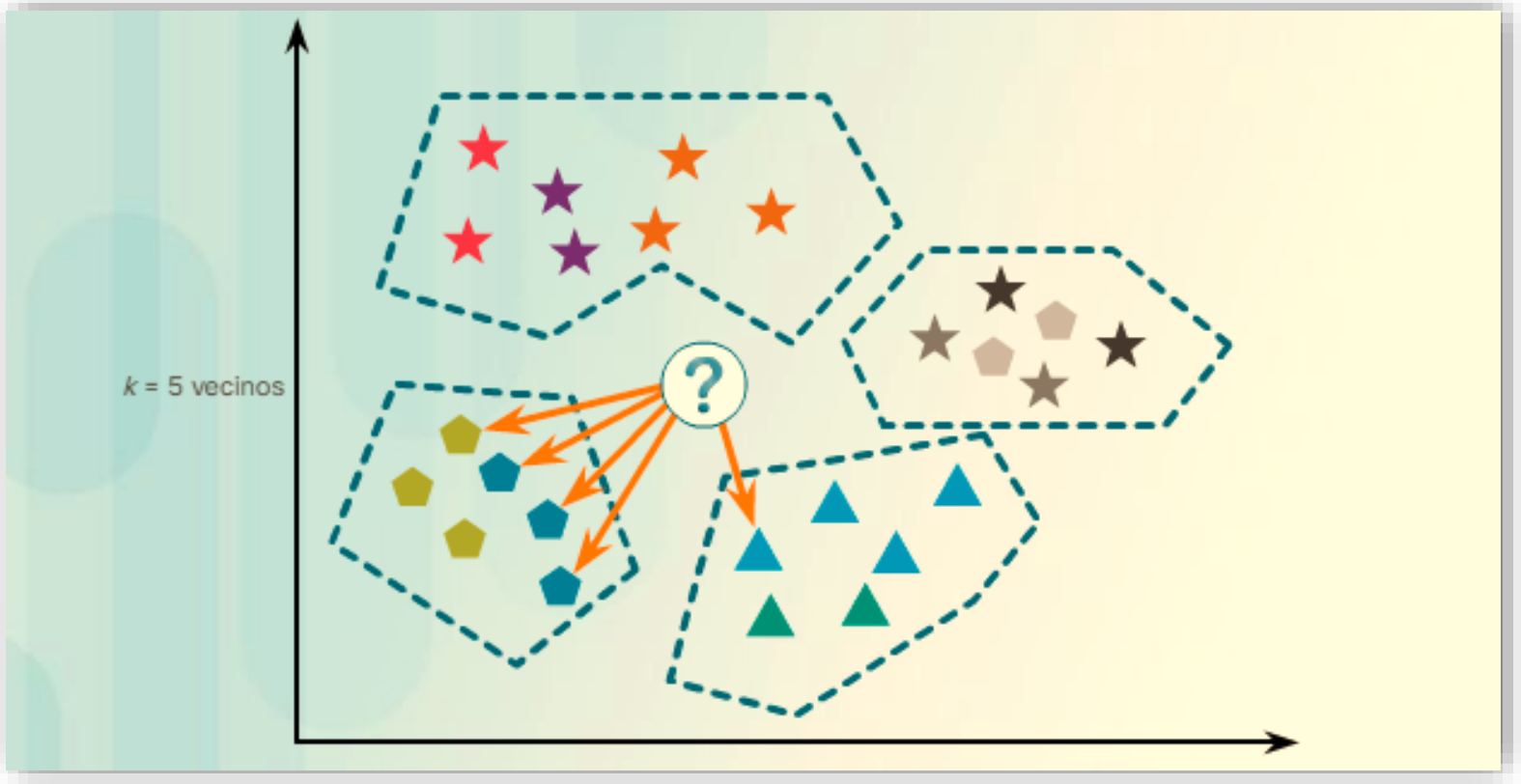
Supervised Machine Learning (técnicas clásicas)

Algoritmo K-nearest neighbor (K vecinos)

K-nearest neighbor (kNN): KNN es posiblemente el **clasificador más intuitivo**, que utiliza la distancia entre los ejemplos de entrenamiento como **medida de similitud**. La distancia entre los puntos representa la diferencia entre los valores de sus funciones. Dado un nuevo punto de datos, un clasificador KNN debe ver los puntos de entrenamiento más cercanos. La **clase predicha** para el nuevo punto será la clase más común entre los K neighbors.

Distancia clásica (euclidea):

$$d_{ij} = \sqrt{\sum_{p=i}^{k} (x_{ip} - x_{jp})^2}$$

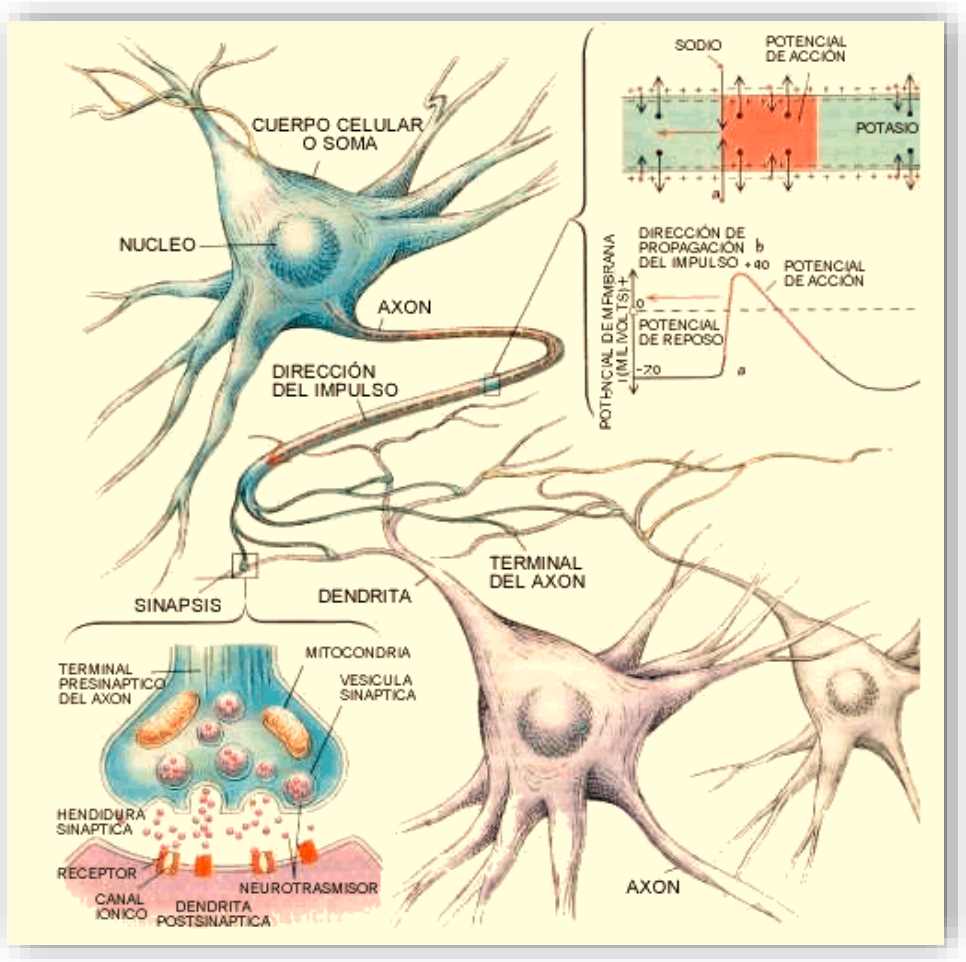


Pasos:

1. Cálculo de distancias
2. Ordenamos las distancias e identificamos los K vecinos
3. Calcular la decisión. En **clasificación** es la clase (función de agregación clásica – el más frecuente). En **regresión** es un valor continuo (función de agregación clásica - el promedio).

Redes Neuronales

- Perceptrón
- Adaline - Funciones de activación
- Función de Costo
- Redes Neuronales y varias capas (MLP)
- Forward y Backpropagation
- Optimización de pesos por backpropagation
- Gradiente descendente
- Gradiente descendente estocástico

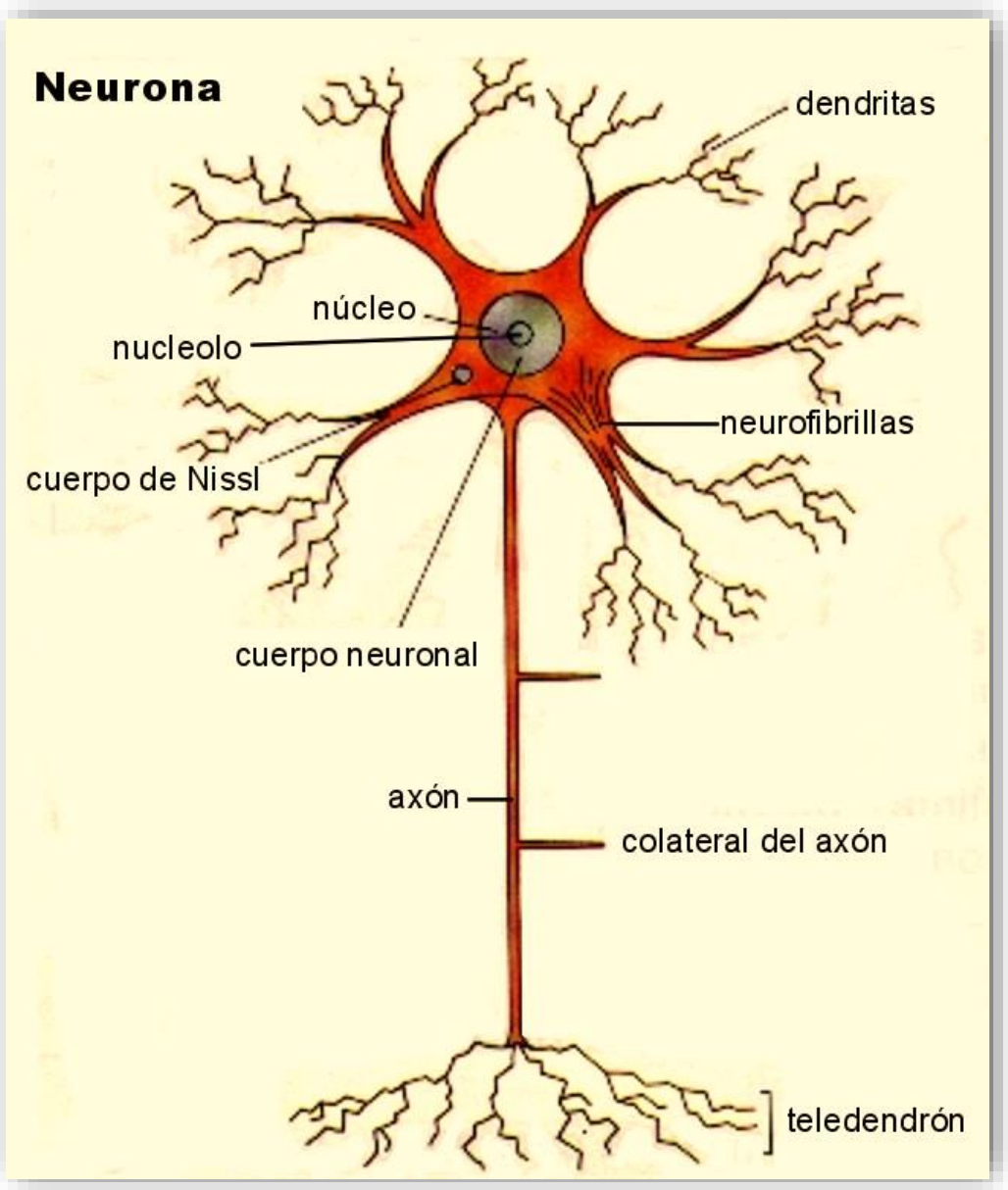


Redes Neuronales: Conceptos

El propósito es intentar imitar el funcionamiento del cerebro humano. Y el propósito de hacer eso posible, es crear una infraestructura para que las máquinas puedan “aprender”.

Las neuronas por si solas son inútiles, son como las hormigas, tienen que vivir en colonia, millones de hormigas pueden armar una colonia, y lo mismo pasa con las neuronas, si trabajan juntas, pueden lograr cosas fantásticas.

Warren McCulloch and Walter Pitts publicaron el primer concepto de una simplificación de una celda del cerebro, la neurona llamada McCulloch-Pitts(MCP), en 1943.

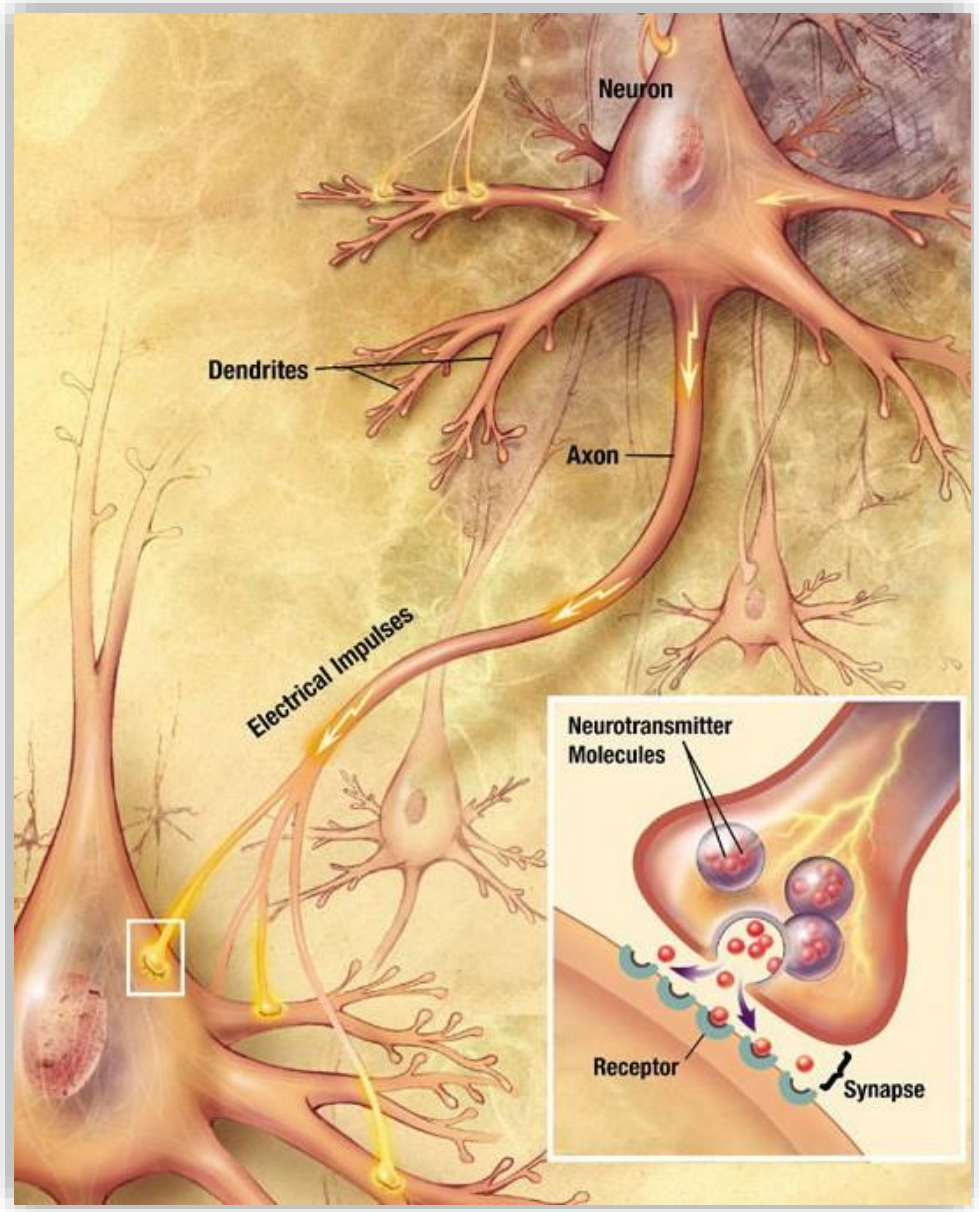


Redes Neuronales: Conceptos

McCulloch y Pitts describieron una neurona con una simple lógica con salida binaria; **múltiples señales de entrada en las dendritas son integradas en el cuerpo de la neurona, y, si la señal acumulada excede cierto umbral (threshold), una señal de salida es generada y será pasada por el axon.**

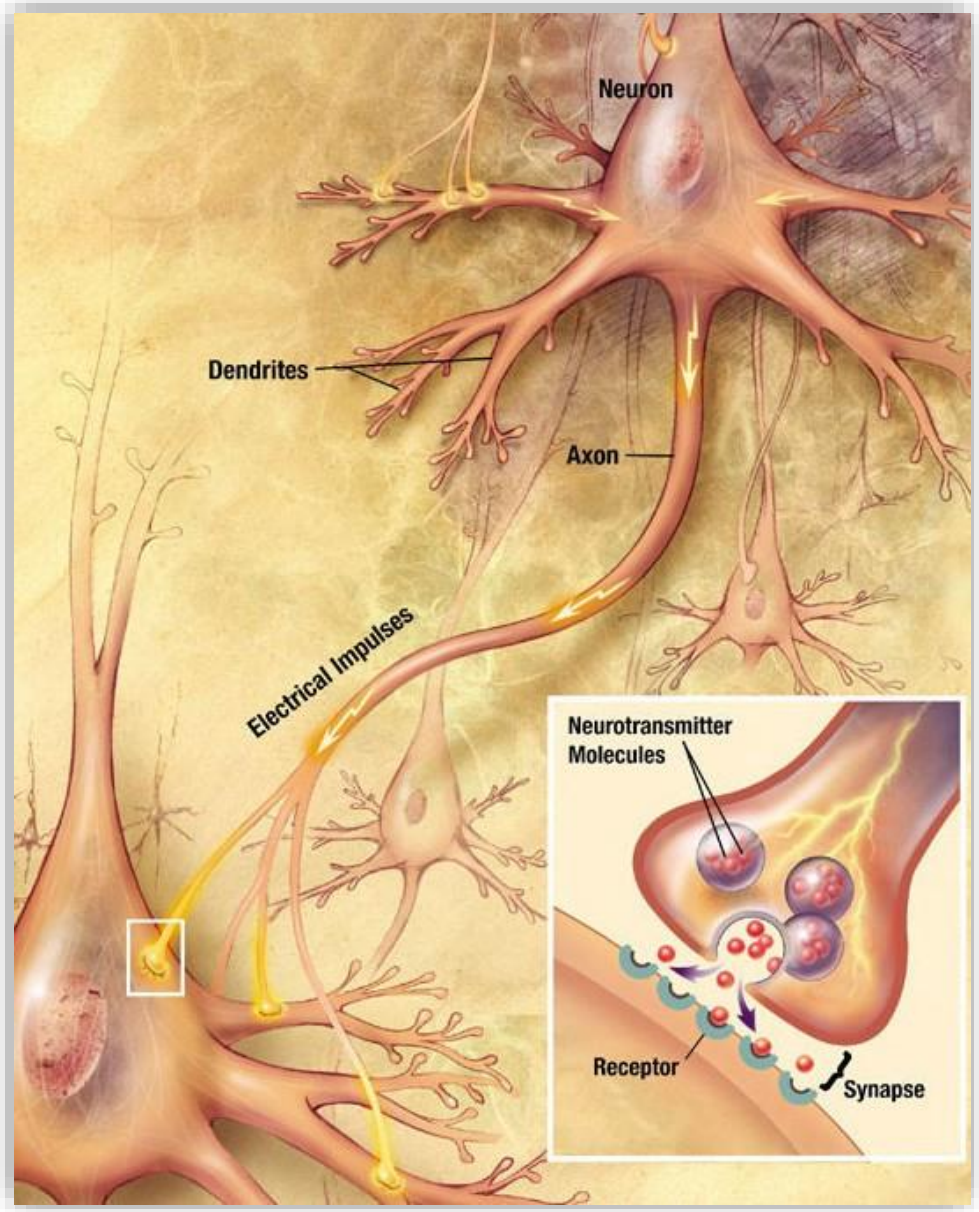
Entonces ¿Las neuronas cómo trabajan juntas?

Las **dendritas** reciben información, y el **axón** transmite esta información (impulsos eléctricos) a las dendritas de la otra neurona. A esta transmisión de información se le llama **sinapsis.** Millones de neuronas forman la red neuronal humana.



Redes Neuronales: Conceptos

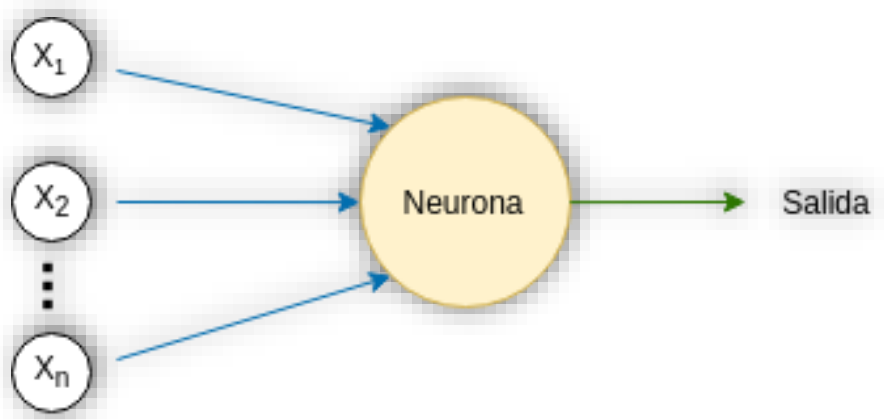
Frank Rosenblatt publicó el primer concepto de la **regla de aprendizaje del Perceptron** basada en el modelo de la neurona MCP. Con esta regla, Rosenblatt propuso un algoritmo que automáticamente aprendería los coeficientes de los pesos que son multiplicados con las características de entrada a fin de decidir si la neurona se enciende o no. En un problema de clasificación, este algoritmo serviría para predecir si una observación pertenece a una clase o a otra.



Redes Neuronales: Conceptos

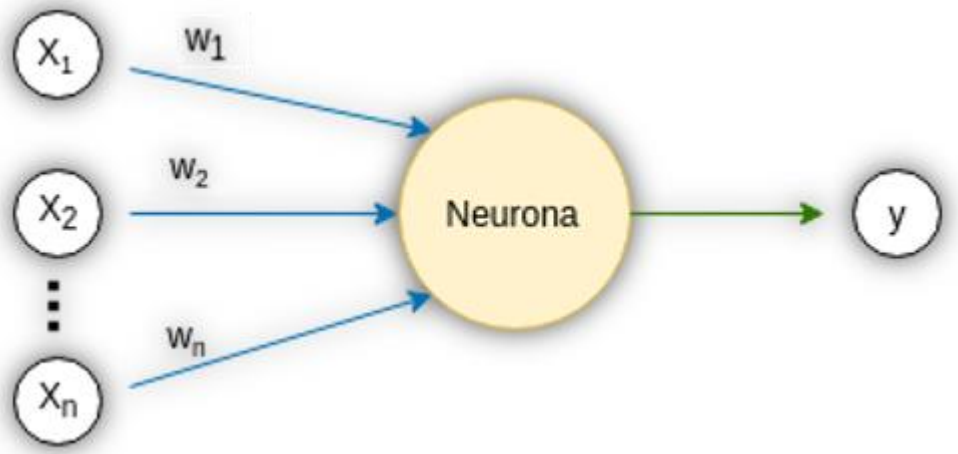
Red Neuronal Artificial (RNA):

Las flechas = **dendritas**
Las X = **información que le pasan otras neuronas**



Entonces:

Flechas = la sinapsis, la señal, un valor de entrada.
A partir de estos valores se generará un valor de salida.



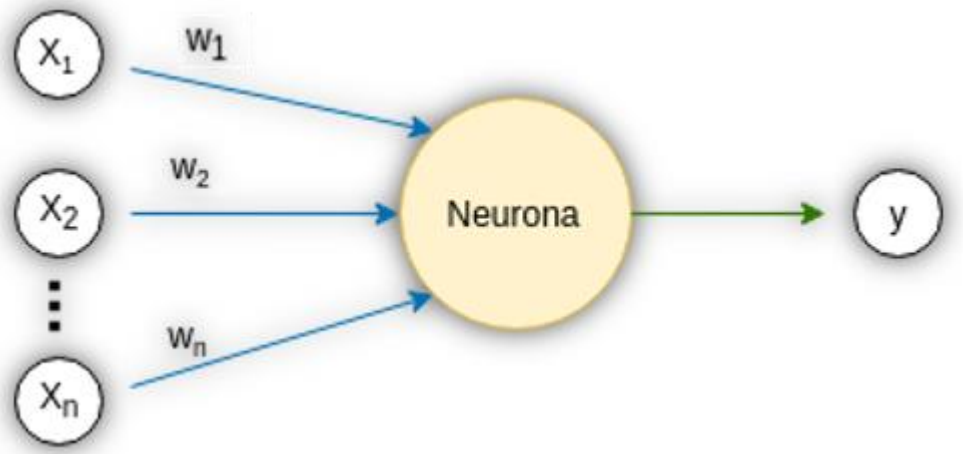
Entradas = variables independientes
Salida = variable dependiente
Las **sinapsis = pesos**

Redes Neuronales: Conceptos

Formalización del Perceptrón:
En una tarea de clasificación binaria por simplicidad tenemos dos clases: 1 (clase positiva) y -1 (clase negativa).
Función de decisión f(z): toma una combinación lineal de valores de entrada x y los pesos correspondientes. Donde z es la entrada de la red.

$$z = \sum_{i=0}^{m} w_i x_i$$

$$f(z) = \begin{cases} 1 \; si \; z \geq \theta \\ -1 \; en \; otro \; caso \end{cases} \qquad f(z) = \begin{cases} 1 \; si \; z \geq 0 \\ -1 \; en \; otro \; caso \end{cases}$$



Redes Neuronales: Conceptos

Regla del Perceptrón:

1. Inicializar los pesos en 0 o pequeños números aleatorios.
2. Por cada observación (sample) de training x:
   - Calcular el valor de salida ŷ -> Es la clase de salida.
   - Actualizar los pesos

$$w_j = w_j + \Delta w_j$$

$$\Delta w_j = n(y - \hat{y})\, x_j$$

Donde y es la etiqueta de clase real, n es la tasa de aprendizaje, n es típicamente una constante entre 0.0 y 1.0. Todos los pesos son actualizados simultáneamente antes de recalcular ŷ. Mientras n es menor, el aprendizaje es más estable.

Si hay una incorrecta clasificación los pesos se empujan hacia la dirección de la clase objetivo positiva o negativa.

Escenarios de ejemplo para entender la regla de aprendizaje:

Si hay una correcta clasificación:

$$\Delta w_j = n(1 - 1)\, x_j = 0$$

$$\Delta w_j = n(-1 - (-1))\, x_j = 0$$

Si hay una incorrecta clasificación:

Predicción ŷ = -1 Real y = 1

$$\Delta w_j = n(1 - (-1))\, x_j = n(2)\, x_j$$

$$\Delta w_j = 1\ (1 - (-1))\ 0.5 = (2)\ 0.5 = 1$$

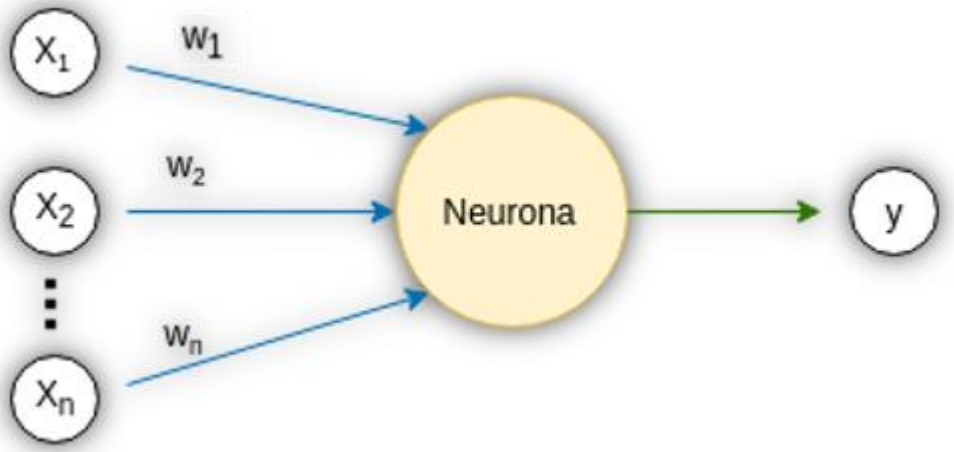
$$\Delta w_j = 1\ (1 - (-1))\ 2 = (2)\ 2 = 4$$

Entonces, el peso es proporcional a x_j para corregir y clasificar correctamente.

Redes Neuronales: Conceptos

Convergencia:

La convergencia es garantizada si las dos clases son linealmente separables y la tasa de aprendizaje es pequeña. Si las dos clases no pueden ser linealmente separables por un umbral de decisión, entonces, necesitaremos un número máximo de pasos sobre el conjunto de entrenamiento (epochs) y/o límite de cantidad de clasificaciones incorrectas toleradas, caso contrario, la red nunca pararía de actualizar los pesos (bucle infinito).



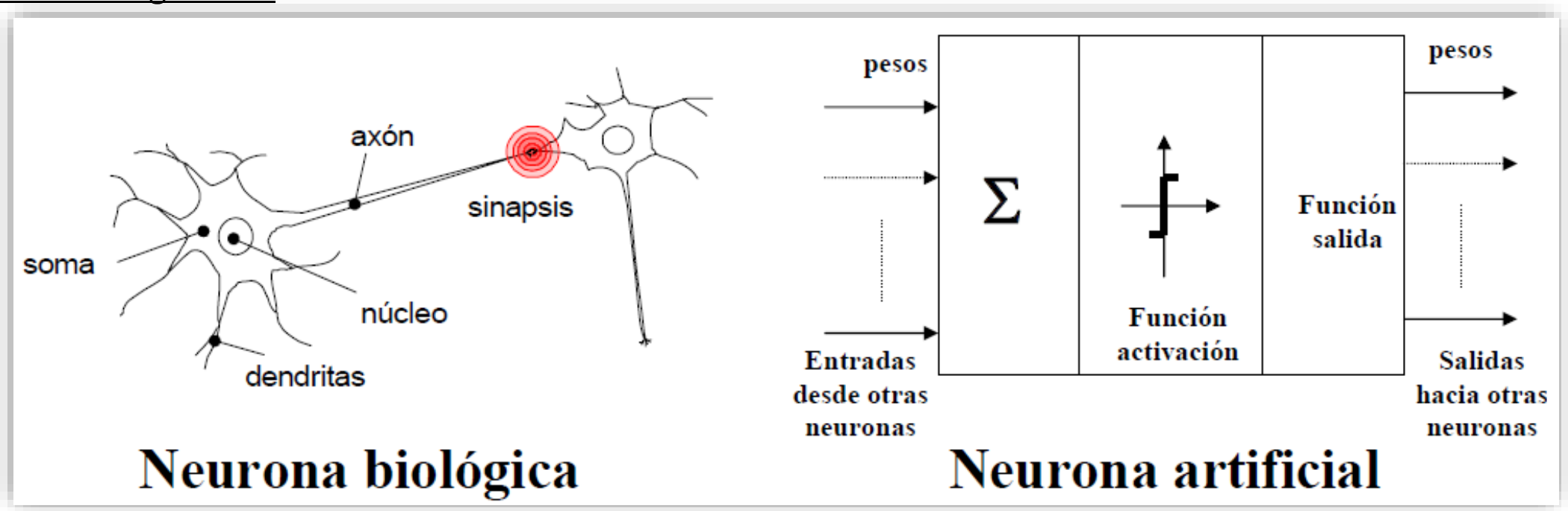
Redes Neuronales: Conceptos

Adaptive Linear Neuron (Adaline): por Bernard Widrow y Tedd Hoff

Perceptrón: ajuste de pesos basado en “unit step function”.

Adaline: ajuste de pesos basado en “linear activation function”.

Adaline ilustra los conceptos clave de definir y minimizar la función de costo. Esto sienta las bases para entender algoritmos avanzados de ML para clasificación, como regresión logística, support vector machines, y modelos de regresión.



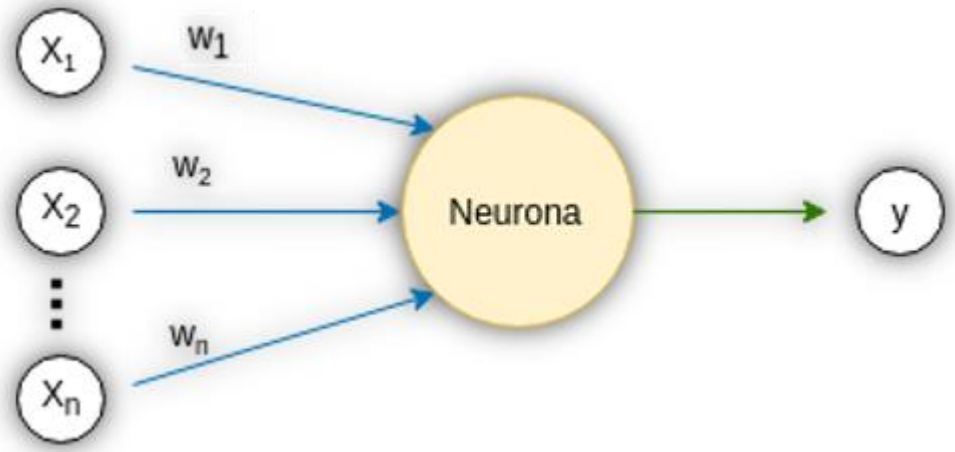
Redes Neuronales: Conceptos

Aprendizaje:

Los “pesos” son los componentes cruciales dentro de una red neuronal artificial, porque las redes neuronales artificiales aprenden ajustando estos valores.

Estos valores deciden en cada caso que señal es importante, o cual no, que señal pasa o que señal no. **Los pesos se ajustan para obtener los resultados que esperamos.**

Pasos:
(1) Realizar una suma ponderada de variables y pesos:
(2) Aplicar sobre el resultado una “función de activación”



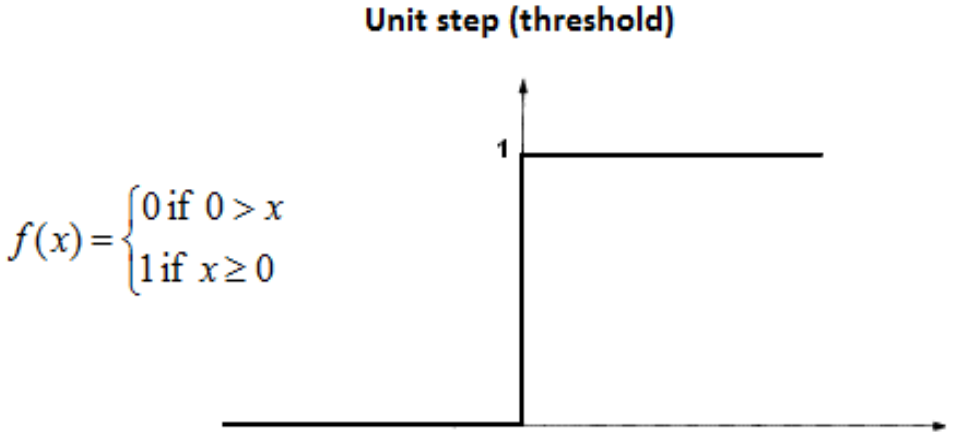
$$z = \sum_{i=0}^{m} w_i x_i \qquad (1)$$

$$f(z) = \varphi\left(\sum_{i=0}^{m} w_i x_i\right) \qquad (2)$$

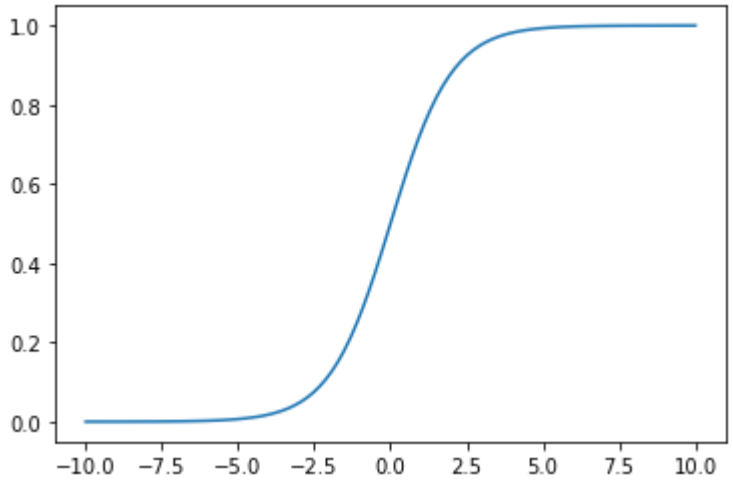
Redes Neuronales: Funciones de Activación

Aprendizaje:

La función de activación: determina el comportamiento de las salidas de cada capa, es decir, escoge que valor pasa o no pasa. Saber escoger una función de activación es fundamental.
Funciones: Threshold, Sigmoide, Rectificadora e Hiperbólica.
Función Threshold: las salidas son binarias. Sí o No. Encendido o Apagado.
Función Sigmoide: es una curva mas suave que va de 0 a aprox. 1 pasando por sus valores intermedios. Es bueno tener esta función en la capa de salida cuando queremos calcular alguna probabilidad.



Función Threshold



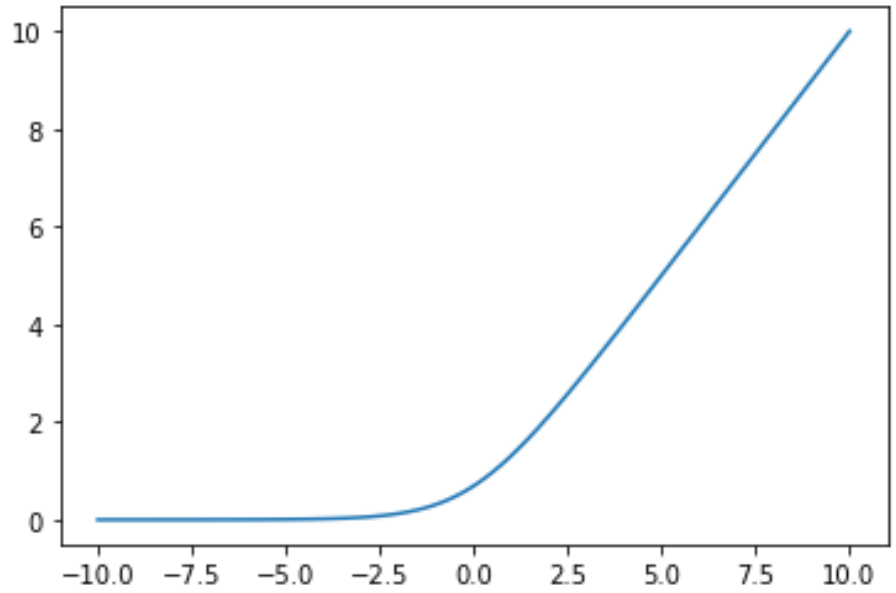
$$P(t) = \frac{1}{1 + e^{-t}}$$

Función Sigmoide

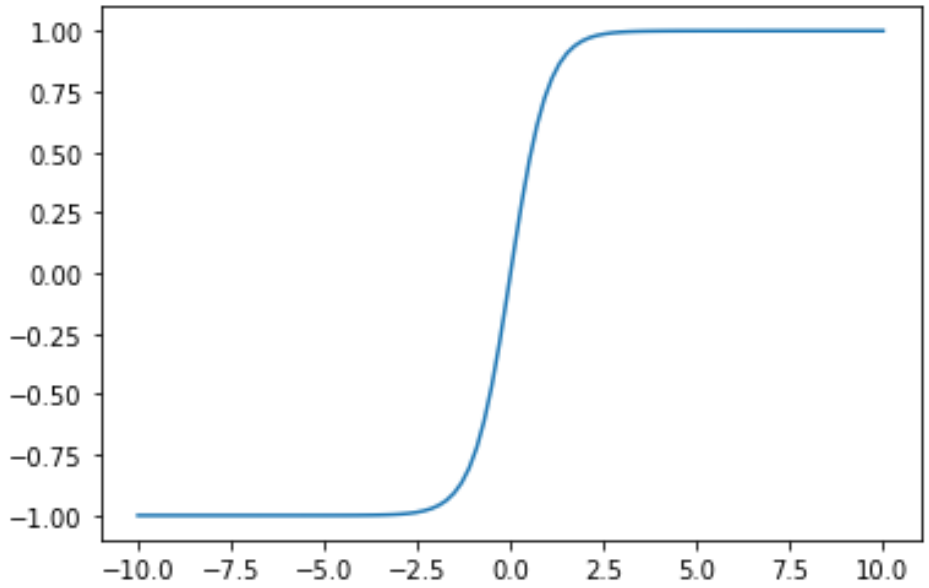
Redes Neuronales: Funciones de Activación

Aprendizaje:

Función Rectificadora: simplemente devuelve el número si es mayor a cero o si no devuelve cero.
Función Hiperbólica: es similar a la sigmoide pero los valores van de aprox. -1 a aprox. 1.



Función Rectificadora

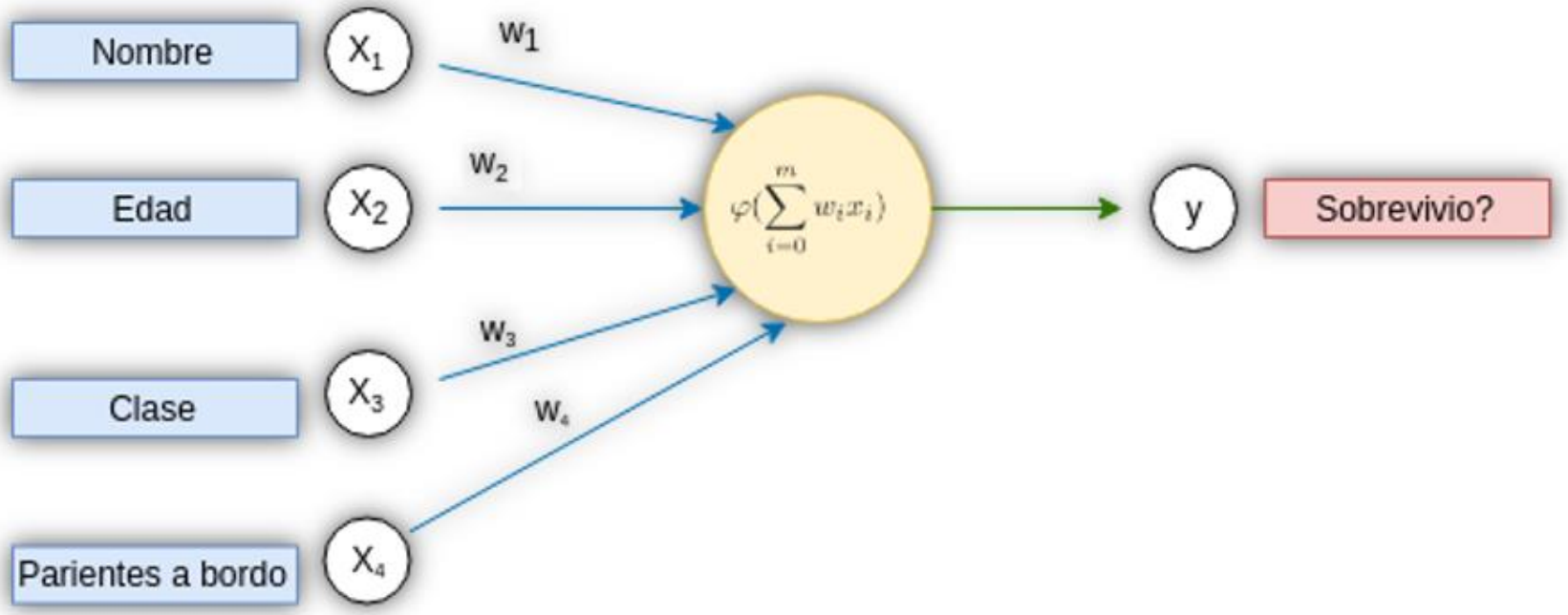


Función Tangencial Hiperbólica

Redes Neuronales: Funciones de Activación

Aprendizaje - Funcionamiento

Las X se llaman **variables independientes**, porque no dependen de ni un factor para ser calculadas, y las Y se llaman **variables dependientes** porque dependen de X para ser calculadas.
En este caso el resultado será muy fácil de calcular, mediante la suma ponderada con la activación.

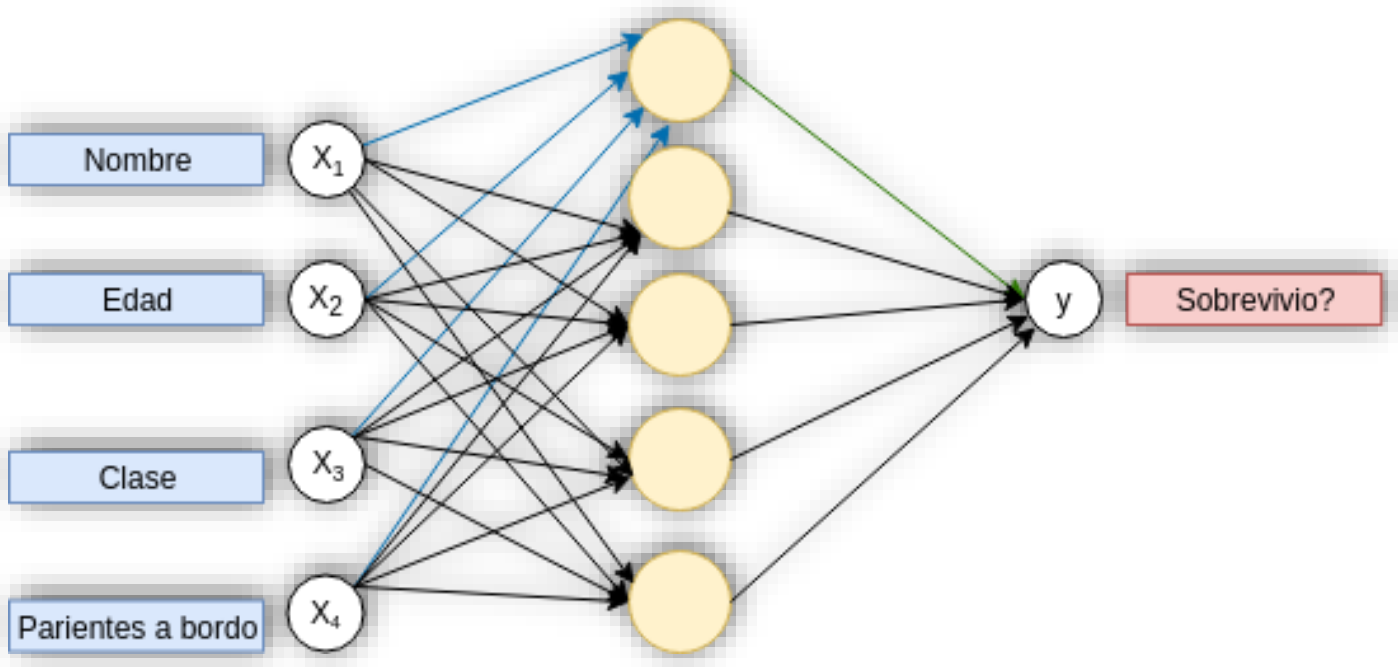


Redes Neuronales: Funciones de Activación - Varias Capas

Aprendizaje – Funcionamiento – Con más capas

Todas las neuronas de la capa anterior irán conectadas a todas las neuronas de la capa siguiente, lo que **aumenta la capacidad de predecir un valor**, mejora mucho el modelo, pero **no se garantiza mejores resultados por tener más capas intermedias**. Mientras más capas intermedias agreguemos, más nivel de abstracción tendrá nuestra red.

Algo que hay que notar es que no todas las neuronas analizan los mismos valores de X, **por las funciones de activación algunos de los valores podrían ser descartados, porque no son significantes para el rendimiento de la red neuronal.** Y de esta forma todas estas neuronas se dividen la información encargándose de una tarea distinta, pero esto es solo una analogía, no siempre es así.

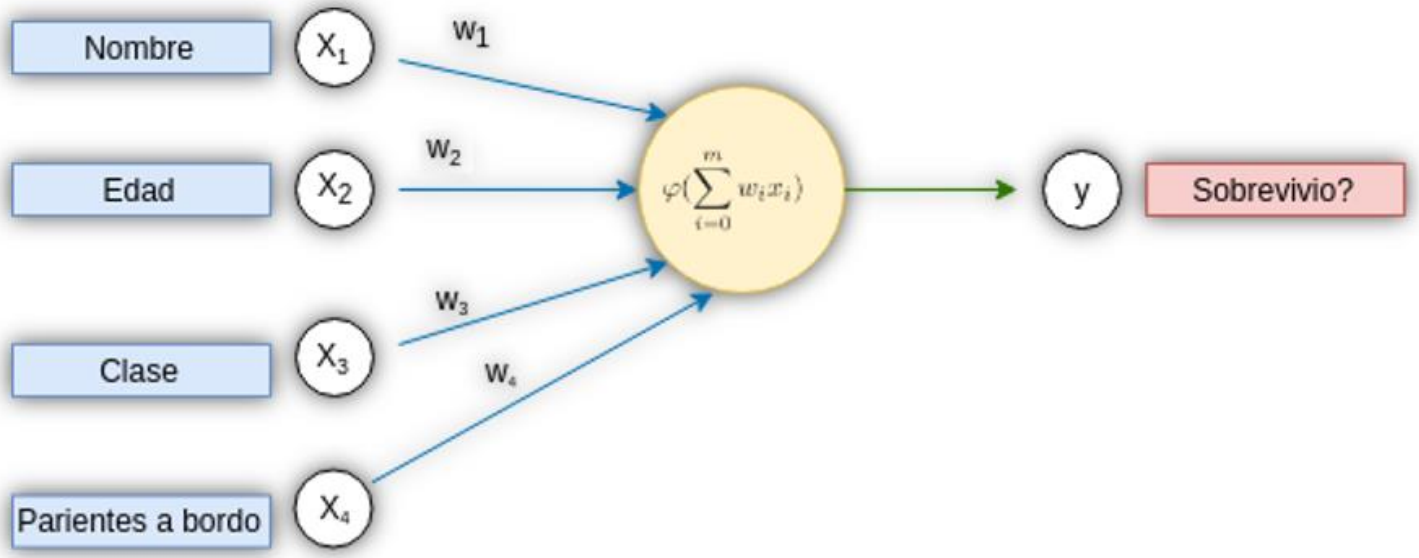


Redes Neuronales: Predicción

Aprendizaje

Las redes neuronales son una herramienta que tiene la ventaja de que puede **“entender” los datos que uno le presenta, y hacer una función que se ajuste a nuestro X**. Ejemplo: tenemos imágenes de perros y gatos, la red neuronal va a extraer todos los patrones de la imagen que diferencian a un perro y un gato, así cuando proporcionemos una nueva imagen de un perro, la red neuronal con todas las reglas que aprendió podrá “entender” de que animal se trata.

“Predicción de Y”: se predice el valor de Y por cada epoch (época).



Redes Neuronales: Función de Costo (o conocida como función de pérdida)

Evaluación

Al final de cada epoch, el modelo compara las predicciones de Y con los valores reales de Y, y esta comparación se hace a través de la función de pérdida. Esta función indica cuánto es el margen de error en nuestra red, y nuestro objetivo es minimizar esa función, es decir, que el margen este lo mas cercano posible a cero. La función más popular es MSE.

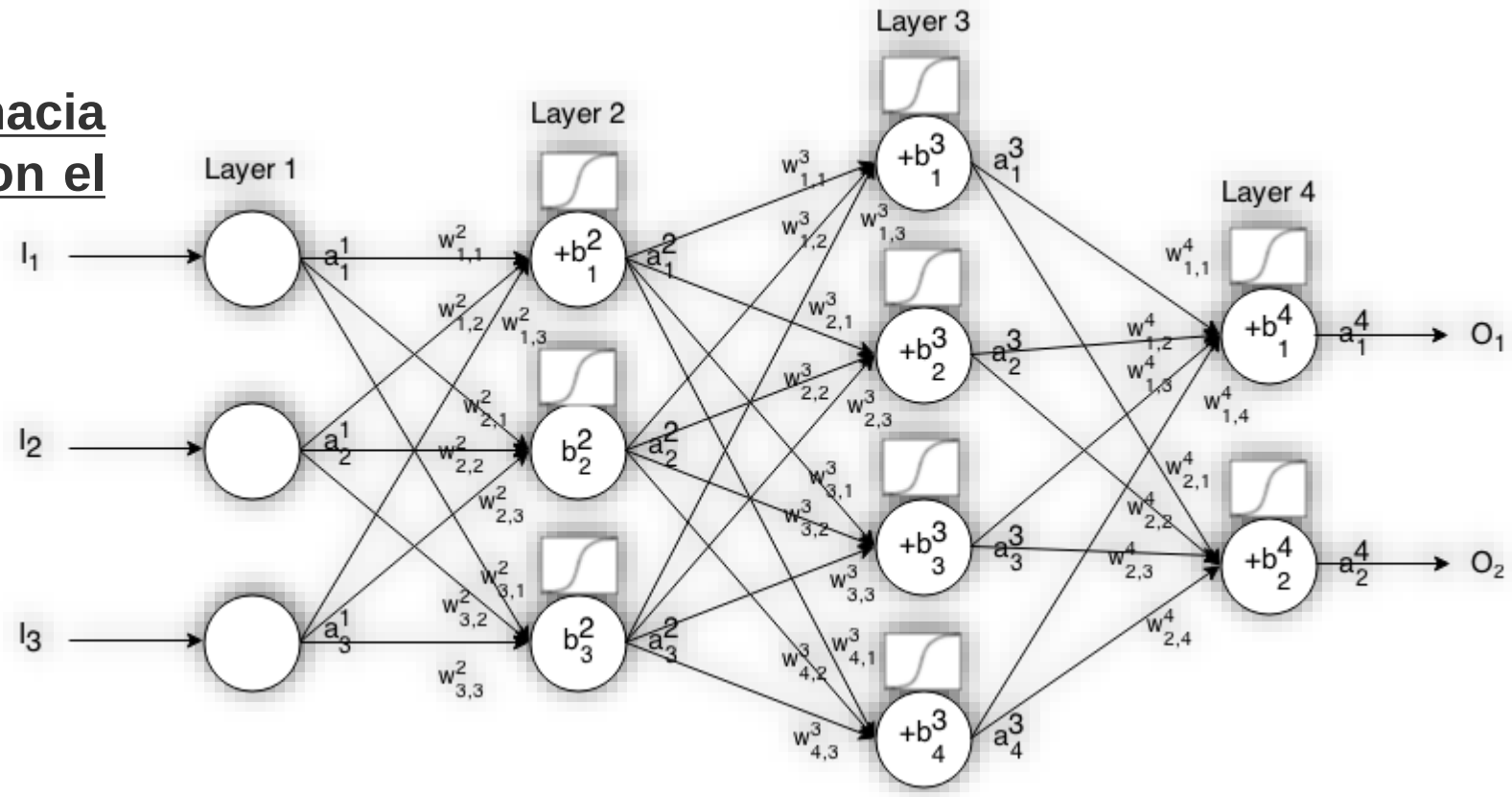
Esta función de error va hacia atrás, y se ajustan los pesos con el fin de minimizar la pérdida.

$$L = \frac{1}{2}(\hat{y} - y)^2$$

$\hat{y}$: Valor predecido

y : Valor real

El término ½ es añadido para que sea más fácil derivar la función de costo.



Redes Neuronales:
- Forward y Backpropagation
- Optimización de pesos por Backpropagation

Forward Propagation: un valor de entrada pasa por toda la red hasta la salida, y se calcula la función de perdida, luego de este proceso, la pérdida se debe propagar hacia atrás (**Backpropagation**) ajustando los valores de los pesos. El **Backpropagation** es un algoritmo avanzado basado en matemáticas muy interesantes y sofisticadas, y nos permite **ajustar todos los pesos simultáneamente**.

Todos los pesos de la red se deben ajustar al mismo tiempo, para que los valores sean consistentes, esta es la clave para que el algoritmo funcione.

Redes Neuronales: Descenso de gradiente

¿Cómo se minimiza la función de pérdida?

Fuerza bruta = literalmente cambiando todos los valores de los pesos para obtener algo parecido a una parábola. Y después de eso revisar los resultados y ver el resultado mas bajo.

Problema de Dimensionalidad -> Ejemplo: 1000 pesos (valor bajo) -> Sunway Taihulight: La super computadora mas rápida del mundo, tardaría exp(10, 50) años en hacer fuerza bruta.

Por lo tanto, la fuerza bruta, no es una opción.

Aquí es donde entra el **descenso de gradiente**.

La **idea fundamental del gradiente descendente** es ir hacia abajo de una colina hasta alcanzar un costo mínimo local o global. *En cada iteración damos un paso en la dirección opuesta del gradiente, donde el tamaño de paso es determinado por el valor de una tasa de aprendizaje (n), o denominada "slope".*

Redes Neuronales: Actualización de pesos

Donde y es la etiqueta de clase real, **n** es la tasa de aprendizaje, n es típicamente una constante entre 0.0 y 1.0. Todos los pesos son actualizados simultáneamente antes de recalcular ŷ. Mientras n es menor, el aprendizaje es más estable.

$$w_j = w_j + \Delta w_j$$

$$\Delta w_j = n(y - \hat{y})\, x_j$$

Si hay una incorrecta clasificación los pesos se empujan hacia la dirección de la clase objetivo positiva o negativa.

Escenarios de ejemplo para entender la regla de aprendizaje:

Si hay una correcta clasificación:

$$\Delta w_j = n(1 - 1)\, x_j = 0$$

$$\Delta w_j = n(-1 - (-1))\, x_j = 0$$

Si hay una incorrecta clasificación:

Predicción ŷ = -1 Real y = 1

$$\Delta w_j = n(1 - (-1))\, x_j = n(2)\, x_j$$

$$\Delta w_j = 1\ (1 - (-1))\ 0.5 = (2)\ 0.5 = 1$$

$$\Delta w_j = 1\ (1 - (-1))\ 2 = (2)\ 2 = 4$$

Entonces, el peso es proporcional a x_j para corregir y clasificar correctamente.

Redes Neuronales: Descenso de gradiente

¿Cómo se minimiza la función de pérdida?

Algoritmo ML: función matemática con entradas y parámetros
Objetivo: ajustar los parámetros de tal forma que se pueda alcanzar aproximadamente otra función objetivo.

Para ello, necesitamos saber como la salida de la función ML cambia cuando cambiamos algún parámetro -> **Apoyo del cálculo diferencial**
El cálculo diferencial se ocupa de la tasa de cambio de una función con respecto a una variable (parámetro) de la que depende la función.
Podemos aproximar cómo cambia f(x) con respecto a x para cualquier valor de x al calcular la pendiente de la función para ese valor. Si la pendiente es positiva, la función incrementa, y la función decrementa si la pendiente es negativa. La inclinación de la pendiente indica la variación de cambio de la función para el valor.

Para un parámetro x, se calcula la pendiente con las siguientes fórmulas:

$$pendiente = \frac{\Delta y}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Lim es el concepto matemático de el límite (Δ aproxima a 0).
Y f'(x) y dy/dx son las notaciones Lagrange y Leibniz para deriviadas, respectivamente.

$$f'(x) = \frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Para n parámetros x, se utiliza derivadas parciales:

$$\frac{\partial f}{\partial x_i} = \lim_{\Delta x_i \to 0} \frac{f(x_1, x_2, \dots, x_i + \Delta x_i, \dots, x_n) - f(x_1, x_2, \dots, x_i, \dots, x_n)}{\Delta x_i}$$

$$\frac{\partial \mathbf{F}}{\partial x_i} = \begin{pmatrix} \frac{\partial F_1}{\partial x_i} \\ \frac{\partial F_2}{\partial x_i} \\ \vdots \\ \frac{\partial F_m}{\partial x_i} \end{pmatrix}$$

Redes Neuronales: Descenso de gradiente

Ecuación forma canónica de la parábola:

$$y = a(x-h)^2 + k$$
$$y = (x-h)^2$$

- a: coeficiente de apertura pos (+)
- (h,k): coordenadas del vértice
- k=0

$$cost = \sum (t-y)^2$$

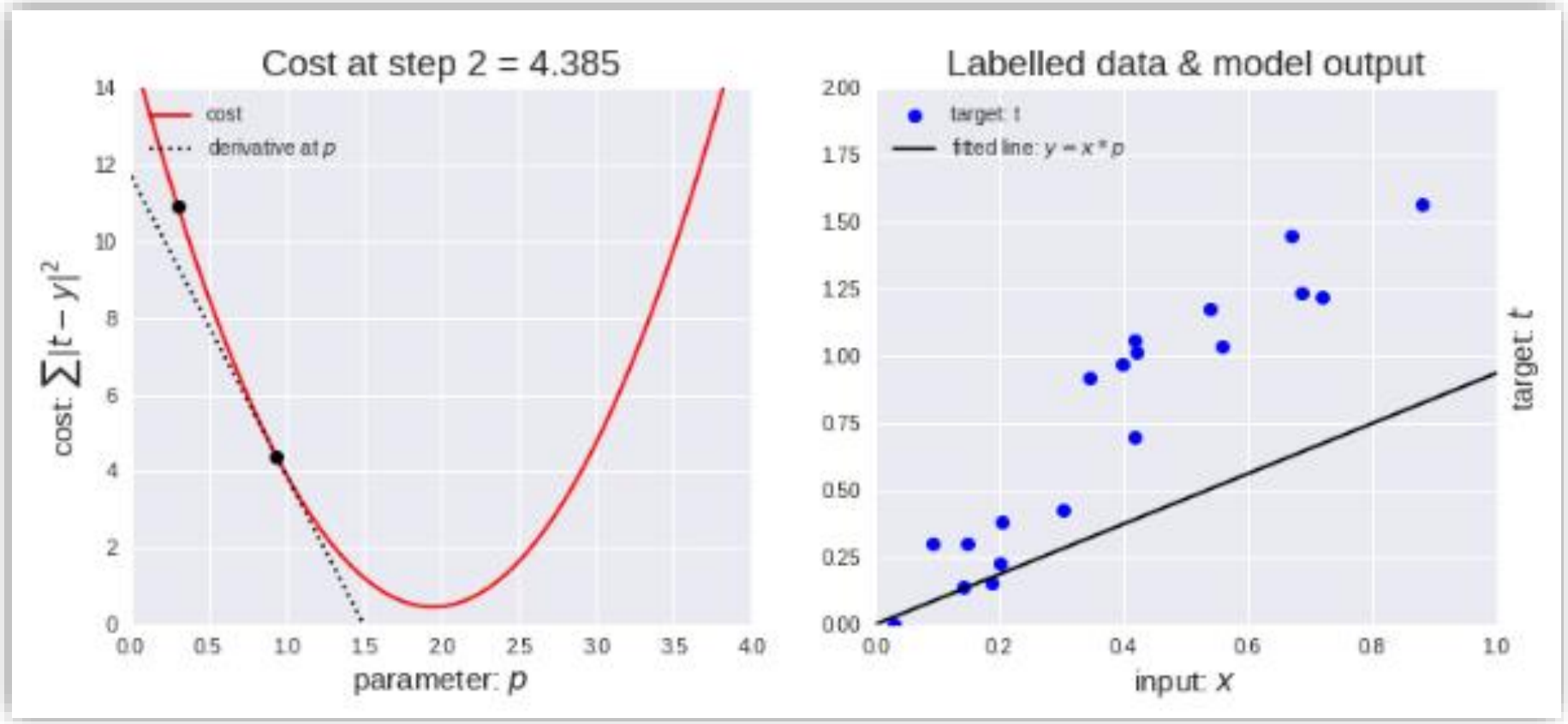
$$y = x * p$$
$$predicción = valorentrada * peso$$

$$y : predicción$$
$$t : real(target)$$
$$x : input$$
$$p : peso$$

Derivada en p: derivada de y, cuando el peso tiene el valor de p.

¿Cómo se minimiza la función de pérdida?

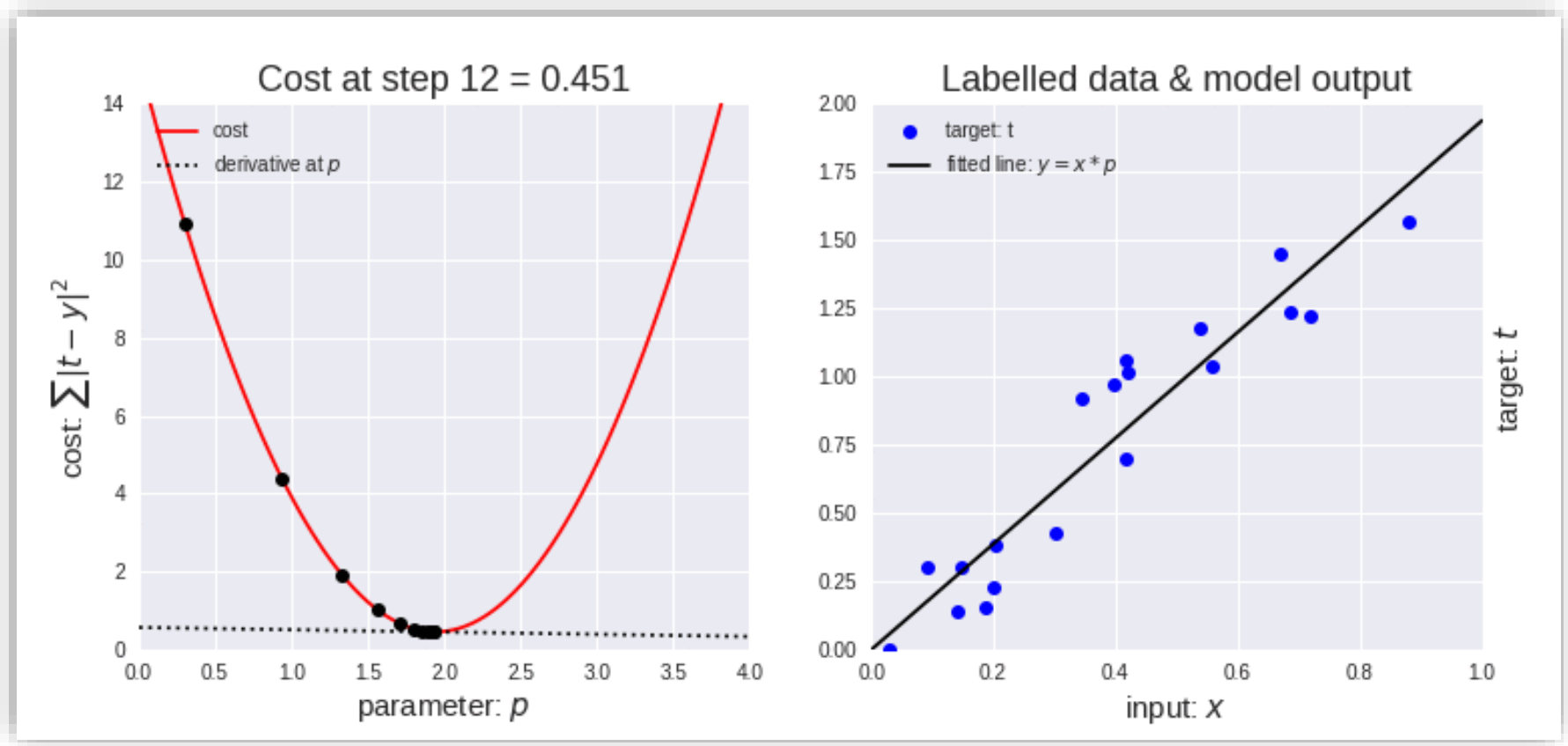
En cualquier punto de nuestra función se calcula la pendiente del punto, si la pendiente es negativa (ver figura), ajusta los pesos después de cada epoch, para ir hacia la derecha (positivo). Si se pasa del mínimo (pendiente positiva), ajusta los pesos para ir hacia la izquierda (atrás), hasta que encuentra los mínimos locales.



Redes Neuronales: Descenso de gradiente

¿Cómo se minimiza la función de pérdida?

Este método nos permite empezar en cualquier punto de nuestra función y calcula la pendiente del punto en el que esta, si la pendiente es negativa, ajusta los pesos después de cada epoch, para ir hacia abajo, si se pasa, ajusta los pesos para ir hacia el otro lado, hasta que encuentra los mínimos locales.



Redes Neuronales

Regla de Adaline (o regla Widrow-Hoff): Formalización

1. Inicializar los pesos en 0 o pequeños números aleatorios.
2. Por cada *sample, batch o epoch* de training:
   - Calcular el valor de salida ŷ → $forward\ propagation$
   - Calcular función de pérdida $J(w)$
   - Actualizar los pesos → $back\ propagation$

$$f(z)= \varphi\left(\sum_{i=0}^{m} w_i x_i\right) \qquad w_j = w_j + \Delta w_j$$

$$J(w) = 1/2 \sum_i (y^{(i)} - f(z^{(i)}))^2$$

La función sigmoide es continuamente diferenciable, y su derivada convenientemente en función de la propia función sigmoide es: f′(z)=f(z)(1−f(z)).

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

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

$$\sigma'^{(x)} = \sigma(x).(1 - \sigma(x))$$

Redes Neuronales

$$z = \left(\sum_{i=0}^{m} w_i x_i\right)$$

Regla de Adaline (o regla Widrow-Hoff): Formalización

1. Inicializar los pesos en 0 o pequeños números aleatorios.
2. Por cada *sample, batch o epoch* de training:
   - Calcular el valor de salida ŷ -> predicción -> $forward\ propagation$
   - Calcular función de pérdida $J(w)$: $para\ medir\ calidad\ y\ visualizar$
   - Actualizar los pesos → $back\ propagation$

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

$$J(w) = 1/2 \sum_i (\boldsymbol{y^{(i)} - f(z^{(i)})})^2$$

$$w_j = w_j + \Delta w_j$$

$$f'(z) = f(z)(1 - f(z))$$

en capa que no es salida: $\Delta w_j = \eta\, \delta_j\, x_j \longrightarrow \delta_j = f'(z) \sum_{k=0}^{m} \delta_k w_k$

en capa de salida: $\Delta w_j = \eta\, \delta_j\, x_j \longrightarrow \delta_j = f'(z)\ \boldsymbol{(y - f(z))}$

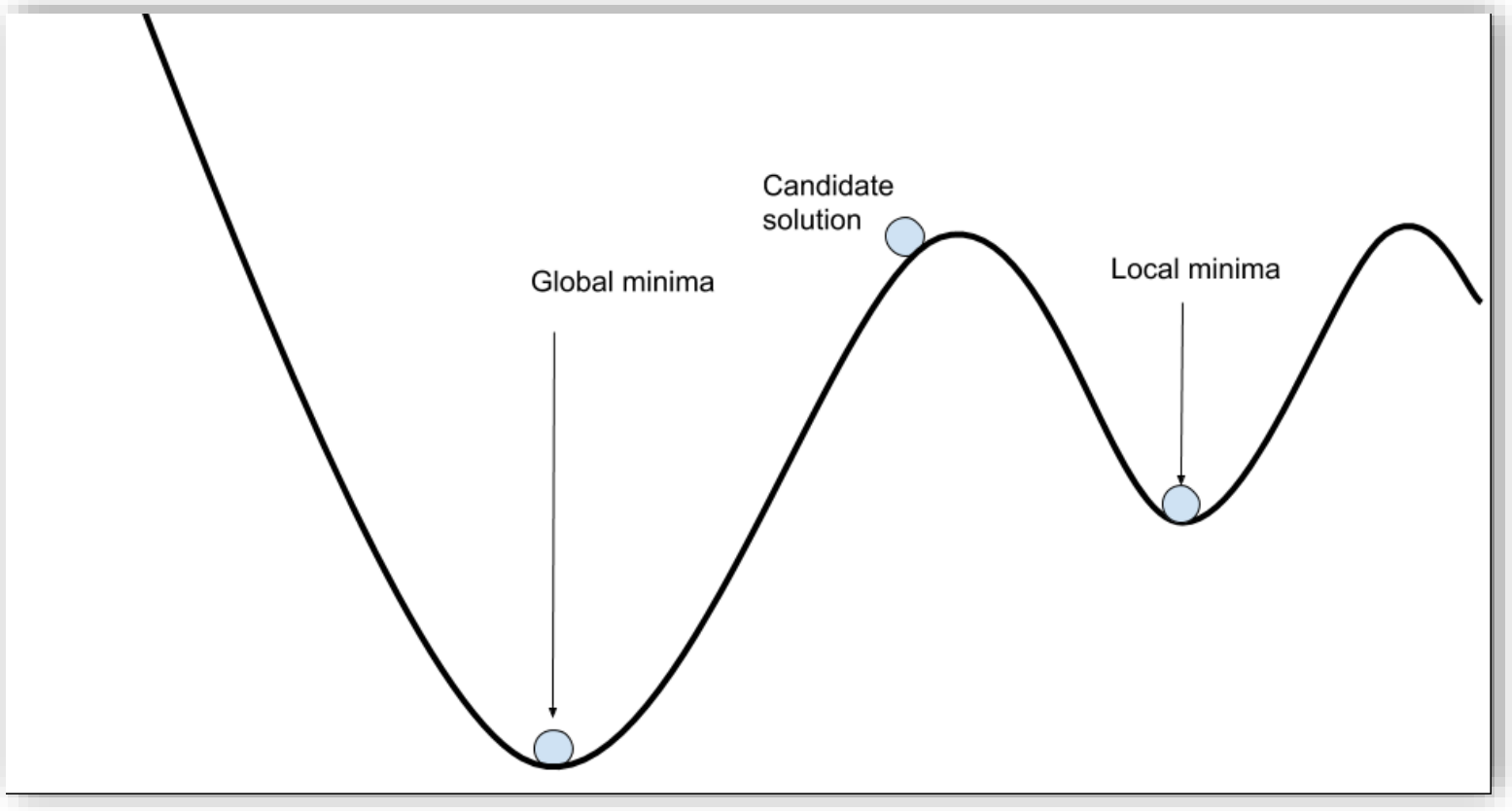
Donde **y** es la salida real, $f(z)$ es la función de activación que da las predicciones, **n** es la tasa de aprendizaje (amplitud de paso). Se actualizan los pesos en la dirección opuesta del gradiente. El ajuste de peso es $\boldsymbol{\Delta w_j}$. δ o ∇ es el gradiente (delta) que se obtiene con la derivada parcial, es decir, es la tasa de cambio o pendiente de la función de transferencia (función de activación), **m** es la cantidad de neuronas de la capa **de la derecha**, *j* hace referencia a una neurona de la capa actual y *k* hace referencia a una neurona de la capa siguiente. El **gradiente en una capa intermedia** es igual a la derivada de la salida de la capa por la sumatoria de productos entre gradientes y pesos de las neuronas de la capa de derecha.

Redes Neuronales: Descenso de gradiente estocástico

Podemos quedar atascados en el **mínimo local**, por lo que la función no será optima, porque no estamos encontrando el mínimo real (**Mínimo Global**). Un mínimo local no sirve para predecir nuestros resultados.
El método de **descenso de gradiente** ajusta los pesos, después de **cada epoch**.
El método de **descenso de gradiente estocástico**, lo hace **cada vez que se pasa un dato**.

Descenso de gradiente:
Paso el dato 1
Paso el dato 2
Paso el dato 3
Ajusto los pesos

Descenso de gradiente estocástico:
Paso el dato 1
Ajusto los pesos
Paso el dato 2
Ajusto los pesos
Paso el dato 3
Ajusto los pesos



Redes Neuronales: Descenso de gradiente estocástico

Características:

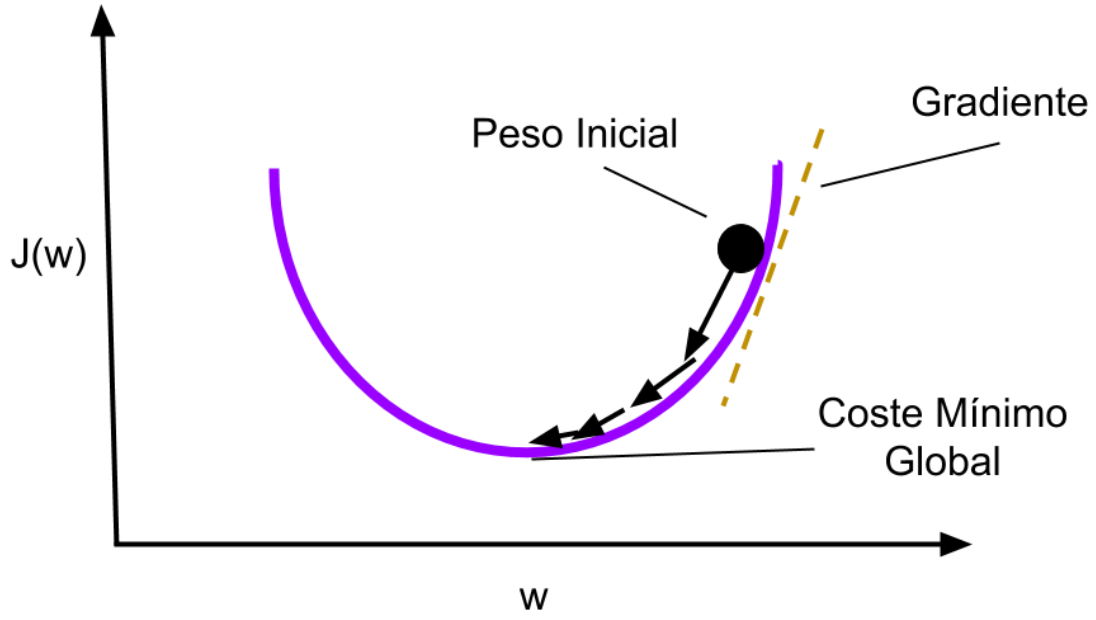
- Se localiza los mínimos globales ajustándose cada vez en ubicaciones distintas, entonces la **fluctuación sobre la función será mucho mas alta**, por esta razón el algoritmo **encontrará el mínimo global a base de estar en movimiento constante sobre la misma función.**

- El método **es rápido**, por la manera en que funciona se tiende a pensar que no, porque se ajusta por cada dato, pero en realidad como **no tiene que cargar todos los datos en memoria para ajustar los pesos y esperar hasta recorrer todas las filas de nuestros datos. Al ir uno por uno, es un algoritmo más liviano (menos uso de memoria) y más rápido (menos uso de CPU).**

Redes Neuronales: efecto de la tasa de aprendizaje (n)

Una tasa de aprendizaje bien escogida permitirá que el error decremente gradualmente, moviéndose en la dirección del mínimo global. Mientras que con una tasa de aprendizaje muy alta, el error se alejaría del mínimo global. *La tasa de aprendizaje toma valores entre 0.0 y 1.0, mientras menor es mejor.*

En el gradiente descendente estocástico esta tasa es a menudo reemplazada por una tasa de aprendizaje adaptativa que decrementa sobre el tiempo (por ejemplo, por el número de iteraciones).



Redes Neuronales: optimizadores

Optimizador	¿En qué consiste?	¿Cuándo se debe usar?
Adam	- Combina el momentum y RMSprop.	- Ampliamente utilizado y efectivo en la mayoría de los casos. - Buen rendimiento en problemas de aprendizaje profundo. - No requiere ajuste fino de hiperparámetros.
RMSprop	- Ajusta la tasa de aprendizaje adaptativamente basada en la media cuadrática de los gradientes pasados.	- Es útil para problemas no estacionarios. - Funciona bien en una amplia gama de aplicaciones de aprendizaje profundo.
SGD (Gradiente Descendente Estocástico)	- Ajusta los pesos utilizando el gradiente de la función de pérdida en cada paso.	- A menudo utilizado como base para comparar otros optimizadores. - Útil con conjuntos de datos grandes y problemas más simples. - Puede requerir una búsqueda más minuciosa de hiperparámetros.
Adadelta	- Variante de RMSprop que elimina la necesidad de ajustar manualmente la tasa de aprendizaje inicial.	- Útil cuando no se quiere preocupar por seleccionar la tasa de aprendizaje. - Funciona bien en problemas de aprendizaje profundo. - Robusto ante problemas con tasas de aprendizaje fijas.

Importantes para diseño

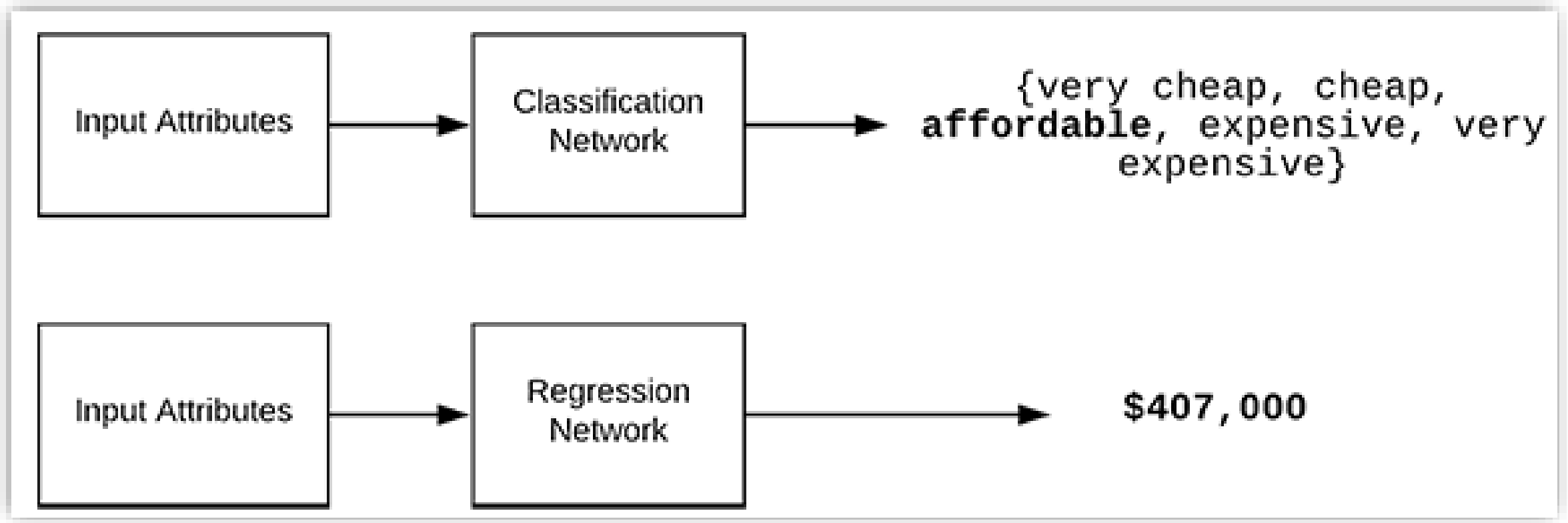
- Inicializar un modelo usando "**Sequential** class' para implementar la red neuronal feedforward (el avance). Podemos añadir muchas capas como queremos, pero hay **dos condiciones**:
  - La cantidad de atributos de entrada deben ser igual al número de variables (columnas) en el training set (en MINIST, 784 variables o pixeles).
  - El número de neuronas en la capa de salida debe ser igual de etiquetas (clases únicas), esto sería el número de columnas de la matriz resultante al aplicar one-hot-encoded.
- **Optimizador**: gradiente descendente estocástico. La tasa de aprendizaje (n) se ajusta en cada epoch.
- Función de costo (o **loss**):
  - Loss='binary_crossentropy' -> clases: 0 o 1, verdadero o negativo
  - Loss='categorical_crossentropy' -> binario para varias clases (la salida está en varias columnas). MNIST: enteros 0-9 (Pag 439-443) -> **Multiclass. Importante: utilizar función de activación Softmax en la última capa.**
  - Loss='sparse_categorical_crossentropy' -> directo clases en decimal (la salida en una sola columna). MNIST: enteros 0-9-> **Multiclass. Importante: utilizar función de activación Softmax en la última capa.**
- **Verbose**=1 (permite ver el avance) durante el training. Verbose=0 no muestra el avance.
- **Predicción**: Model.predict -> retorna las etiquetas de clase como enteros. print ('primeras 3 predicciones: ', y_train_pred[:3])

Importantes para diseño

Loss function	Usage	Examples	
		Using probabilities *from_logits=False*	Using logits *from_logits=True*
BinaryCrossentropy	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

Clasificación vs Regresión

¿Cuánto pagará una persona por un vehículo?

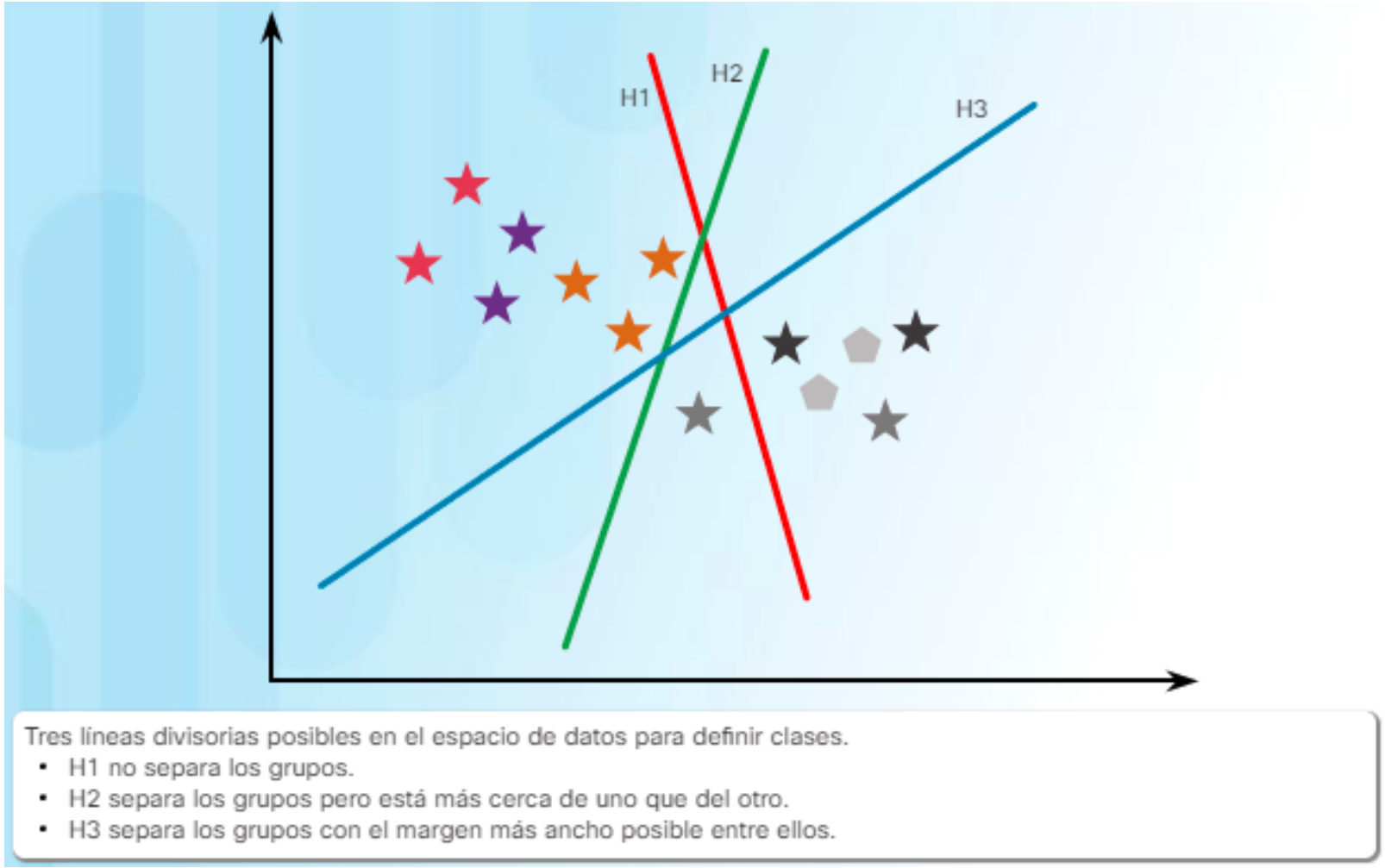


Máquinas de Soporte Vectorial

Introducción
Kernel trick
Problemas de Clasificación usando SVM
Problemas de Regresión usando SVM

Máquinas de Soporte Vectorial (o Máquinas de vectores de soporte): Definición

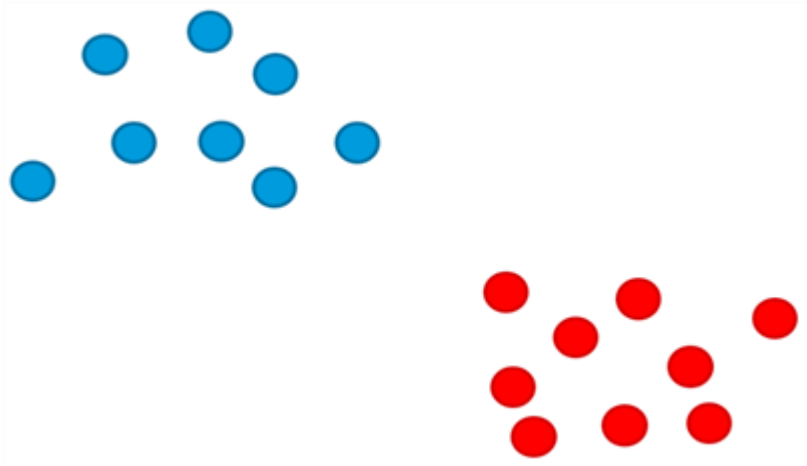
Máquinas de vectores de soporte (SVM): las SVM son ejemplos de **clasificadores** de aprendizaje automático supervisados. En lugar de basar la asignación de membresía de la categoría en distancias de otros puntos, las máquinas de vector de soporte computan la frontera, o el hiperplano, que mejor separa los grupos. En la figura, el H3 es el hiperplano que maximiza la distancia entre puntos de entrenamiento de las dos clases, visibles en color o en blanco y negro. Cuando se presenta un nuevo punto de datos, se clasifica según si se encuentra en un lado o en el otro de H3.



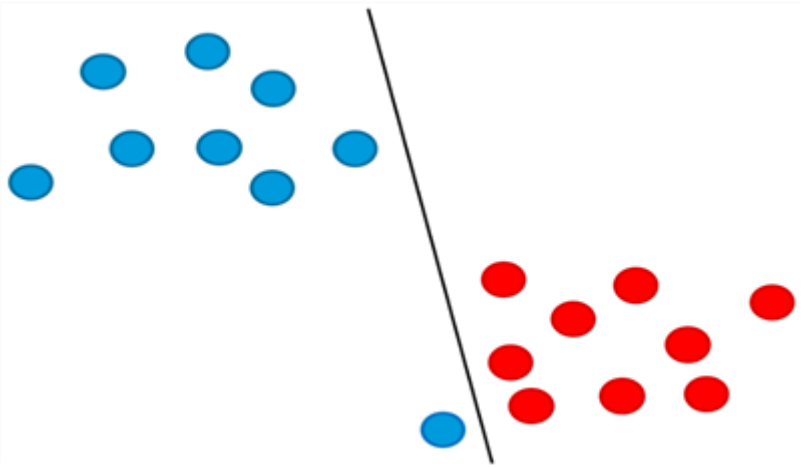
Tres líneas divisorias posibles en el espacio de datos para definir clases.
- H1 no separa los grupos.
- H2 separa los grupos pero está más cerca de uno que del otro.
- H3 separa los grupos con el margen más ancho posible entre ellos.

Máquinas de Soporte Vectorial: Clasificación

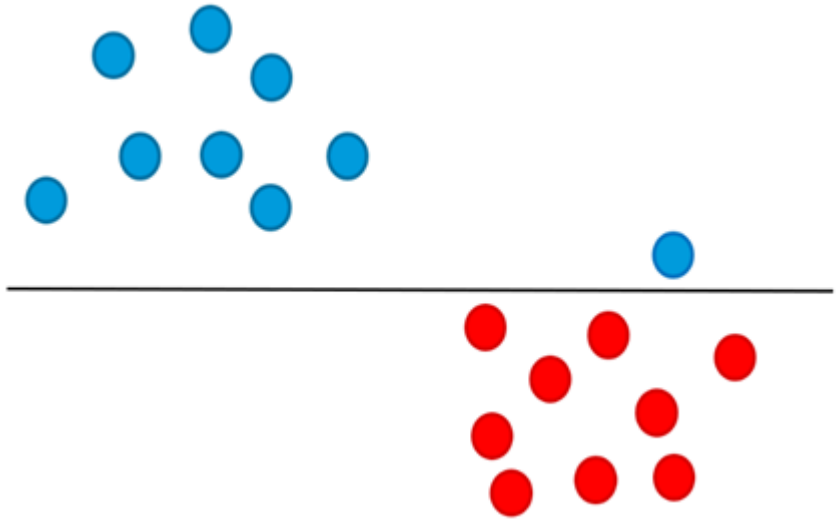
a)



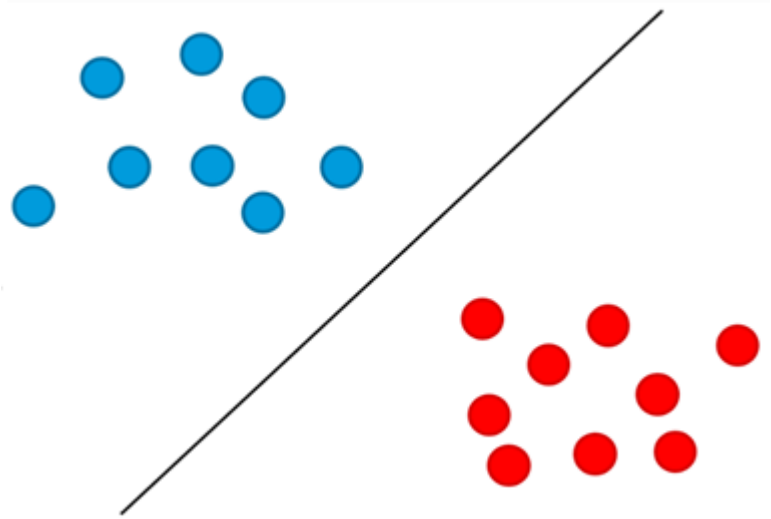
b)



c)



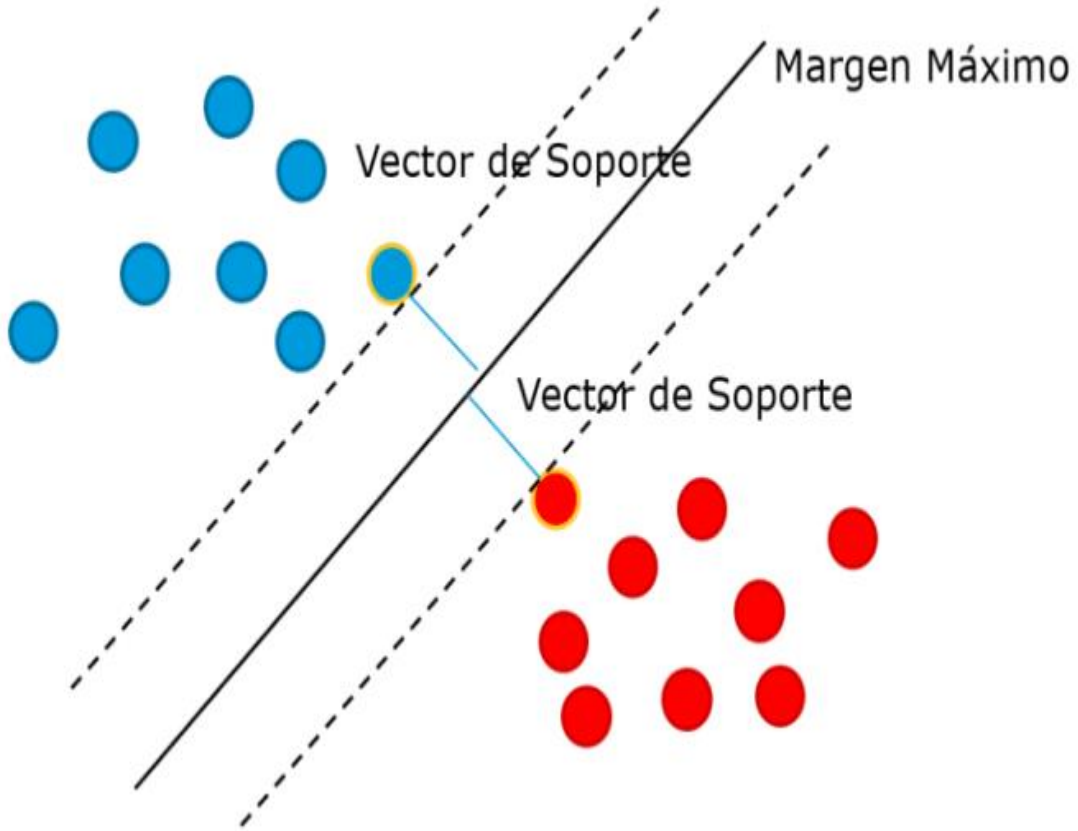
d)



Máquinas de Soporte Vectorial: Clasificación

Los problemas de aprendizaje automático tienen muchísimas dimensiones. Así que en lugar de encontrar la línea óptima, el SVM **encuentra el hiperplano que maximiza el margen de separación entre clases.**

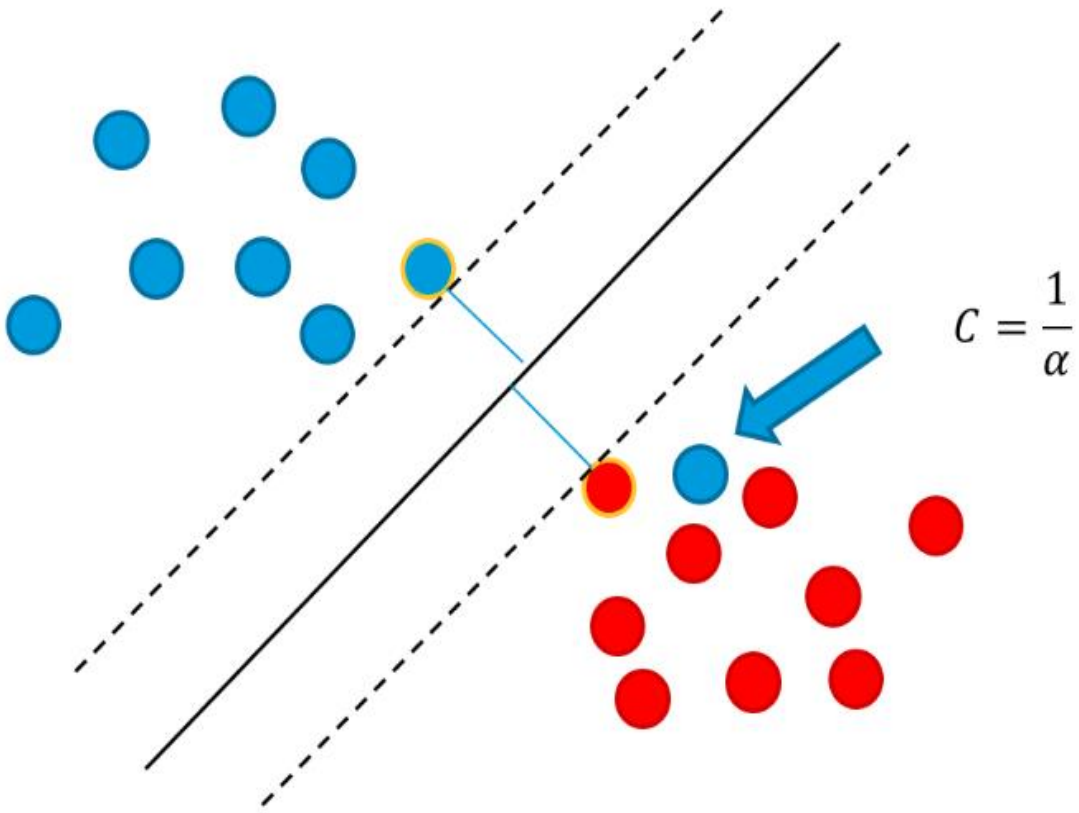
Se llaman "**máquinas**" en español por la parte de machine learning. Los vectores de soporte son los puntos que definen el **margen máximo de separación del hiperplano que separa las clases**. Se llaman **vectores**, en lugar de puntos, porque estos "puntos" tienen tantos elementos como dimensiones tenga nuestro espacio de entrada. Es decir, estos puntos multidimensionales se representan con vector de n dimensiones.



Máquinas de Soporte Vectorial: Clasificación

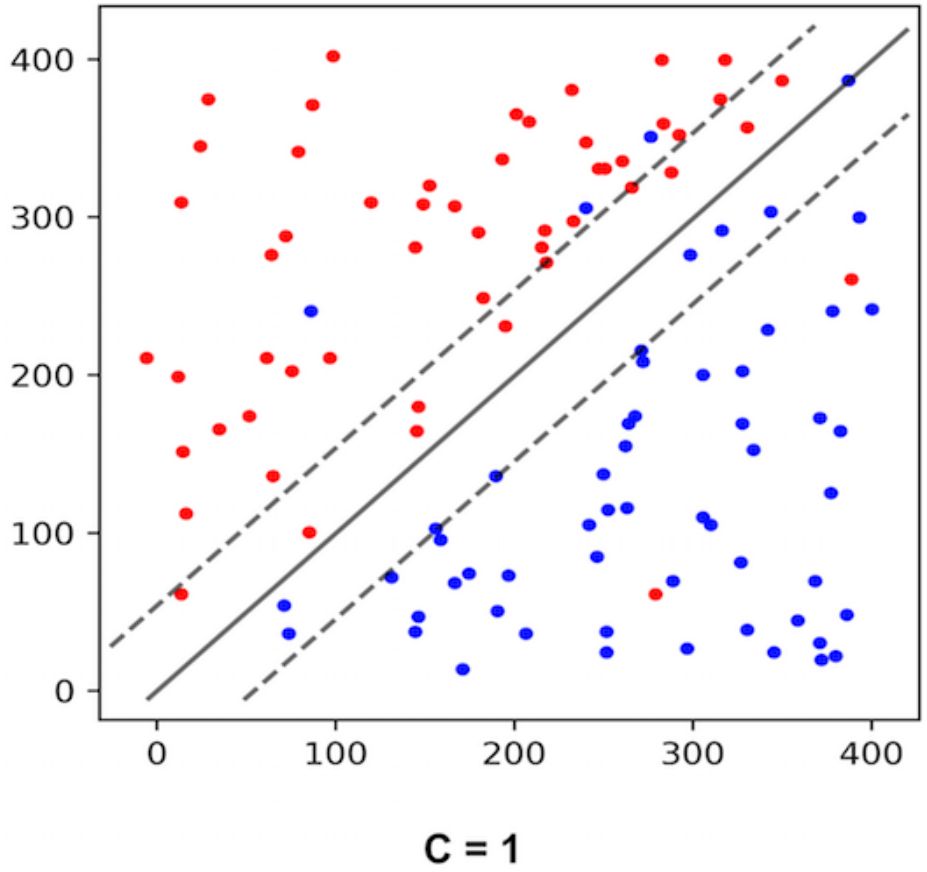
Regularización:

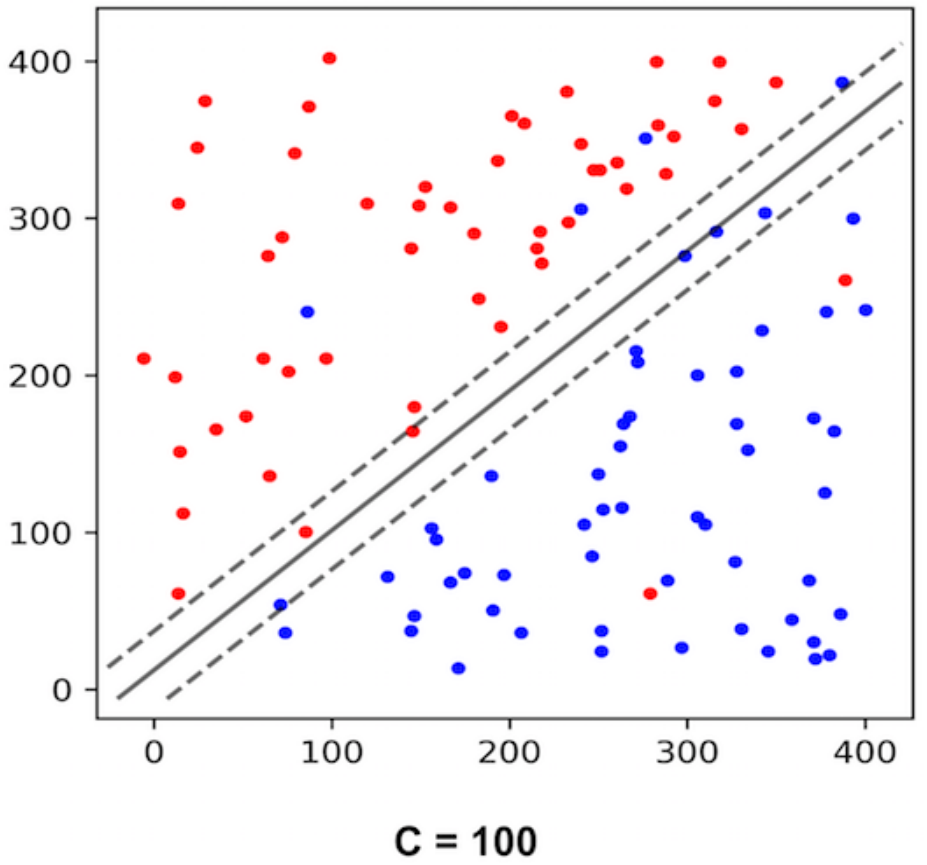
Es bastante frecuente que los datos tengan ruido, que no estén etiquetados perfectamente, o que sea muy complicado clasificar los puntos correctamente. Para estos casos, podemos configurar una SVM para que generalice bien para la mayoría de los casos, aunque algunos pocos casos del conjunto de entrenamiento no estén perfectamente clasificados. Lo que normalmente se busca es la construcción de modelos de aprendizaje automático que generalicen bien. Para **controlar la cantidad de regularización, podemos usar el hiper-parámetro C**. C es la penalización para el error de training. Si se incrementa C, el modelo se ajusta más al conjunto de training, pero puede llegar a un overfitting. El parámetro alfa o épsilon es un **threshold**, no hay penalización si la predicción esta debajo del umbral (distancia).



Máquinas de Soporte Vectorial: Clasificación

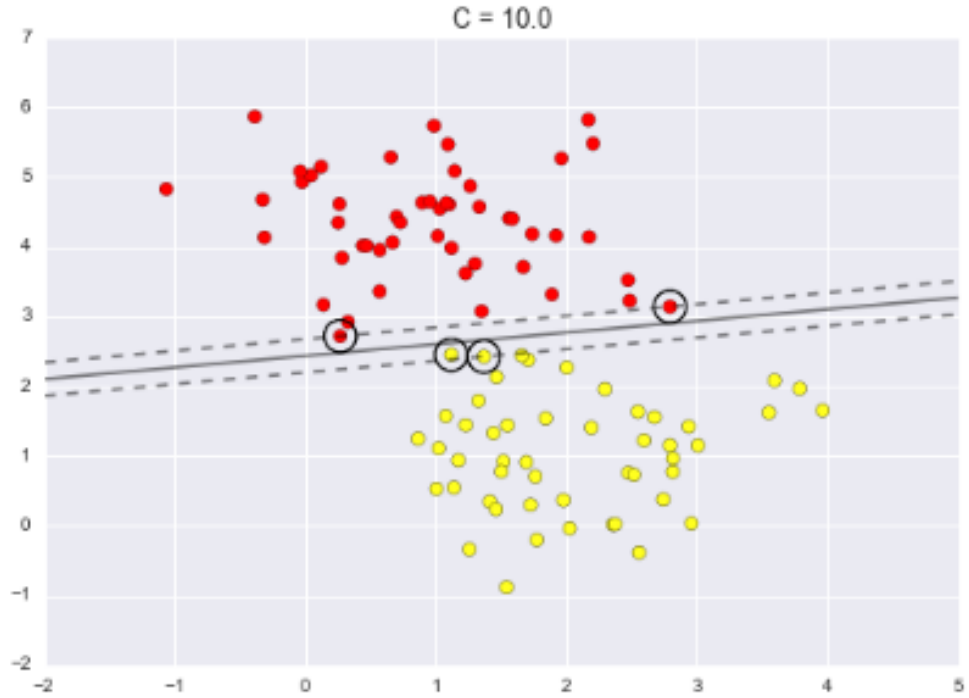
Regularización:

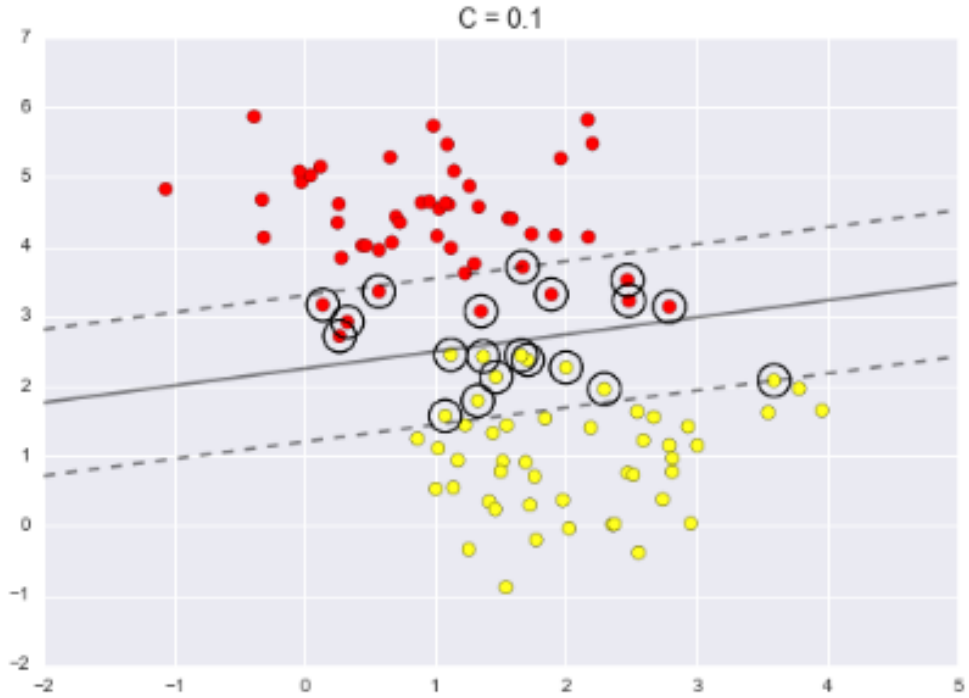




Máquinas de Soporte Vectorial: Clasificación

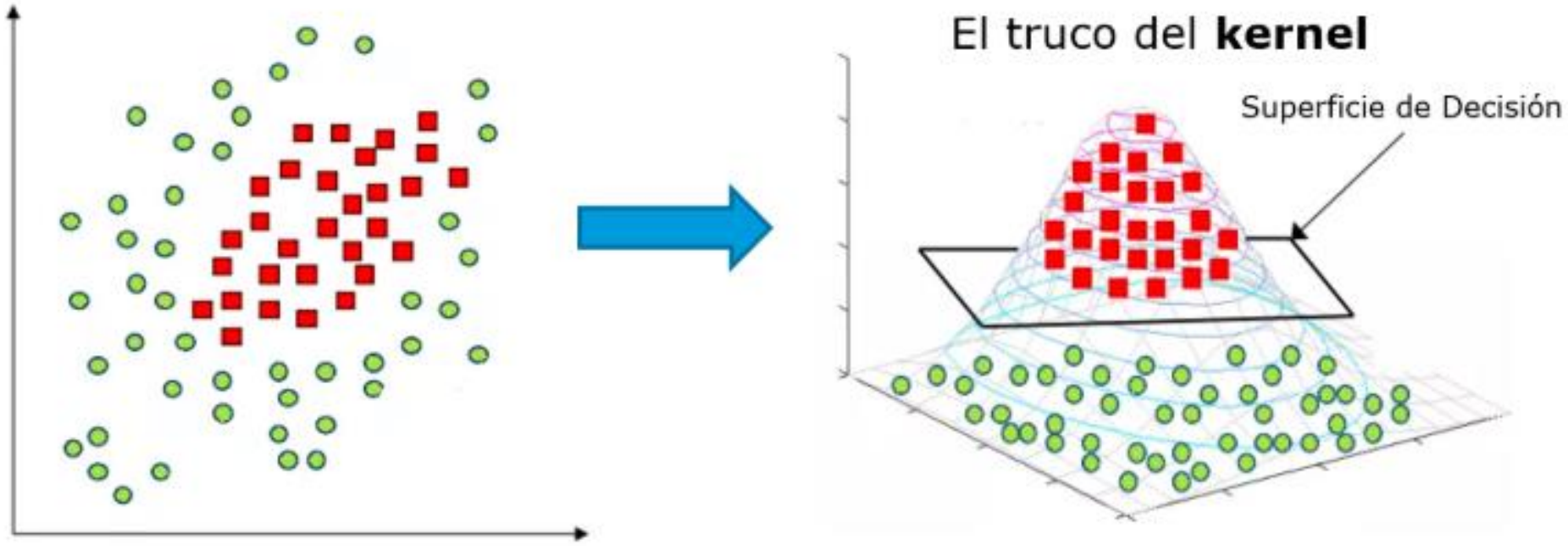
Regularización:





Máquinas de Soporte Vectorial (SVM): kernel trick

SVM es una técnica que permite trabajar con límites de decisión no lineales. Hay situaciones en las que no hay forma de encontrar una hiperplano que permita separar dos clases. En estos casos decimos que las clases no son linealmente separables. Para resolver este problema podemos usar el truco del kernel. **El truco del kernel consiste en inventar una dimensión nueva en la que podamos encontrar un hiperplano para separar las clases.** En la siguiente figura vemos cómo al añadir una dimensión nueva, podemos separar fácilmente las dos clases con una superficie de decisión.



Máquinas de Soporte Vectorial (SVM): Aplicaciones con Clasificación

Las máquinas de vectores de soporte eran muy utilizadas antes de la era del deep learning. Para muchas aplicaciones se prefería el uso de SVM en lugar de redes neuronales. La razón era que las matemáticas de las SVM se entienden muy bien y la propiedad de obtener el margen de separación máximo era muy atractivo. Las redes neuronales estándares realizaban clasificaciones de forma equivocada.

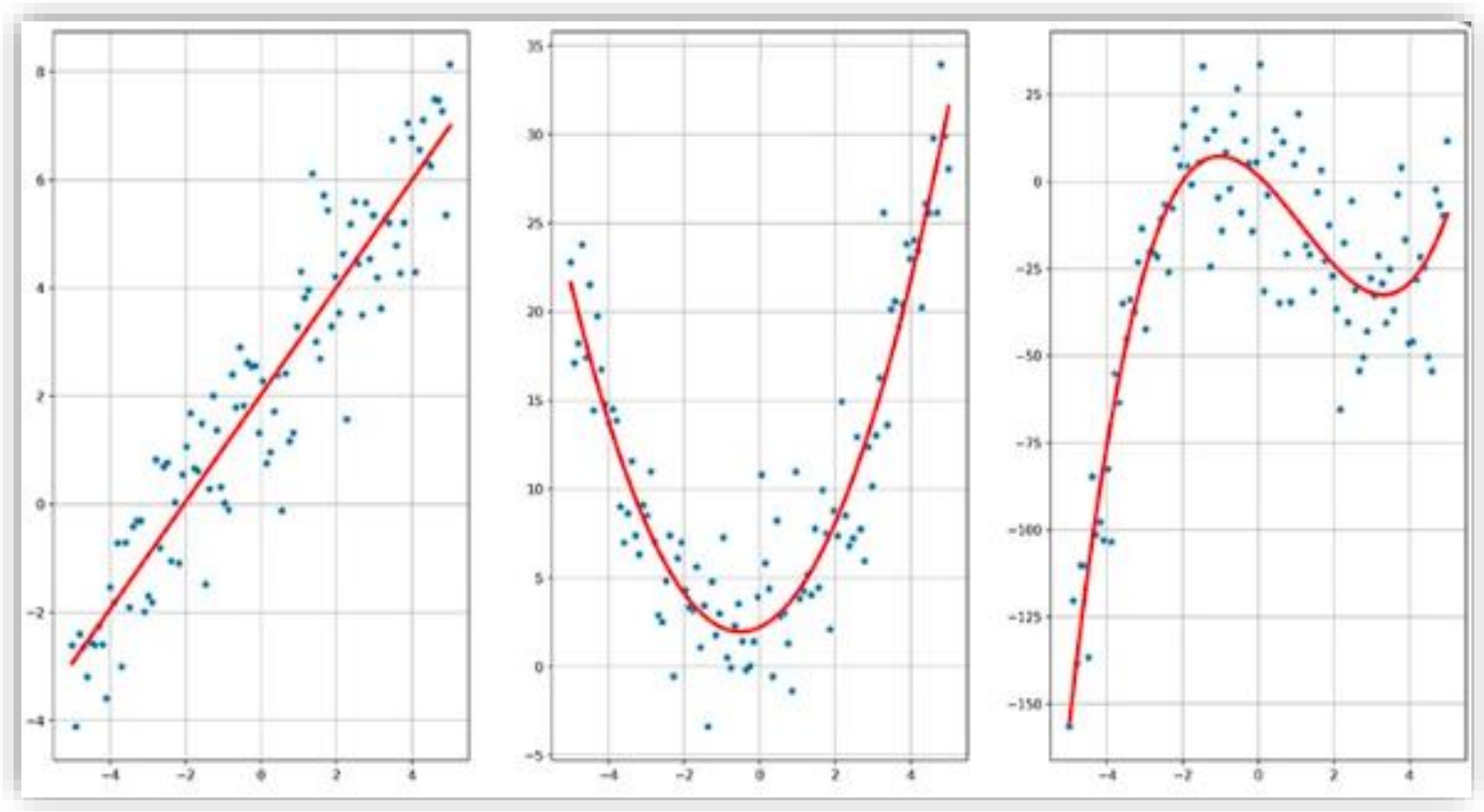
Algunos casos de éxito de las SVM son:

- reconocimiento óptico de caracteres
- detección de rostros para que las cámaras digitales enfoquen correctamente
- filtros de spam para correo electrónico
- reconocimiento de imágenes satelitales (saber qué partes de una imagen tienen nubes, tierra, agua, hielo, etc.)

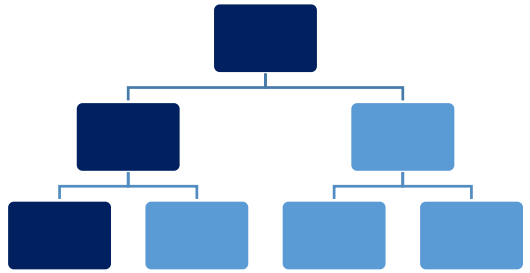
Actualmente, las redes neuronales profundas tienen una mayor capacidad de aprendizaje y generalización que las SVM.

Máquinas de Soporte Vectorial (SVM): Regresión

Se puede tomar el concepto de SVM para construir un regresor. Se pueden definir distintos kernels para estimar las salidas.



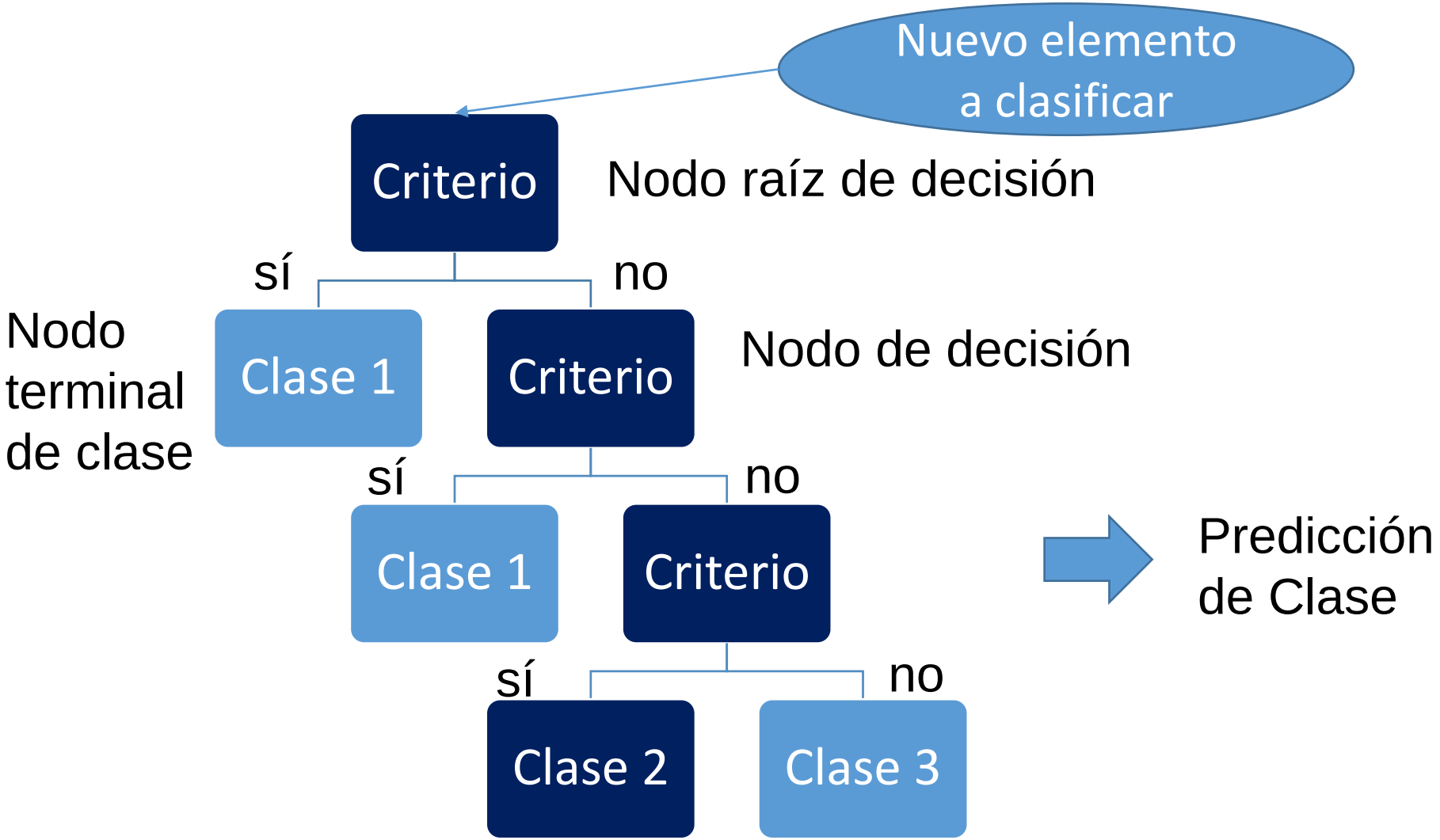
Árboles de decisión y bosques aleatorios



1 Fundamentos de árboles de decisión
2 Fundamentos de bosques aleatorios: cantidad de árboles, profundidad, tamaño de submuestras (bootstrap)
3 Funciones de agregación
4 Bosques aleatorios aplicados a problemas de regresión y clasificación

Algoritmos para clasificación: Decision Tree

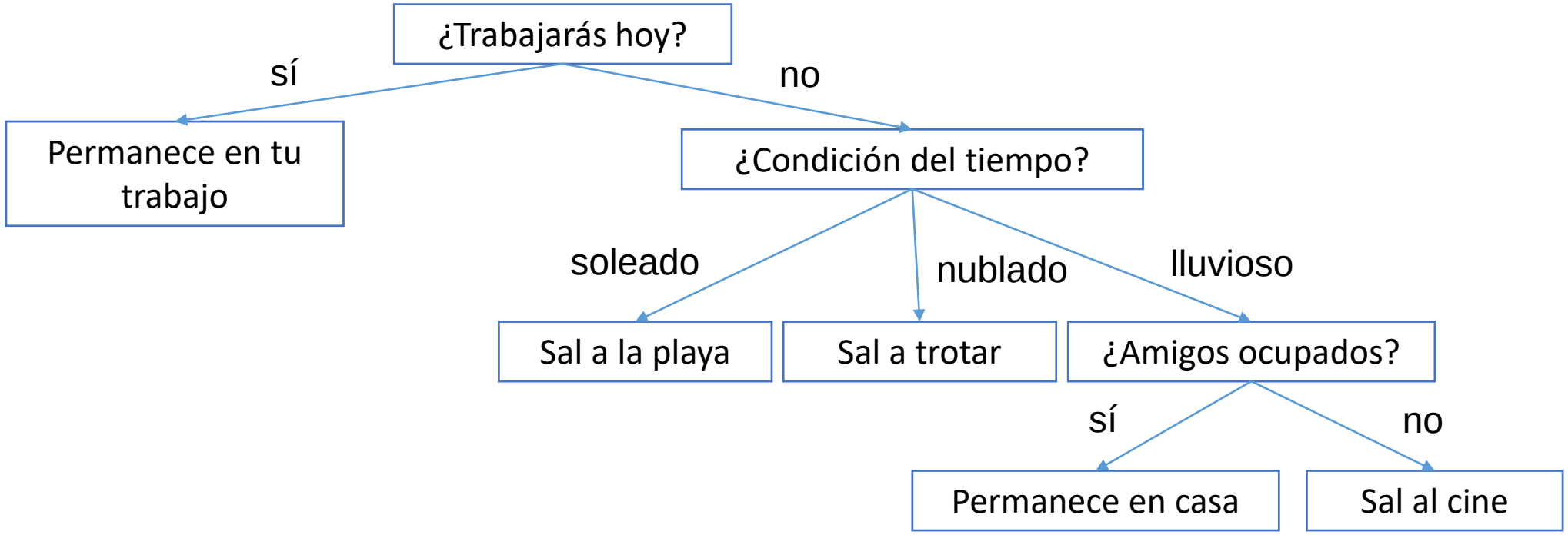
Decision Tree (o Árboles de decisión): los árboles de decisión representan un problema de clasificación como un conjunto de decisiones basadas en los valores de las funciones. Cada nodo del árbol representa un umbral sobre el valor de una función, y parte los ejemplos de entrenamiento en dos grupos más pequeños. Entrenamiento: el proceso de decisión se repite sobre todas las características, con lo que el árbol crece hasta que se calcula una manera óptima de dividir los ejemplos. **Predicción:** La clasificación de un nuevo ejemplo luego puede obtenerse siguiendo las ramas del árbol según los valores de sus funciones.



Algoritmos para clasificación: Decision Tree

Fundamentos:

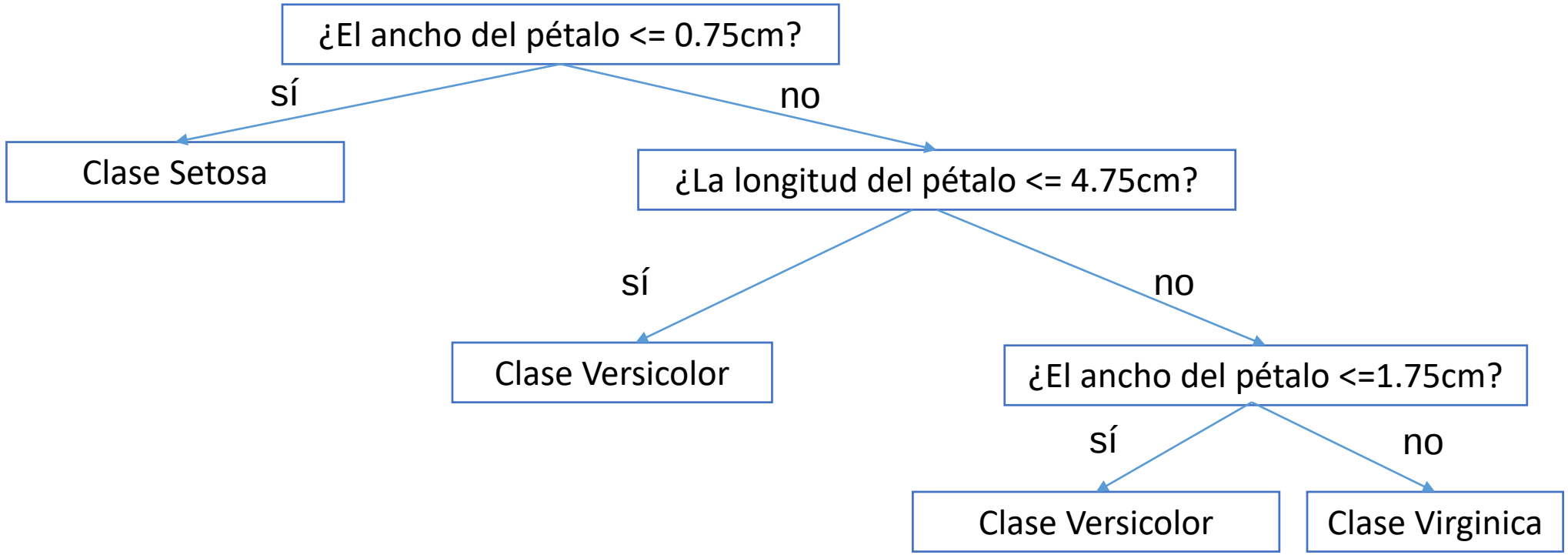
- Un árbol de decisión es un modelo que descompone las observaciones tomando una decisión basada en una serie de preguntas.



Algoritmos para clasificación: Decision Tree

Fundamentos:

- Basado en las variables (features) del conjunto de entrenamiento, el árbol de decisión aprende una serie de condiciones (reglas) para inferir la etiqueta de la clase de las observaciones (samples). **Dataset Iris (3 clases de plantas)**. Un ejemplo con variables con números reales:



Algoritmos para clasificación: Decision Tree

Fundamentos:

- **Pureza de una hoja o nodo:** todos los samples de cada nodo pertenecen a la misma clase.
- En la práctica esto puede generar un árbol con muchos nodos y puede conducir a un overfitting. Por lo tanto, se poda el árbol al configurar un límite para la máxima profundidad del árbol.

Proceso:

- Entrenamiento:
  - Se inicia en la raíz del árbol
  - **Repetir** proceso hasta que las hojas sean puras o hasta que se cumpla la máxima profundidad del árbol:
    - Se divide los datos en las características que den como resultado un mayor **Information Gain (IG)**
    - Descender al nodo hijo
- Predicción de un nuevo sample:
  - Se sigue las ramas del árbol según los valores de sus funciones hasta definir la predicción

Algoritmos para clasificación: Decision Tree

Fundamentos:

- **Maximización de Information Gain (IG):** mejor calidad-precio
- **Función objetivo** para maximizar IG en cada Split:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

- P es el nodo parent (padre) y j es el j-ésimo hijo. I es la medida de impureza. N es el total de samples en el nodo. Por lo tanto, IG es la diferencia entre la impureza del nodo padre y la suma de impurezas de nodos hijo. La menor impureza indica un mayor IG. Para simplicidad en el espacio de búsqueda combinatoria se implementa árboles de decisión binarios (dos hijos). N_p es el número total de ejemplos de entrenamiento en el nodo padre. N_j es el número de ejemplos en el nodo j.

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Algoritmos para clasificación: Decision Tree

Fundamentos:

- Medidas de impureza:
  1. Entropía (I_H): intenta maximizar la información mutua en el árbol

  P (i|t) es la proporción de samples que pertenecen a la clase i para un nodo particular t. La entropía es 0 si todos los samples en el nodo pertenecen a la misma clase y la entropía es máxima (es decir, 1) si la distribución de clase es uniforme.

$$\text{Cross-entropy} = -\sum_{x} p(x) \cdot log\ q(x) \longrightarrow I_H(t) = -\sum_{i=1}^{c} p(i|t)\ log_2\ p(i|t)$$

  2. Gini impurity (I_G): criterio para minimizar la probabilidad de error de clasificación. Similar a la entropía, es máxima si las clases son perfectamente distribuidas.

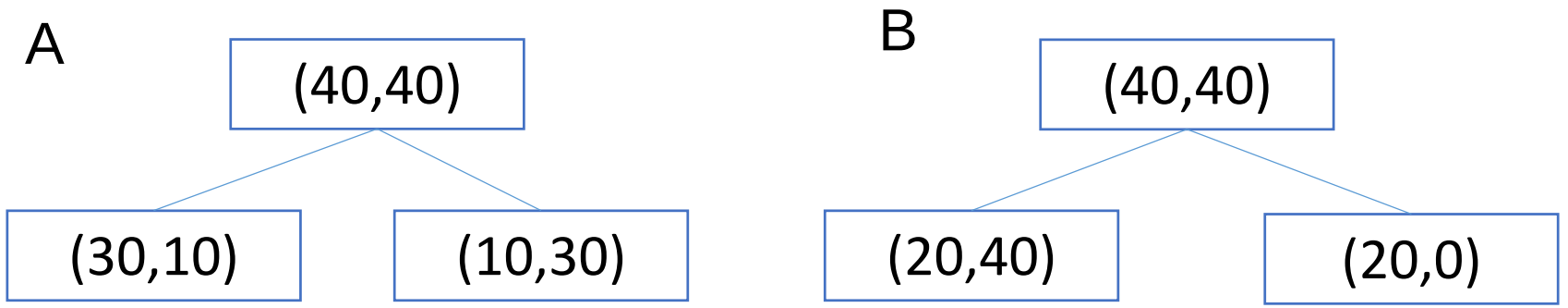
$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2 \longrightarrow I_G(t) = 1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

  3. Classification error (I_E): $I_E(t) = 1 - \max\{ p(i|t)\}$

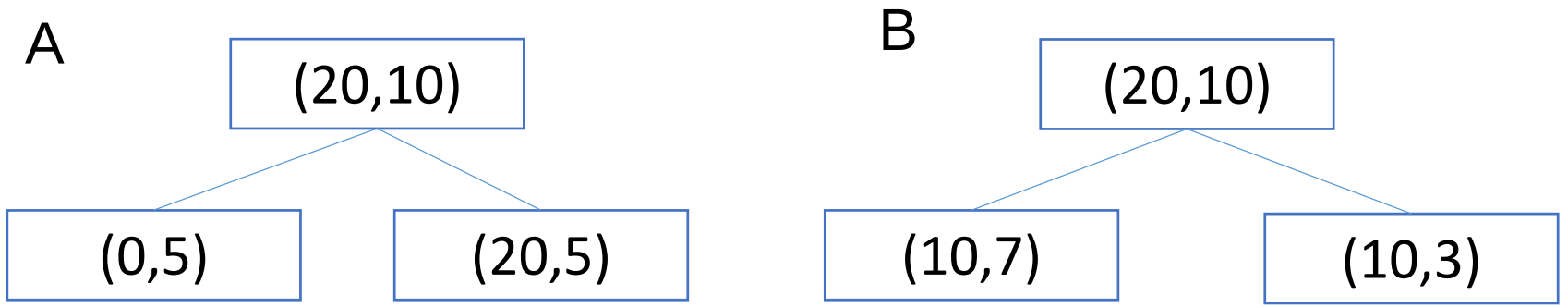
Algoritmos para clasificación: Decision Tree

Fundamentos:

- Ejercicio 1: dos posibles escenarios de splitting. Por classification error, IG_E = 0.25 en ambos escenarios. Por Gini impurity a favor de escenario B (IG_G= 0.16) sobre A (IG_G= 0.125). Por entropía, también es favorable el escenario B (IG_H= 0.31) sobre A (IG_H= 0.19). Revisar página 93, 94 Python Machine Learning (Raschka, 2017).

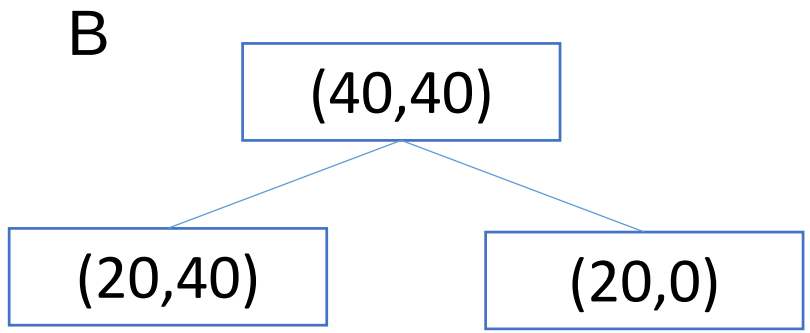


- Ejercicio 2: dos posibles escenarios de splitting. Resolver y presentar resultados.



Algoritmos para clasificación: Decision Tree

- Importante: P (i|t) es la proporción de samples que pertenecen a la clase i para un nodo particular t (el mismo nodo).



$$B: \quad I_E(D_{left}) = 1 - \frac{4}{6} = \frac{1}{3}$$

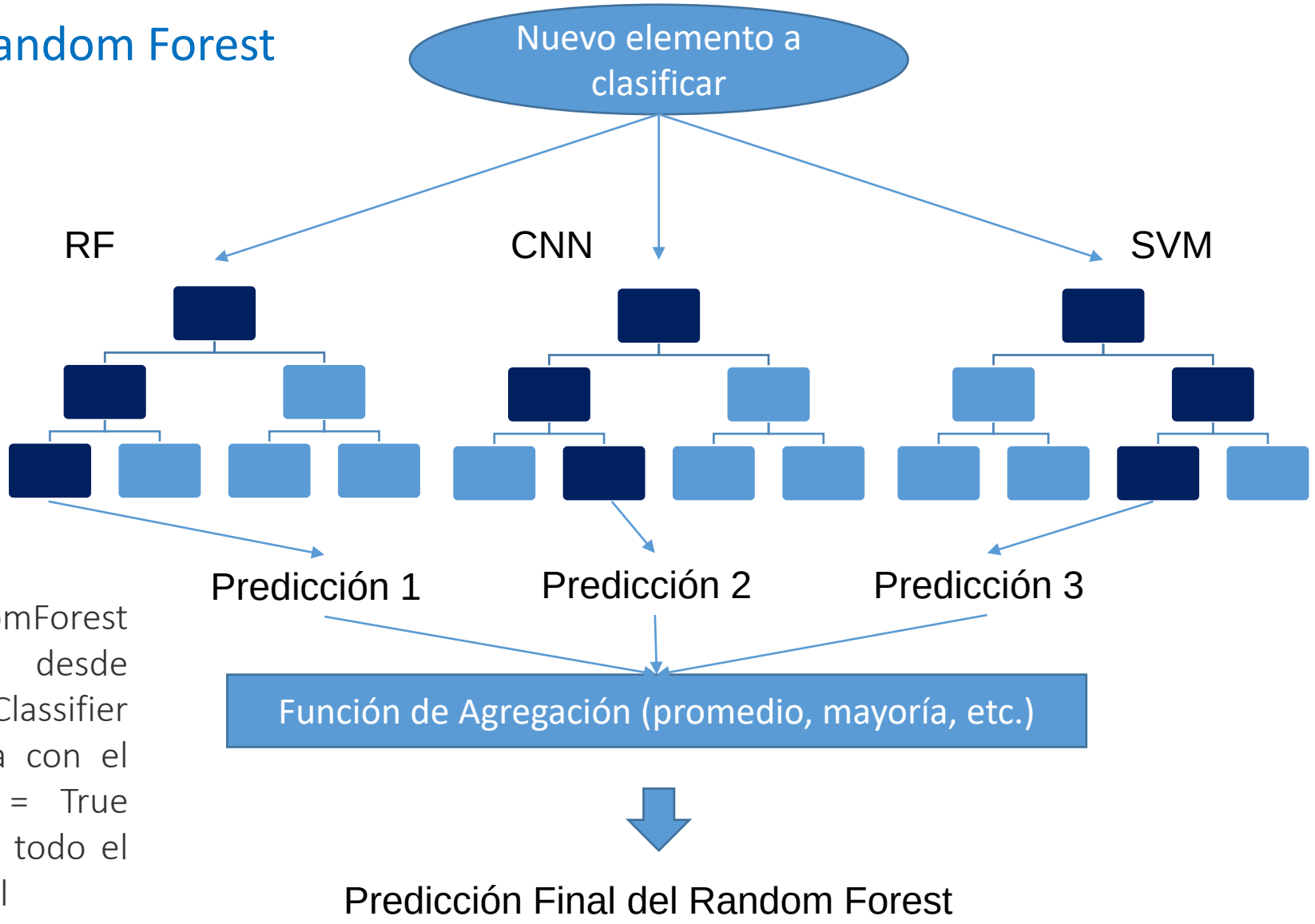
$$B: \quad I_E(D_{Right}) = 1 - 1 = 0$$

I_E considera la información del mismo nodo. Esto es la proporción p(i|t). Por ello, en el ejercicio, en el nodo de la **izquierda** es 4/6 que es el valor máximo entre 20/60 y 40/60. En el nodo hijo de la **derecha** es 1, ya que es el valor máximo entre 20/20 y 0/20, esto es 1 y 0. El máximo es 1. El nodo de la derecha es más puro. Todos los elementos en ese nodo pertenecen a la misma clase.

Algoritmos para clasificación: Random Forest

Un **Random Forest** es un **metaestimador** que se ajusta a una serie de **clasificadores** de árboles de decisión "**Decision Tree**" en varias submuestras del conjunto de datos y utiliza una función de agregación (ejemplo: promedios) para mejorar la precisión predictiva y controlar el sobreajuste.

En Python se puede desarrollar un RandomForest con RandomForestClassifier desde sklearn.ensemble. Con un RandomForestClassifier el tamaño de la submuestra se controla con el parámetro max_samples si bootstrap = True (predeterminado); de lo contrario, se usa todo el conjunto de datos para construir cada árbol



Referencias

[1] C. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.

[2] P. Joshi. Artificial intelligence with python. Packt Publishing Ltd, 2017.

[3] S. Raschka, V. Mirjalili. (2007). Python Machine Learning, Packt Publishing Ltd.

[4] Scholkopf, B., & Smola, A. J. Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press, 2001.

[5] Johnson, A. A., Ott, M. Q., & Dogucu, M. (2022). Bayes Rules!: An Introduction to Applied Bayesian Modeling. CRC Press.

[6] Pourret, O., Na, P., & Marcot, B. (Eds.). (2008). Bayesian networks: a practical guide to applications. John Wiley & Sons.

Enlaces material práctico:

https://github.com/PacktPublishing/Artificial-Intelligence-with-Python

https://github.com/PacktPublishing/Python-Machine-Learning-Second-Edition

https://github.com/PacktPublishing/Python-Parallel-Programming-Cookbook-Second-Edition

Material adicional:

https://machinelearningmastery.com/master-machine-learning-algorithms/