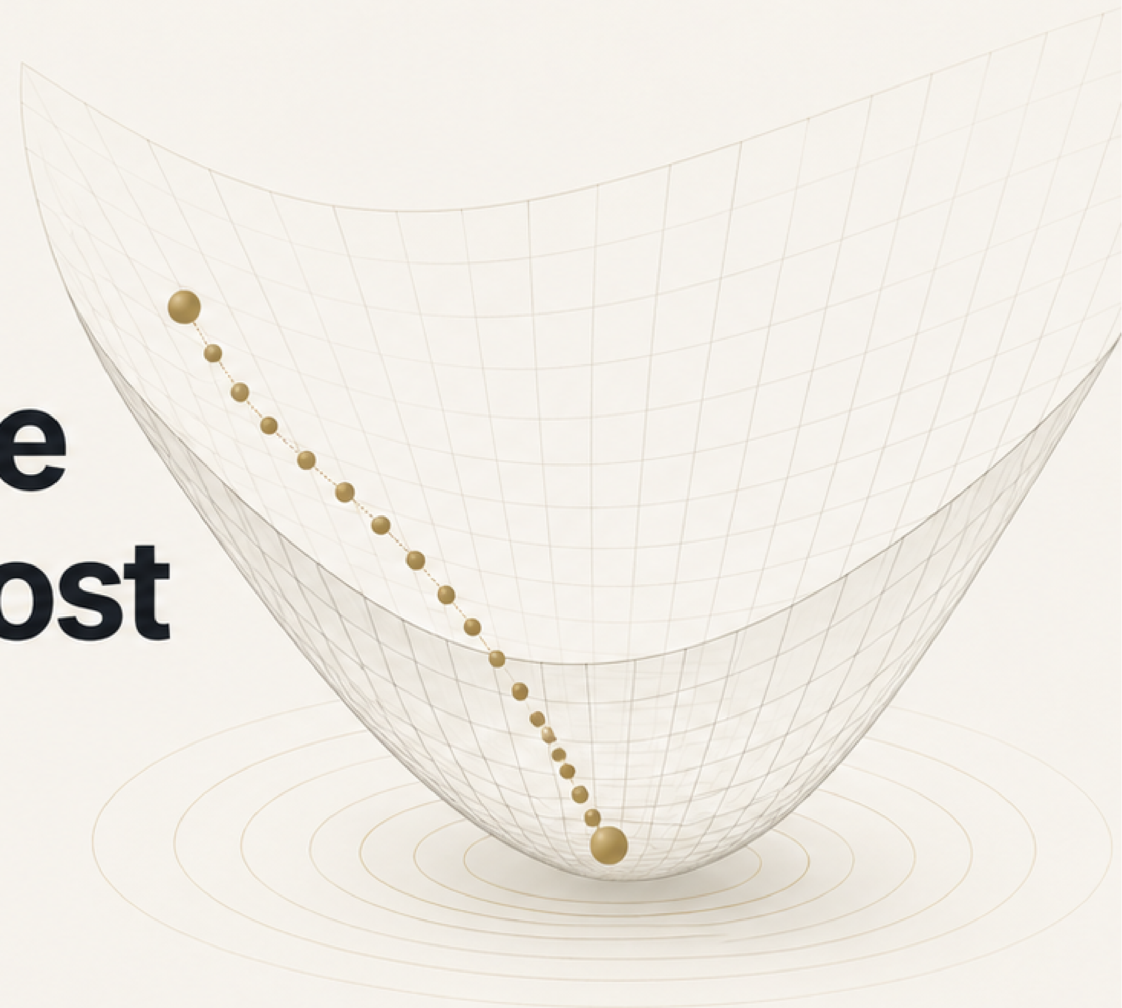
UNIVERSIDAD
Panamericana

Clase 2: De Gradiente Descendente a XGBoost

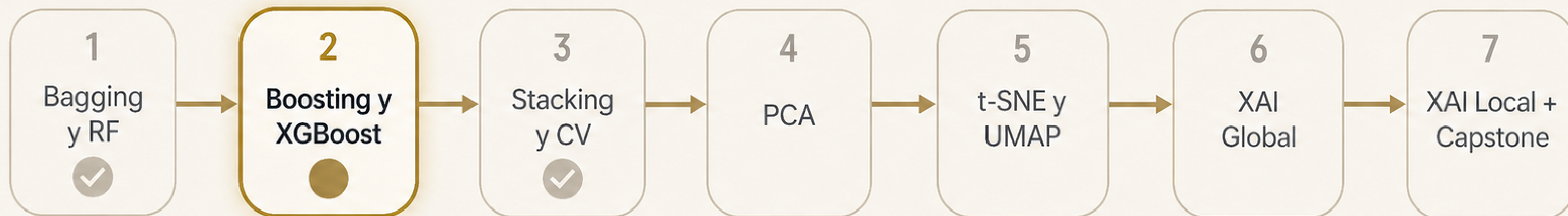
Boosting como descenso por gradiente en el espacio de funciones



Dr. León Palafox — ML2 2026

UNIVERSIDAD
Panamericana

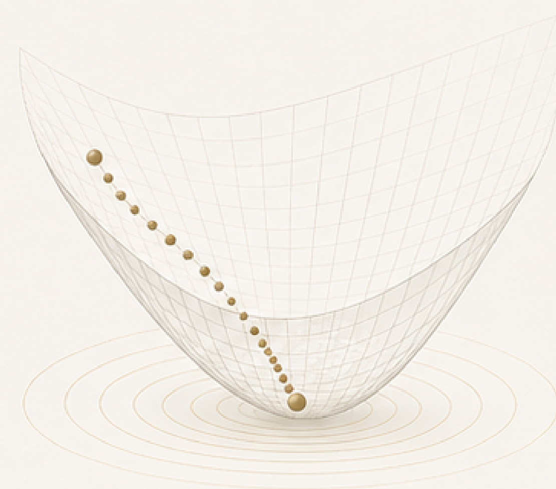
¿Dónde estamos en el curso?



La clase pasada bajamos varianza promediando muchos modelos paralelos (bagging). Hoy vamos a construir modelos secuencialmente, donde cada uno corrige los errores del anterior. Esto cambia qué término del tradeoff atacamos.

Bagging reduce varianza. Boosting reduce sesgo (y, con cuidado, también varianza).

UNIVERSIDAD
Panamericana



Plan de la sesión

Repaso de gradiente descendente

El cuestionario diagnóstico mostró que esto es el hueco más grande del grupo. Vamos a recuperarlo en serio antes de seguir.

2 — 2. 20 min

AdaBoost

El primer algoritmo de boosting (Schapire 1990, Freund & Schapire 1996). Reponderar los puntos mal clasificados.

3 — 3. 40 min

Gradient Boosting

La generalización: en lugar de reponderar puntos, ajustar el siguiente modelo a los residuales del anterior.

4 — 4. 15 min

XGBoost

La implementación que dominó Kaggle 2014–2018 y que sigue siendo el default para datos tabulares en producción.

Y al final del día: Colab donde implementan gradient boosting desde cero (~30 líneas) y comparan contra XGBoost en el mismo dataset de crédito de la Clase 1.

Universidad Panamericana

Por qué empezamos con un repaso de GD

El cuestionario diagnóstico de la semana pasada reveló un patrón claro: gradiente descendente es el concepto donde más respuestas se quedaron en blanco o vagas.

RESPUESTAS REPRESENTATIVAS DEL GRUPO

“No lo recuerdo.”

“Bajar el error del modelo (no sé cómo funciona).”

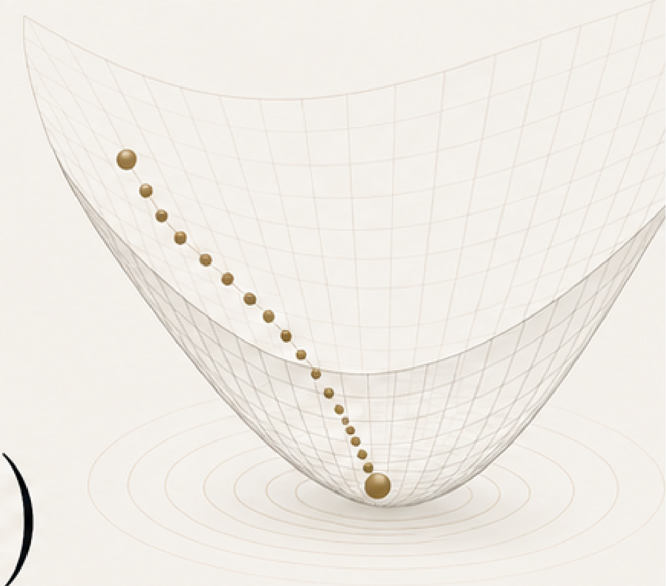
“Es como un tutor.”

No es un problema de inteligencia, es un problema de no haberlo usado recientemente. Pero en boosting, GD no es opcional — el algoritmo **ES** gradiente descendente, sólo que en un espacio menos familiar. Si no entendemos uno, no podemos entender el otro.

Vamos a redibujarlo desde cero. 25 minutos. Después de esto, el resto de la clase tiene sentido.

UNIVERSIDAD Panamericana

¿Qué problema resuelve gradiente descendente?



Casi todos los modelos de ML se entrenan resolviendo un problema de optimización:

$$\hat{\theta} = \arg\min_{\theta} L(\theta; \text{datos})$$

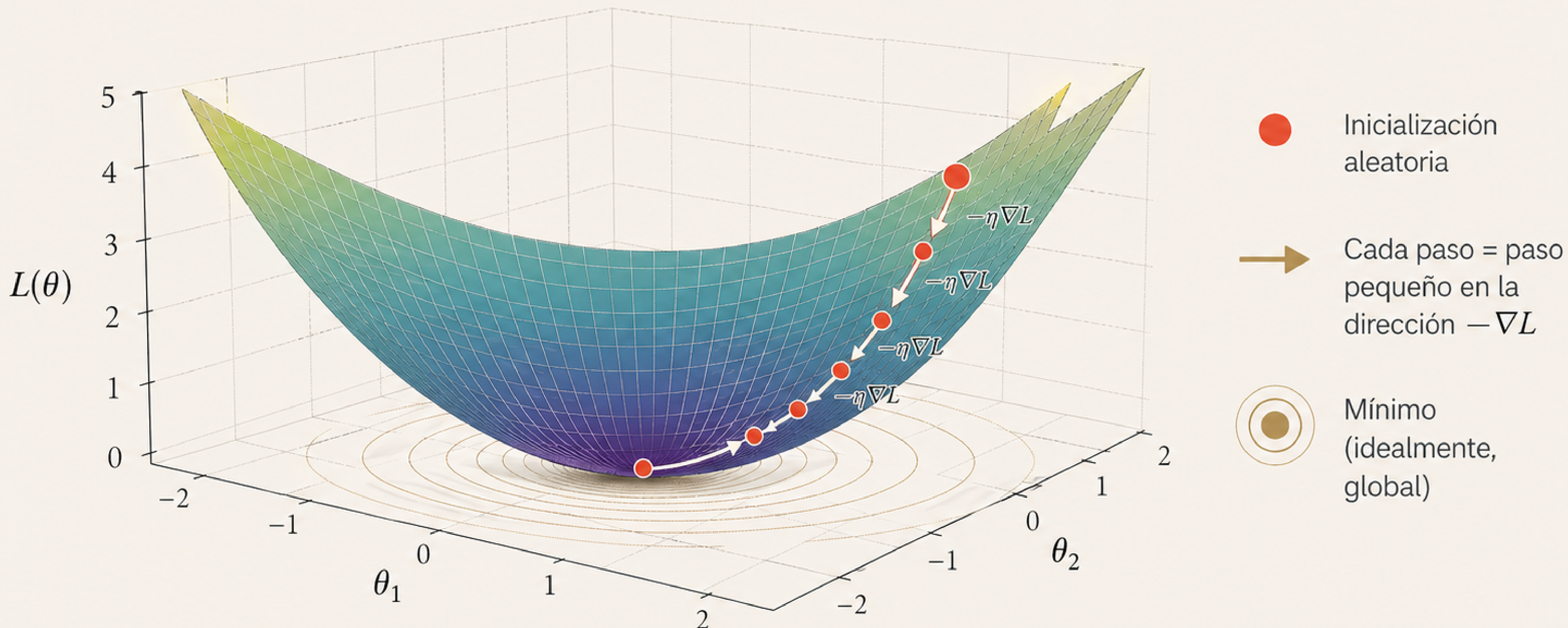
- θ — parámetros del modelo (pesos, coeficientes, lo que sea ajustable)
- $L(\theta; \text{datos})$ — función de pérdida: qué tan mal predice el modelo en los datos, dada la elección de θ
- $\arg\min$ — buscamos los θ que minimizan L

Modelo	θ	L
Regresión lineal	β	MSE
Regresión logística	β	Log-loss
Red neuronal	pesos	cross-entropy

El problema: para regresión lineal hay solución cerrada ($\beta = (X^\top X)^{-1} X^\top y$). Para casi todo lo demás, no. Necesitamos un método numérico iterativo.

UNIVERSIDAD Panamericana

GD: la idea geométrica



En cada punto, el gradiente ∇L apunta en la dirección donde L crece más rápido.
Si nos movemos en la dirección opuesta, bajamos hacia el mínimo.

Es una metáfora útil pero limitada — en la práctica, la superficie tiene millones de dimensiones, no dos. La intuición sigue siendo válida.

La regla de actualización

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

θ_t: parámetros en la iteración t

El signo negativo: nos movemos en contra del gradiente (queremos bajar, no subir)

η (eta): tasa de aprendizaje — qué tan grande es el paso

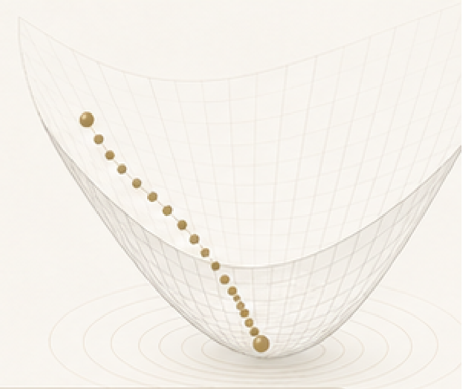
$\nabla L(\theta_t)$: gradiente de la pérdida evaluado en θ_t (la dirección de máximo crecimiento)

```
Inicializar θ aleatoriamente
Repetir hasta convergencia:
    calcular g = ∇L(θ)
    actualizar θ ← θ − η · g
Regresar θ
```

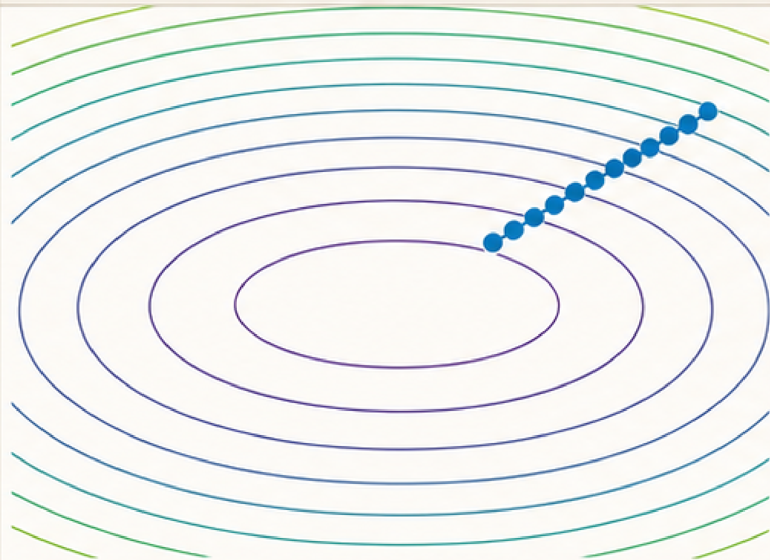
Esto es todo lo que hace GD. La complejidad no está en el algoritmo — está en (1) calcular ∇L eficientemente y (2) elegir η bien.

Universidad Panamericana

La tasa de aprendizaje: el hiperparámetro que rompe todo si lo eliges mal

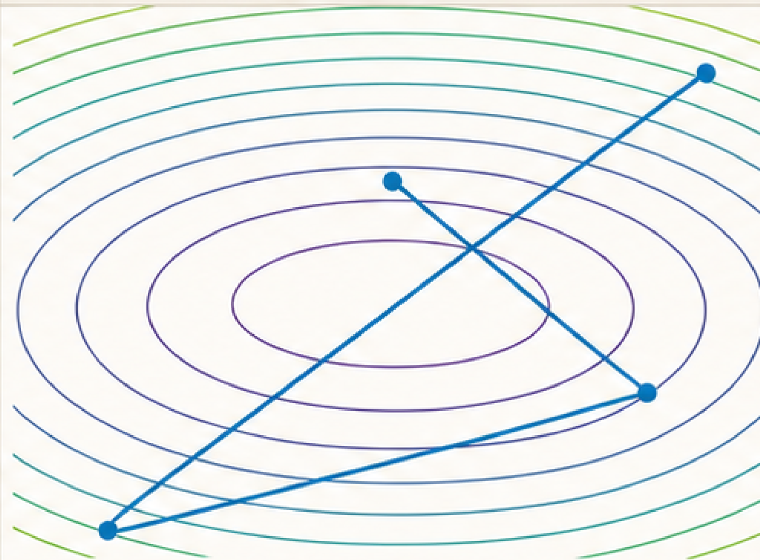


η muy pequeño



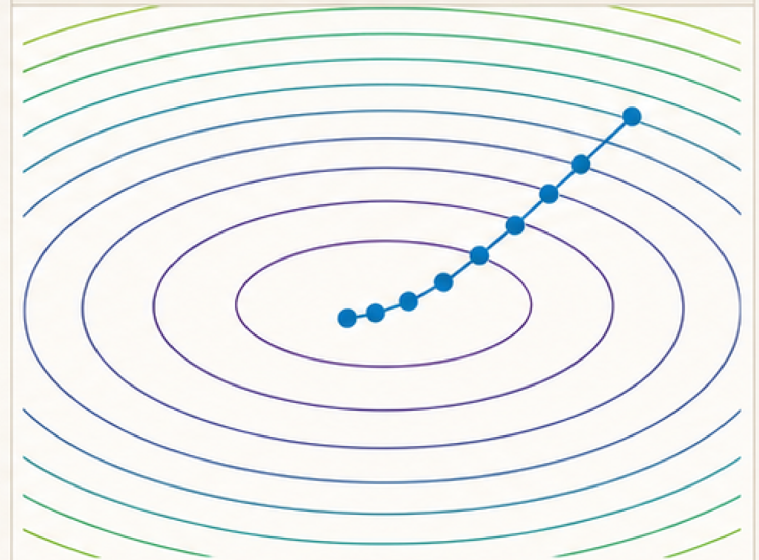
Convergencia lenta. Tu modelo nunca termina de entrenar.

η muy grande



El paso es tan grande que rebotas. La pérdida puede crecer en lugar de bajar.

η apropiado

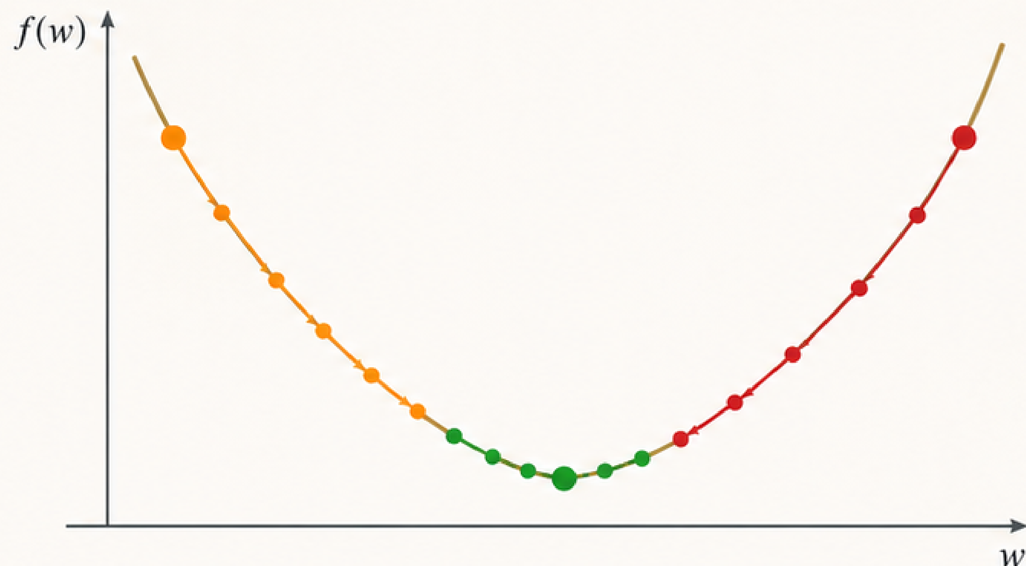


El balance correcto. En la práctica, requiere experimentación o schedules adaptativos.

Heurísticas comunes: empezar en 0.01 o 0.001 para redes neuronales, 0.1 para boosting (LR pequeño + muchos árboles). **Optuna** para tuning serio.

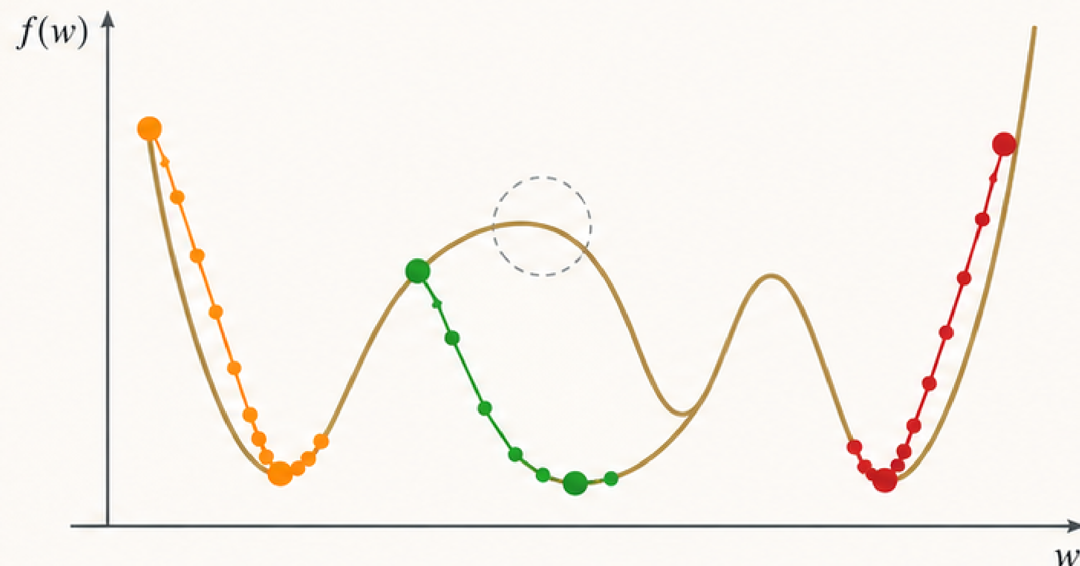
Una advertencia: GD no garantiza el mínimo global

Convexa (un solo mínimo)



Regresión lineal, regresión logística (con pérdida log-loss). GD garantiza el óptimo global.

No-convexa (varios mínimos locales)



Redes neuronales, boosting con árboles. GD encuentra un mínimo local — no necesariamente el mejor. La inicialización importa.

En boosting con árboles de decisión, la pérdida es no-convexa en el espacio de funciones.
Pero como veremos, esto no es tan malo en la práctica — el algoritmo es robusto.

Panamericana

El puente conceptual: GD en parámetros vs GD en funciones

GD clásico (lo que acabamos de ver)

- Optimizamos sobre parámetros θ
- En cada paso, actualizamos:

$$\theta_{t+1} = \theta_t - \eta \nabla L$$

- Ejemplo: ajustar los β de una regresión logística
- El modelo tiene una estructura fija; sólo cambian los números

Boosting (lo que vamos a hacer hoy)

- Optimizamos sobre funciones F
- En cada paso, actualizamos:

$$F_{t+1} = F_t + \eta \cdot h_t$$

- Donde h_t es un nuevo árbol que apunta en la dirección $-\nabla L$
- El modelo mismo crece en cada iteración

Boosting es gradiente descendente. La diferencia es que el paso no es un cambio en los parámetros — es un nuevo modelo añadido al ensemble. Cada árbol nuevo es un paso en la dirección que reduce más la pérdida.

Esto va a tomar dos secciones para construirse. Primero AdaBoost (donde la conexión es implícita). Después gradient boosting (donde es explícita).

UNIVERSIDAD
Panamericana

¿Se pueden combinar muchos modelos débiles en uno fuerte?

> *"Can a set of weak learners be combined to create a stronger learner?"*

Kearns & Valiant (1988) — pregunta original en teoría del aprendizaje.

Un *weak learner* es un modelo que predice apenas mejor que el azar.
Para clasificación binaria balanceada, eso significa accuracy > 50%.
Una pregunta más natural: si tengo muchos modelos así, ¿puedo combinarlos en algo bueno?

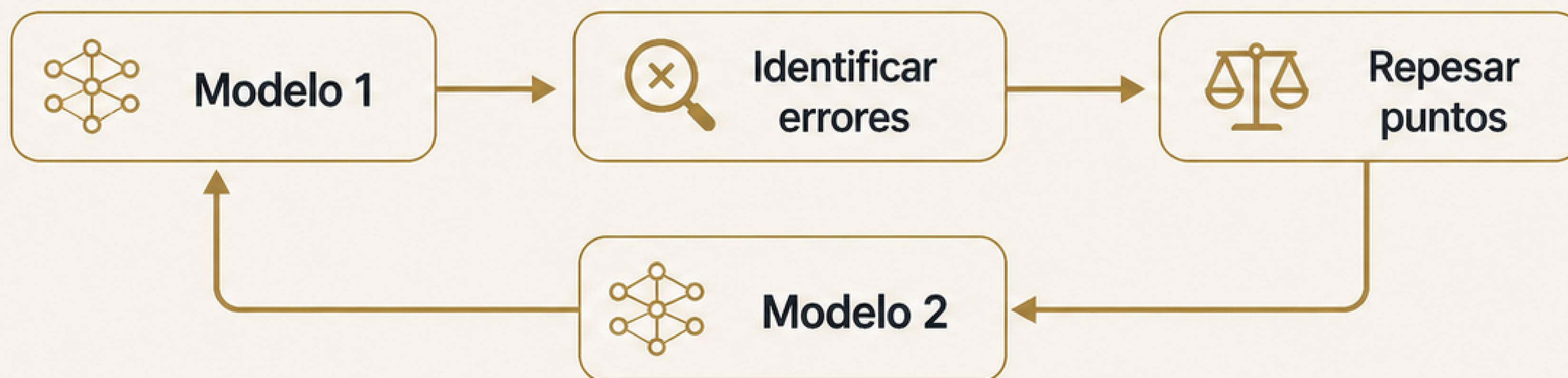
Sí. Schapire (1990) demostró que es posible.
El algoritmo concreto vino después: **AdaBoost** (Freund & Schapire, 1996).

AdaBoost fue tan exitoso que ganó el **Premio Gödel** en 2003.
Es uno de los algoritmos más influyentes en la historia del ML.

Universidad Panamericana

AdaBoost en una idea

Entrena un modelo. Identifica los puntos que clasificó mal.
Entrena el siguiente modelo dándole más peso a esos puntos. Repite.



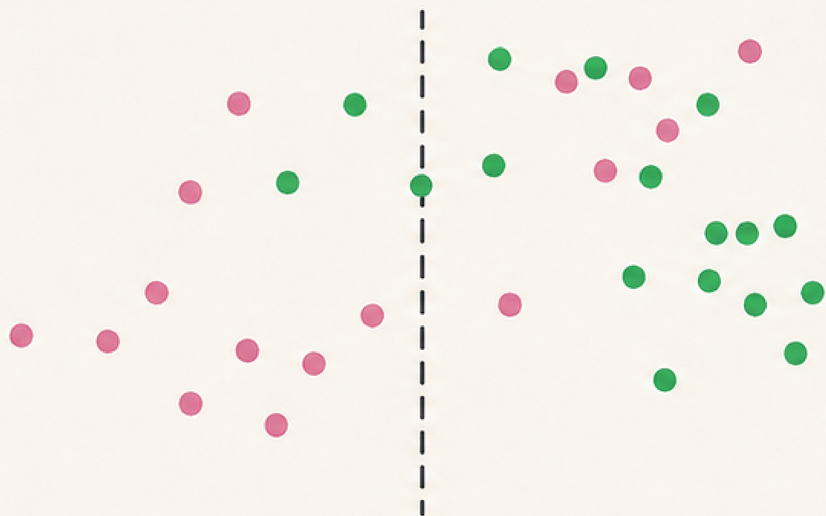
Predicción final = voto ponderado de todos los modelos

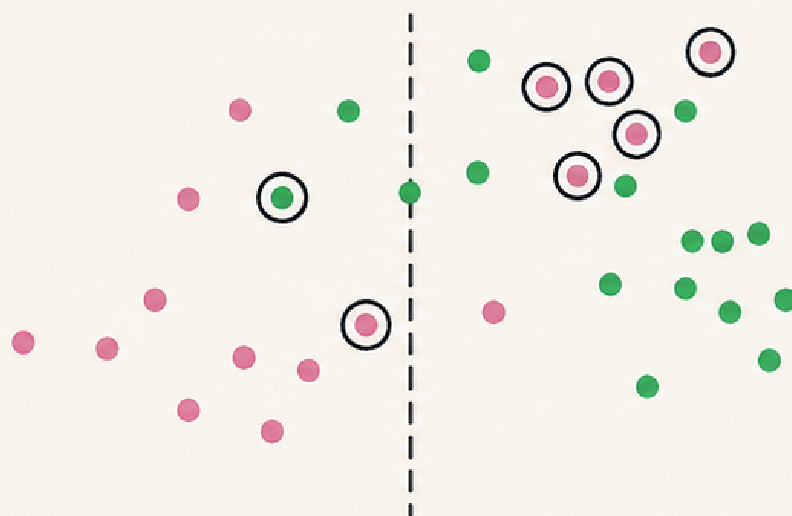
Cada modelo se enfoca en lo que el modelo anterior no logró.
Por eso boosting reduce sesgo: cada iteración está atacando errores que aún quedan.

Compara con **bagging:** ahí cada modelo entrena en una muestra independiente.
En boosting, cada modelo depende del anterior. Es secuencial, no paralelo.

Universidad Panamericana

Ronda 1: el primer decision stump





Punto de partida: todos los puntos pesan igual ($w_i = 1/n$). Entrenamos un decision stump — el modelo más simple posible: un árbol con un solo split. Identificamos los puntos mal clasificados (los círculos). En la ronda 2, esos puntos van a pesar más.

Error de la ronda 1:

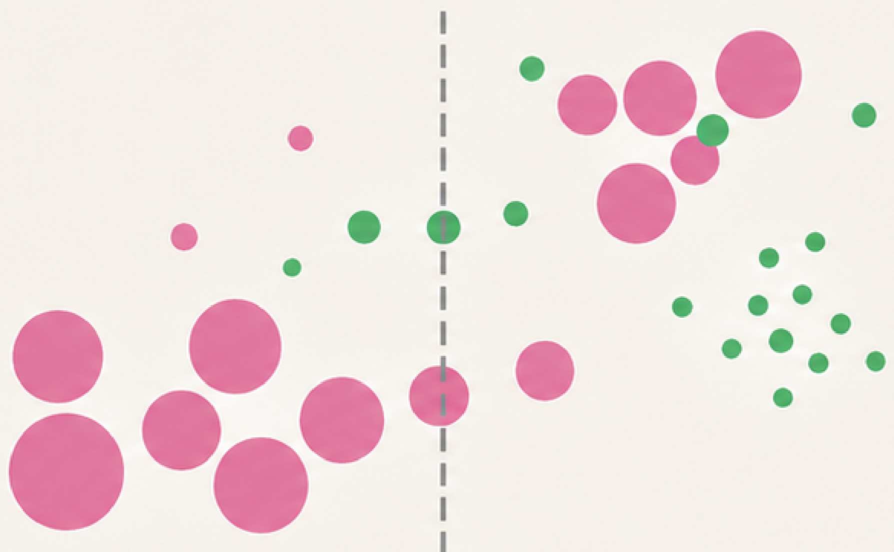
$\varepsilon_1 =$ (suma de pesos de los mal clasificados)

Peso del modelo 1 en el ensemble final:

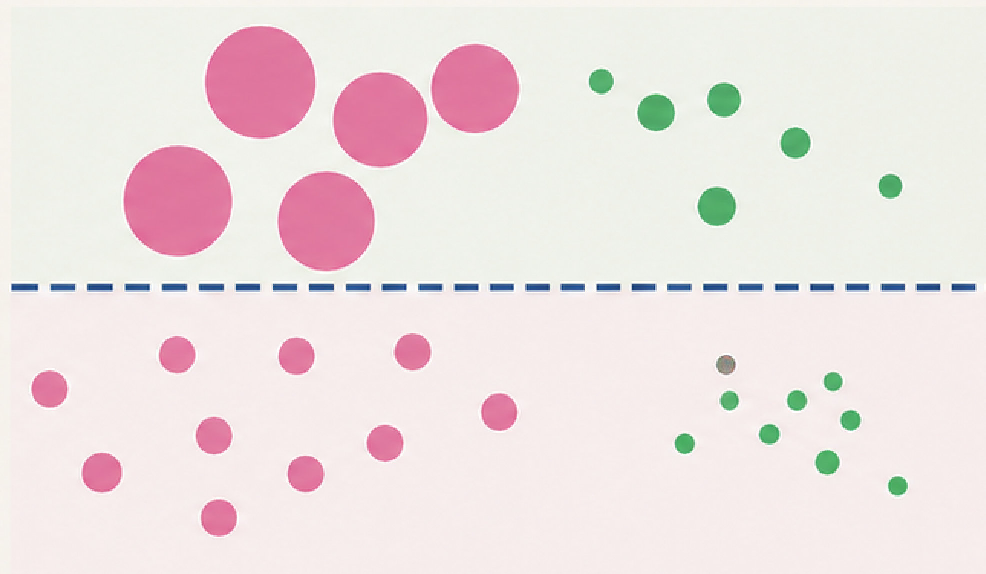
$$\alpha_1 = \frac{1}{2} \cdot \log\left(\frac{1-\varepsilon_1}{\varepsilon_1}\right)$$

Ronda 2: re-pesar y volver a entrenar

Datos re-pesados tras el modelo 1



Nuevo stump entrenado en los datos re-pesados

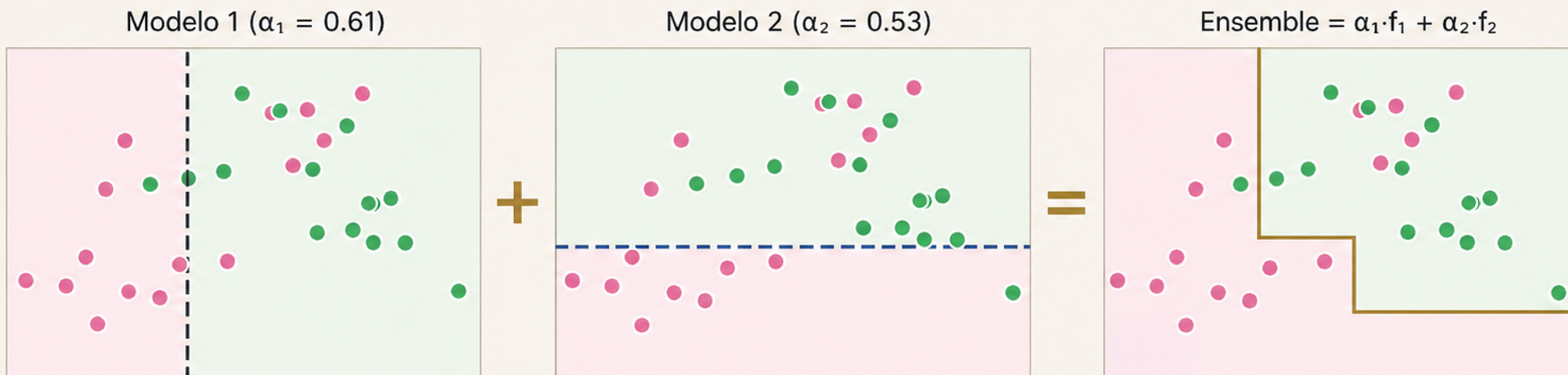


Los puntos que el modelo 1 clasificó mal ahora **pesan más.** Cuando entrenamos el modelo 2 en estos datos repesados, el algoritmo se ve forzado a prestar atención a esos puntos — incluso si eso significa equivocarse en otros que antes estaban bien.

Esto es deliberadamente mal para el modelo 2 individualmente. Su accuracy va a ser menor que la del modelo 1. Pero el modelo 2 es complementario: corrige lo que el 1 no pudo.

Universidad Panamericana

Combinación: el ensemble final



La predicción final es el voto ponderado de los modelos. Modelos con menor error reciben mayor peso α. La frontera de decisión final es más compleja que cualquier modelo individual — y eso a pesar de que cada modelo es trivialmente simple (un solo split).

Esta es la magia de AdaBoost: weak learners + reponderar errores + voto ponderado = strong learner.

Universidad Panamericana

Después de 30 iteraciones

Training error = 0. AdaBoost convergió a un clasificador que separa perfectamente los datos de entrenamiento.

¿Esto es bueno?

Depende. Recordando la Clase 1: *training error* = 0 puede ser sobreajuste. Pero — y este es uno de los resultados teóricos más sorprendentes de AdaBoost — empíricamente, **sigue mejorando el test error** incluso después de que training error llega a cero. La razón está relacionada con el **margen** (concepto de SVM), pero no es trivial.

Schapire et al. (1998), *"Boosting the margin"*. Lectura opcional para los curiosos.

AdaBoost: la matemática en una página

Inicializar: $w_i = \frac{1}{n}$ para $i = 1, \ldots, n$

Para $t = 1, \ldots, T$:

1. Entrenar weak learner f_t en datos con pesos w
2. Calcular error pesado: $\varepsilon_t = \sum w_i \cdot I[y_i \neq f_t(x_i)]$
3. Calcular peso del modelo: $\alpha_t = \frac{1}{2} \cdot \log\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$
4. Actualizar pesos: $w_i \leftarrow w_i \cdot \exp(\alpha_t \cdot I[y_i \neq f_t(x_i)])$
5. Normalizar w para que sumen 1

Predicción final: $F(x) = \text{sign}\left(\sum \alpha_t \cdot f_t(x)\right)$

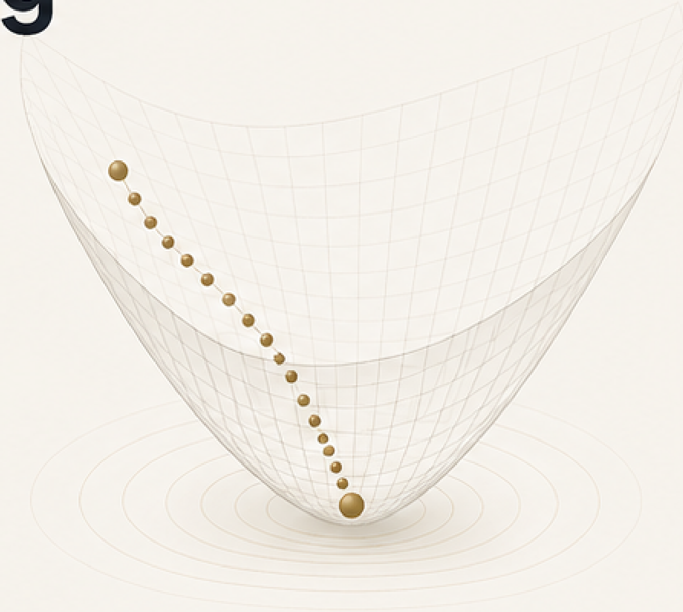
ε_t es el error pesado, no el error simple

α_t crece cuando ε_t es pequeño — modelos buenos pesan más

Puntos mal clasificados ven sus pesos multiplicados por $\exp(\alpha_t) > 1$

Este algoritmo se puede derivar como gradiente descendente sobre la pérdida exponencial $L(y, F(x)) = \exp(-y \cdot F(x))$. Pero la conexión es sutil. Para que la conexión sea explícita, necesitamos *gradient boosting*.

De AdaBoost a Gradient Boosting



AdaBoost funciona, pero tiene limitaciones:

- **Sólo funciona con pérdida exponencial.** Cambiar la pérdida (a log-loss, a MSE, a Huber, etc.) requiere re-derivar el algoritmo desde cero.
- **Sólo funciona con clasificación binaria de manera natural.** Extenderlo a regresión o multiclase es engorroso.
- **La conexión con gradiente descendente es implícita.** No es obvio que estamos minimizando algo de manera principial.

Gradient Boosting (Friedman, 1999) generaliza AdaBoost: cualquier pérdida diferenciable, cualquier tipo de problema (clasificación, regresión, ranking), conexión explícita con GD.

En lugar de reponderar puntos, vamos a entrenar el siguiente modelo para predecir el error del modelo anterior.

Friedman, J. (1999). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*.

La intuición: cada modelo predice los errores del anterior

Ejemplo concreto: precio de una casa

Precio real:	Predicción del modelo F:	Error (residual):
$y = \$1,000,000$	$F(x) = \$800,000$	$r = y - F(x) = \$200,000$

Replanteamiento:

¿Y si entrenamos un segundo modelo $h(x)$ que prediga este residual?
Si $h(x) \approx \$200,000$, entonces $F(x) + h(x) \approx \$1,000,000$ — corregimos el error.

Generalización (la idea central):

$$F_{nuevo}(x) = F(x) + h(x), \quad \text{donde } h \approx y - F(x)$$

Cada modelo nuevo es entrenado para predecir lo que el modelo actual se equivoca.
No estamos reponderando puntos — estamos prediciendo residuales.

Universidad Panamericana

Formalización: lo que queremos

Tenemos pares de datos (x_i, y_i) y un modelo actual $F(x)$. Hay errores:

$$F(x_1) = 0.8, \quad y_1 = 0.9$$

El modelo F no podemos modificarlo (es un árbol ya entrenado, no podemos "mover" sus hojas). Pero podemos añadirle un modelo nuevo $h(x)$:

$$\begin{aligned} F(x_1) + h(x_1) &= y_1 \\ F(x_2) + h(x_2) &= y_2 \\ &\vdots \\ F(x_n) + h(x_n) &= y_n \end{aligned}$$

Despejando h:

$$h(x_i) = y_i - F(x_i)$$

Queremos entrenar $h(x)$ para que prediga $y_i - F(x_i)$ — exactamente los **residuales**. Esto convierte el problema de boosting en un problema de regresión estándar.

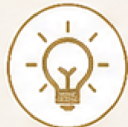
Pseudocódigo: Gradient Boosting (versión simple)

```
def gradient_boost(X, y, n_rounds, learning_rate):
    # Inicialización: predicción constante (la media)
    F = mean(y)
    models = [F]

    for t in range(n_rounds):
        # 1. Calcular residuales del modelo actual
        residuals = y - F(X)

        # 2. Entrenar un weak learner para predecir los residuales
        h_t = DecisionTree(max_depth=3).fit(X, residuals)

        # 3. Actualizar el modelo: F ← F + η · h_t
        F = F + learning_rate * h_t
        models.append(h_t)

    return F, models
```

Aquí vive la conexión con GD — esto es un gradiente

Regresión estándar; el target son los residuales, no las y originales

η pequeño + muchas iteraciones > η grande + pocas



Esto es esencialmente lo que van a implementar en el Colab. ~30 líneas reales en Python.

Universidad Panamericana

Por qué se llama ‘gradient’ boosting

GD clásico

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathcal{L}$$

Movemos los parámetros en la dirección opuesta al gradiente de la pérdida respecto a los parámetros.

Gradient Boosting

$$F_{t+1}(x) = F_t(x) + \eta \cdot h_t(x), \quad h_t \approx -\nabla_F \mathcal{L}$$

Movemos la función F en la dirección opuesta al gradiente de la pérdida respecto a la función. h_t es un árbol que aproxima ese gradiente.

Para pérdida cuadrática $L = \frac{1}{2}\,(y - F(x))^2$, el gradiente respecto a F es:

$$\nabla_F \mathcal{L} = -(y - F(x)) = -\ residual$$

Por eso entrenar h para predecir el residual es entrenar h para aproximar $-\nabla \mathcal{L}$.

Gradient Boosting es gradient descent en el espacio de funciones.

Para otras pérdidas (log-loss, Huber, etc.) el gradiente cambia, pero la lógica es la misma: ajusta h para que aproxime el negativo del gradiente actual.

Universidad Panamericana

Visualización: gradient boosting iteración por iteración

t=0 | t=1 | t=3 | t=10 | t=50 | t=200

Datos
Función verdadera
F_t (modelo acumulado)

Cada panel suma un árbol más al ensemble. Al principio, los árboles capturan estructura grande (sesgo bajando rápido). Eventualmente, empiezan a capturar ruido (varianza subiendo). Detener al momento correcto = ***early stopping***.

En la práctica: usar ***early stopping*** con un set de validación. XGBoost lo hace automáticamente.

Hiperparámetros que importan en Gradient Boosting

1. **learning_rate (η, también llamado shrinkage)**
   Multiplicador del paso. Valores típicos: 0.01–0.3.
   Más pequeño → más árboles necesarios pero mejor generalización. Trade directo con *n_estimators*.

2. **n_estimators (número de árboles)**
   Cuántas iteraciones de boosting. A diferencia de RF, más NO siempre es mejor — eventualmente se sobreajusta. Usar early stopping.

3. **max_depth (profundidad de cada árbol)**
   Aquí los árboles no son profundos como en RF. Típico: 3–8.
   Árboles más profundos en boosting → sobreajuste casi garantizado.

4. **subsample (stochastic gradient boosting)**
   Fracción de filas que cada árbol ve (similar al bootstrap de bagging, pero sin reemplazo).
   Valores típicos: 0.5–1.0. Reduce overfitting y acelera entrenamiento.

5. **colsample_bytree (similar a max_features de RF)**
   Fracción de columnas consideradas en cada árbol.
   Otra fuente de regularización.

RF era tolerante a hiperparámetros mal elegidos.
XGBoost no. Tunear es obligatorio. Defaults son un punto de partida, no un destino.

UNIVERSIDAD Panamericana

Cualquier pérdida diferenciable

Una de las ventajas de Gradient Boosting sobre AdaBoost: funciona con cualquier pérdida diferenciable. Sólo necesitas saber calcular su gradiente.

Problema	Pérdida $L(y, F)$	Gradiente $-\frac{\partial L}{\partial F}$
Regresión (MSE)	$\frac{1}{2}(y - F)^2$	$y - F$ (residual estándar)
Regresión robusta (Huber)	piecewise	piecewise
Clasificación binaria (log-loss)	$\log(1 + \exp(-yF))$	$y - \sigma(F)$
Clasificación multiclase	softmax cross-entropy	$y_{\text{one_hot}} - \text{softmax}(F)$
Ranking (LambdaRank)	pairwise	gradientes pairwise

Esto es por qué XGBoost es el caballo de batalla en industria.
Mismo algoritmo, distinto problema, sólo cambias la pérdida.
En finanzas: regresión para predicción de retornos, log-loss para default, Huber para datos con outliers.

Universidad Panamericana

XGBoost: una implementación, varias mejoras

Chen & Guestrin, 2016 — el paper más citado de KDD

Gradient Boosting es el algoritmo. XGBoost es una implementación.

Otras implementaciones: scikit-learn's *GradientBoostingClassifier*, LightGBM (Microsoft), CatBoost (Yandex).

Lo que hace especial a XGBoost:

1

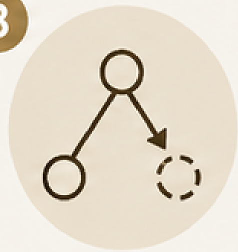
Aproximación de Taylor de segundo orden.

Usa no sólo el gradiente sino también el hessiano (segunda derivada). Pasos más informados, convergencia más rápida.

2

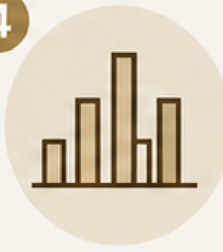
Regularización en el objetivo.

Penaliza directamente la complejidad del árbol (número de hojas, magnitud de los pesos en cada hoja). No sólo `max_depth` — regularización formal en la función objetivo.

3 

Sparsity-aware split finding.

Maneja valores faltantes nativamente. Aprende la dirección óptima para enviar los NaN en cada split. No requiere imputación previa.

4 

Histogram-based splitting + paralelización.

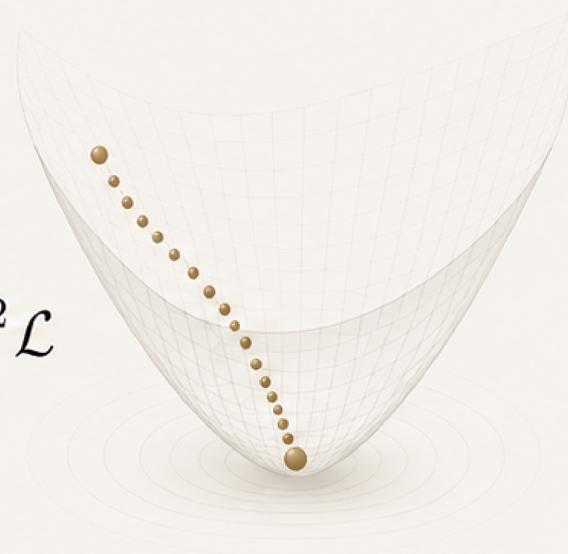
Discretiza features continuos en buckets, hace splits sobre buckets en lugar de sobre todos los valores únicos. Mucho más rápido.

Chen, T., & Guestrin, C. (2016). *XGBoost: A scalable tree boosting system. KDD '16.*

Por qué XGBoost converge más rápido: segundas derivadas

Gradient Boosting clásico aproxima la pérdida con una expansión de Taylor de primer orden:

$$\mathcal{L}(F+h) \approx \mathcal{L}(F) + g \cdot h, \qquad g = \nabla \mathcal{L}$$



XGBoost usa segundo orden:

$$\mathcal{L}(F+h) \approx \mathcal{L}(F) + g \cdot h + \frac{1}{2} \cdot h \cdot H \cdot h, \qquad H = \nabla^2 \mathcal{L}$$

$$g_i = \frac{\partial \mathcal{L}}{\partial F(x_i)}, \qquad h_i = \frac{\partial^2 \mathcal{L}}{\partial F(x_i)^2}$$

Y el peso óptimo para una hoja con instancias I:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

El gradiente te dice **dirección**. El hessiano te dice **curvatura**. Con ambos, sabes no sólo hacia dónde ir sino qué tan grande puede ser el paso. Esa es la conexión con el método de Newton — XGBoost es Newton, gradient boosting es gradiente puro.

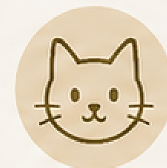
El parámetro λ en el denominador es la regularización L_2. Aparece naturalmente en la derivación, no es un add-on.

UNIVERSIDAD
Panamericana

Las otras implementaciones: LightGBM y CatBoost

LightGBM (Microsoft, 2017)

- **Leaf-wise growth** en lugar de level-wise (XGBoost). Crece la hoja con mayor pérdida primero.
- **Más rápido** que XGBoost para datasets grandes (10M+ filas).
- **Más propenso a sobreajuste** — requiere num_leaves con cuidado.
- **Cuándo usar:** datasets muy grandes, cuando velocidad importa.



CatBoost (Yandex, 2017)

- **Ordered boosting:** maneja variables categóricas nativamente sin one-hot.
- Maneja target encoding sin leakage automáticamente.
- Más lento que LightGBM, más robusto a hiperparámetros mal elegidos.
- **Cuándo usar:** muchas variables categóricas, cuando no quieres tunear demasiado.

Para industria financiera con datos tabulares mixtos: empezar con **XGBoost** (default seguro), probar **LightGBM** si necesitas más velocidad, probar **CatBoost** si tienes muchas categóricas con alta cardinalidad. Los tres son razonables.

UNIVERSIDAD
Panamericana

Boosting vs Bagging: comparación final

Dimensión	Bagging / RF	Boosting / XGBoost
Estructura	Modelos paralelos, independientes	Modelos secuenciales, dependientes
Qué reduce	Varianza	Sesgo (y varianza, con cuidado)
Base learner típico	Árboles profundos (alta varianza)	Árboles poco profundos (alta sesgo)
Tolerancia a hiperparámetros	Alta (defaults funcionan)	Baja (tunear es obligatorio)
Sobreajuste con muchos árboles	No (asintóticamente plateau)	Sí (más es eventualmente peor)
Paralelización	Trivial (entrenar árboles en paralelo)	Sólo dentro de cada árbol
Robustez a ruido en y	Alta	Baja (boosting amplifica outliers)
Performance típico tabular	Bueno	Mejor (en general)

En la práctica: si tu dataset tiene mucho ruido en la variable objetivo o no tienes tiempo para tunear, RF. Si quieres extraer hasta el último 1% de performance y puedes tunear, XGBoost. Si no estás seguro, prueba ambos.