

Privacidad Diferencial

Basado en Programming Differential Privacy, de Joseph P. Near y Chiké Abuah

Alvaro J. Riascos Villegas

Universidad de los Andes & Quantil

24 septiembre 2026

Hoja de ruta

¿Cómo publicamos estadísticas útiles sobre un conjunto de personas sin revelar nada sobre *ninguna* de ellas en particular?

1. **Privacidad diferencial** — la definición y el mecanismo de Laplace
2. **Propiedades** — post-procesamiento y composición
3. **Sensibilidad** — cuánto ruido hace falta
4. **Privacidad diferencial aproximada** — (ϵ, δ) y el mecanismo Gaussiano
5. **Aprendizaje automático** — descenso de gradiente privado

Todo el código de estas slides se **ejecuta al renderizar**: los números y las figuras que verás se calculan en el momento.

El dato de partida

Trabajamos con un censo sintético de 5.000 personas, con la estructura del dataset *Adult* del US Census que usa el libro.

```
censo.head(4)
```

	nombre	edad	educacion	ganancia_capital	ingreso_alto
0	Karen	30	10	0.0	1
1	P0001	17	16	0.0	1
2	P0002	32	13	0.0	0
3	P0003	49	13	0.0	1

Los agregados no son anónimos

Publicamos solo un **conteo**. Sin nombres, sin identificadores. Parece seguro. Consideremos el caso de Karen.

```
def cuenta_ingresos_altos(df):  
    return len(df[df.ingreso_alto == 1])  
  
total = cuenta_ingresos_altos(censo)  
sin_karen = cuenta_ingresos_altos(censo[censo.nombre != "Karen"])  
  
print(f"Personas con ingreso alto:      {total}")  
print(f"Idem, excluyendo a Karen:      {sin_karen}")  
print(f"Diferencia -> ingreso alto de Karen: {total - sin_karen}")
```

```
Personas con ingreso alto:      2738  
Idem, excluyendo a Karen:      2737  
Diferencia -> ingreso alto de Karen: 1
```

Ataque diferencial. Dos agregados, restados, reconstruyen un dato individual.

1 • Privacidad diferencial

La intuición: negación plausible

Si el resultado publicado hubiera sido **casi el mismo** con o sin Karen en la base, entonces observarlo no dice nada sobre ella.

Karen puede sostener: *ese resultado se habría producido igual si yo no estuviera en los datos*. Y sería creíble.

Formalizamos “casi el mismo” con dos piezas:

- **Bases vecinas:** D y D' difieren en **un solo individuo**
- **Indistinguibilidad:** el mecanismo asigna probabilidades similares a cada salida, corra sobre D o sobre D'

La definición

ϵ -privacidad diferencial. Un mecanismo aleatorio M satisface ϵ -privacidad diferencial si para todo par de bases vecinas D, D' y todo conjunto de salidas S :

$$\frac{\Pr[M(D) \in S]}{\Pr[M(D') \in S]} \leq e^\epsilon$$

Tres lecturas de la misma desigualdad:

- M es aleatorio. Un mecanismo determinista nunca es privado, salvo que ignore por completo los datos.
- La desigualdad se puede expresar de forma equivalente como: el valor absoluto de la diferencia logarítmica de las probabilidades es menor o igual a ϵ .
- **Es una garantía en el peor caso.** Vale para *todo* par de vecinas y *toda* salida: no depende de qué sepa el atacante.
- ϵ es el presupuesto. Con $\epsilon \rightarrow 0$ hay ruido puro (mayor privacidad); con ϵ grande, menos privacidad.

¿Qué tan grande es e^ϵ ?

ϵ acota cuánto puede cambiar la creencia de un atacante sobre Karen.

```
epsilon = 0.01 -> razon de probabilidades <= e^eps = 1.01
epsilon = 0.10 -> razon de probabilidades <= e^eps = 1.11
epsilon = 0.50 -> razon de probabilidades <= e^eps = 1.65
epsilon = 1.00 -> razon de probabilidades <= e^eps = 2.72
epsilon = 2.00 -> razon de probabilidades <= e^eps = 7.39
epsilon = 10.00 -> razon de probabilidades <= e^eps = 22026.47
```

$\epsilon \leq 1$ se considera protección fuerte; $\epsilon \approx 10$, poca privacidad. No hay umbral universal: es una decisión del investigador.

El mecanismo de Laplace

La receta más simple que satisface la definición: **sumar ruido de Laplace** calibrado al presupuesto.

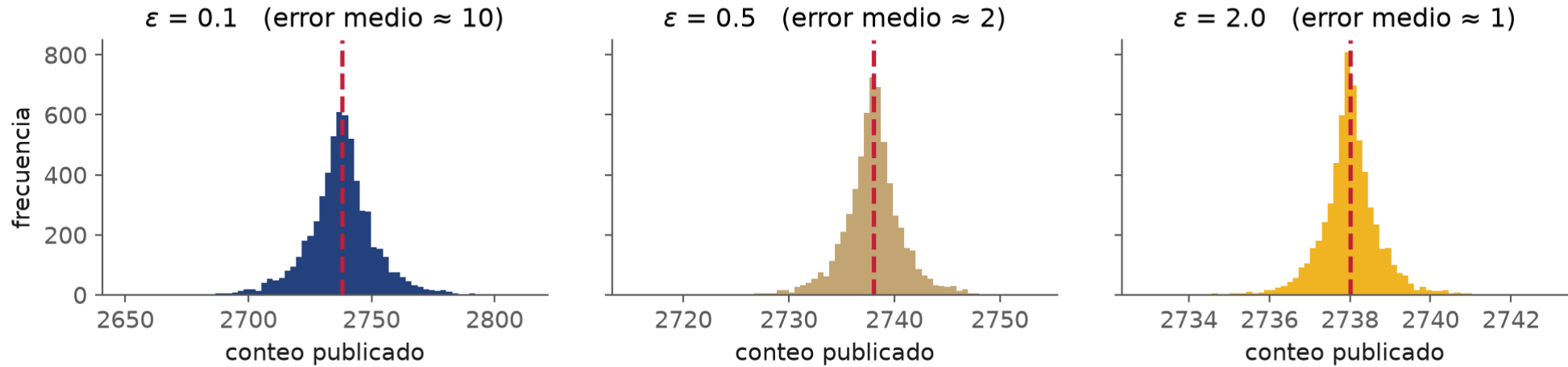
```
def mecanismo_laplace(valor, sensibilidad, epsilon):  
    """Devuelve valor + ruido Laplace(0, sensibilidad/epsilon)."""  
    escala = sensibilidad / epsilon  
    return valor + np.random.laplace(loc=0.0, scale=escala)  
  
print(f"Conteo real:          {total}")  
for _ in range(3):  
    print(f"Conteo con eps = 1.0: {mecanismo_laplace(total, 1, 1.0):.2f}")
```

```
Conteo real:          2738  
Conteo con eps = 1.0: 2737.55  
Conteo con eps = 1.0: 2737.43  
Conteo con eps = 1.0: 2739.28
```

La sensibilidad de un conteo es **1**: una persona cambia el resultado en, como máximo, uno. Volveremos sobre esto en el capítulo 3.

ϵ determina la sensibilidad

Respuesta ruidosa al conteo (valor real = 2738, línea roja)



Menos ϵ significa más dispersión: más privacidad y menos utilidad. **El mismo parámetro gobierna ambas cosas.**

¿Neutraliza el ataque diferencial?

Repetimos el ataque, pero ahora las dos consultas pasan por el mecanismo.

```
ensayos = np.array([
    mecanismo_laplace(total, 1, 0.5) - mecanismo_laplace(sin_karen, 1, 0.5)
    for _ in range(10_000)
])
acierta = np.mean(np.round(np.clip(ensayos, 0, 1)) == 1)

print(f"Respuesta real del ataque:    {total - sin_karen}")
print(f"Estimacion media del atacante: {ensayos.mean():.3f}")
print(f"Desviacion estandar:          {ensayos.std():.2f}")
print(f"Acierta al redondear:          {acierta:.1%}    (azar = 50%)")
```

```
Respuesta real del ataque:    1
Estimacion media del atacante: 1.003
Desviacion estandar:          3.98
Acierta al redondear:          56.1%    (azar = 50%)
```

La señal que busca el atacante vale **1**; el ruido tiene desviación estándar de varias unidades. La señal queda enterrada.

Pregunta

¿Por qué la definición exige que la garantía valga para *todo* par de bases vecinas, y no en promedio sobre las bases plausibles?

Porque una garantía promedio se rompe justo para quien más la necesita: el individuo atípico. El millonario en un censo de ingresos, el paciente con la enfermedad rara.

Una garantía en el peor caso es además la única que sobrevive a un atacante con **conocimiento auxiliar arbitrario**: no necesitamos suponer qué sabe.

Es la diferencia con la anonimización clásica: k -anonimato protege contra los ataques que *imaginaste*; la privacidad diferencial, contra todos.

2 · Propiedades de la privacidad diferencial

Por qué una definición no basta

Nadie publica **una sola** consulta. Un instituto de estadística publica docenas de tablas; un modelo se entrena con miles de iteraciones.

Necesitamos saber qué pasa cuando los resultados privados se **combinan**, se **transforman** o se **repiten**. Tres propiedades responden eso:

- **Post-procesamiento** — transformar una salida privada no gasta presupuesto
- **Composición secuencial** — consultas repetidas suman sus ϵ
- **Composición paralela** — consultas sobre partes disjuntas no suman

Post-procesamiento

Si M es ε -DP y f es cualquier función que **no vuelve a mirar los datos**, entonces $f(M(D))$ también es ε -DP.

```
bruto = mecanismo_laplace(total, 1, 1.0)

redondeado = max(0, round(bruto))      # conteos negativos no tienen sentido
proporcion = redondeado / len(censo)   # reescalar a proporción

print(f"Salida cruda del mecanismo: {bruto:.3f}")
print(f"Redondeada y truncada:      {redondeado}      <- sigue siendo 1.0-DP")
print(f"Como proporción:            {proporcion:.4f}   <- sigue siendo 1.0-DP")
```

```
Salida cruda del mecanismo: 2739.311
Redondeada y truncada:      2739      <- sigue siendo 1.0-DP
Como proporción:            0.5478   <- sigue siendo 1.0-DP
```

Podemos limpiar, reescalar y graficar los resultados. La información privada ya fue protegida; ninguna función puede recuperarla.

Composición secuencial

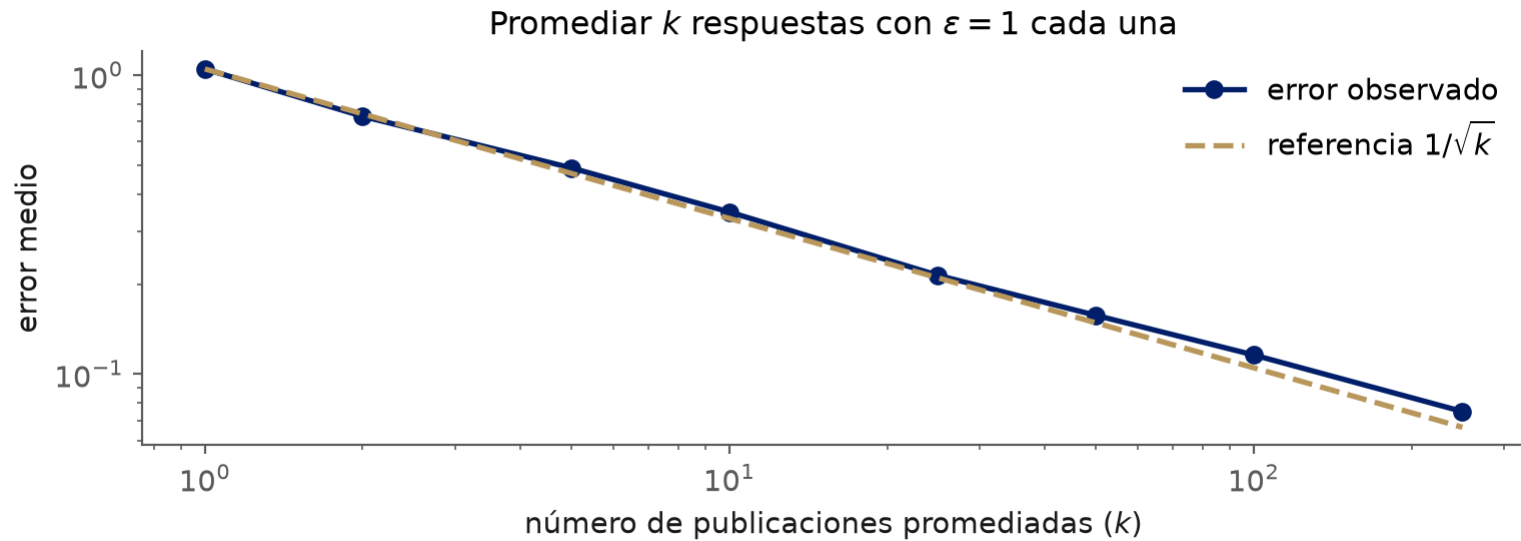
Si M_1 es ε_1 -DP y M_2 es ε_2 -DP, publicar ambas es $(\varepsilon_1 + \varepsilon_2)$ -DP.

El presupuesto se **gasta**: repetir una consulta permite promediar el ruido hasta hacerlo desaparecer.

```
def error_al_promediar(k, epsilon=1.0, repeticiones=400):  
    """Error medio al promediar k respuestas ruidosas independientes."""  
    estimaciones = np.array([  
        np.mean([mecanismo_laplace(total, 1, epsilon) for _ in range(k)])  
        for _ in range(repeticiones)  
    ])  
    return np.mean(np.abs(estimaciones - total))  
  
for k in [1, 10, 100]:  
    err = error_al_promediar(k)  
    print(f"k = {k:3d} publicaciones -> error medio = {err:6.3f}")
```

```
k = 1 publicaciones -> error medio = 0.973  
k = 10 publicaciones -> error medio = 0.346  
k = 100 publicaciones -> error medio = 0.109
```

El ruido no se acumula: se cancela



Con k publicaciones el error cae como $1/\sqrt{k}$: con suficientes consultas el atacante **recupera el valor exacto**. Por eso el presupuesto debe sumar.

Composición paralela

Si los M_i son ε -DP y se aplican sobre **subconjuntos disjuntos** de los datos, el total es ε -DP — no $k\varepsilon$.

Un histograma es el caso canónico: cada persona cae en **exactamente un** nivel educativo, así que solo afecta a una barra.

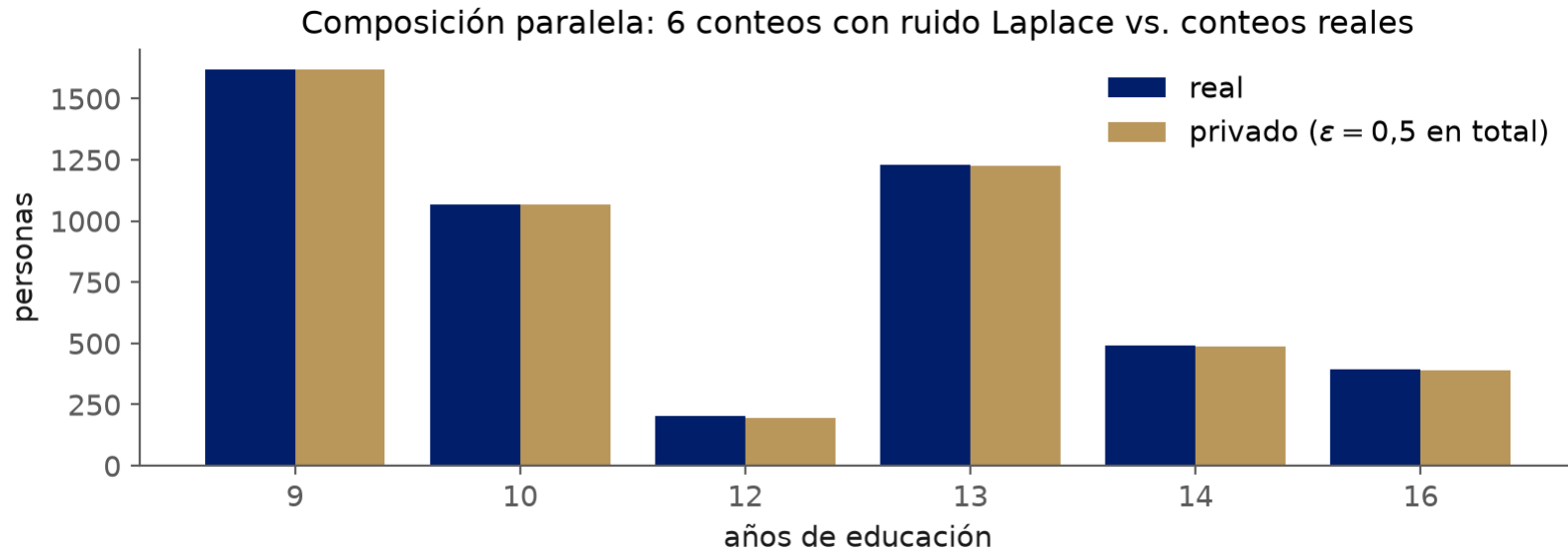
```
conteos = censo.educacion.value_counts().sort_index()
ruidosos = {nivel: mecanismo_laplace(c, 1, 0.5) for nivel, c in conteos.items()}

for nivel in list(conteos.index)[:4]:
    print(f"educacion = {nivel:2d} años | real: {conteos[nivel]:5d} | "
          f"privado: {ruidosos[nivel]:9.2f}")
print(f"\nCosto total del histograma completo: eps = 0.5 (no 0.5 x "
      f"{len(conteos)} = {0.5*len(conteos)})")
```

```
educacion = 9 años | real: 1620 | privado: 1617.71
educacion = 10 años | real: 1067 | privado: 1068.83
educacion = 12 años | real: 202 | privado: 196.46
educacion = 13 años | real: 1228 | privado: 1225.96

Costo total del histograma completo: eps = 0.5 (no 0.5 x 6 = 3.0)
```

El histograma privado



La diferencia es visualmente imperceptible porque el ruido es de orden $1/0,5 = 2$ personas sobre barras de cientos. **Los histogramas son el caso exitoso de la privacidad diferencial.**

3 · Sensibilidad

¿Cuánto ruido es suficiente?

Hasta ahora usamos sensibilidad = 1 sin justificarlo. Esa es la pieza que conecta la **consulta** con el **ruido**.

Sensibilidad global. Para una consulta f ,

$$\Delta f = \max_{D, D' \text{ vecinas}} |f(D) - f(D')|$$

es decir: cuánto puede mover el resultado **una sola persona**, en el peor caso.

El mecanismo de Laplace con escala $\Delta f / \epsilon$ satisface ϵ -DP. Todo el trabajo está en calcular Δf correctamente.

Conteos: sensibilidad 1

```
def universitarios(df):  
    return len(df[df.educacion >= 13])  
  
dentro = censo.index[censo.educacion >= 13][0] # alguien que SI cuenta  
fuera = censo.index[censo.educacion < 13][0] # alguien que NO cuenta  
  
print(f"Base completa: {universitarios(censo)}")  
print(f"Quitando a quien SI cuenta: {universitarios(censo.drop(dentro))}")  
print(f"Quitando a quien NO cuenta: {universitarios(censo.drop(fuera))}")
```

```
Base completa: 2111  
Quitando a quien SI cuenta: 2110  
Quitando a quien NO cuenta: 2111
```

Agregar o quitar una persona cambia un conteo en **0** o en **1**. Nunca más. $\Delta f = 1$, independientemente del tamaño de la base.

Por eso los conteos son baratos: el ruido necesario **no crece** con n , así que el error *relativo* se desvanece en bases grandes.

Sumas: el problema

```
suma_real = censo.ganancia_capital.sum()
maximo = censo.ganancia_capital.max()

print(f"Suma de ganancias de capital: {suma_real:>15,.0f}")
print(f"Maximo individual observado: {maximo:>15,.0f}")
print(f"Sensibilidad = el mayor valor *posible*, no el observado -> sin cota")
```

```
Suma de ganancias de capital:      4,224,162
Maximo individual observado:       545,772
Sensibilidad = el mayor valor *posible*, no el observado -> sin cota
```

Nadie impide que exista una persona con una ganancia de capital de mil millones. La sensibilidad de una suma sin restricciones es **infinita**, y un ruido infinito destruye toda utilidad.

La solución: recortar (*clipping*)

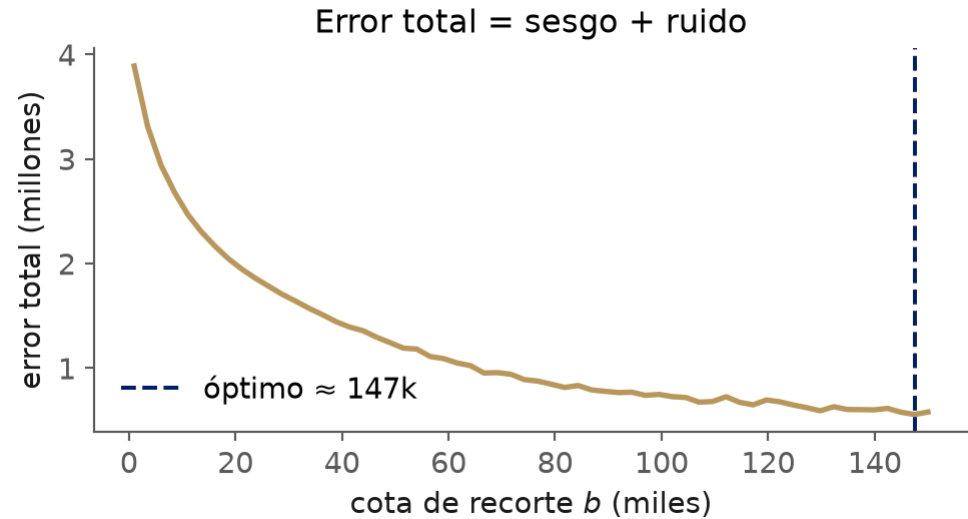
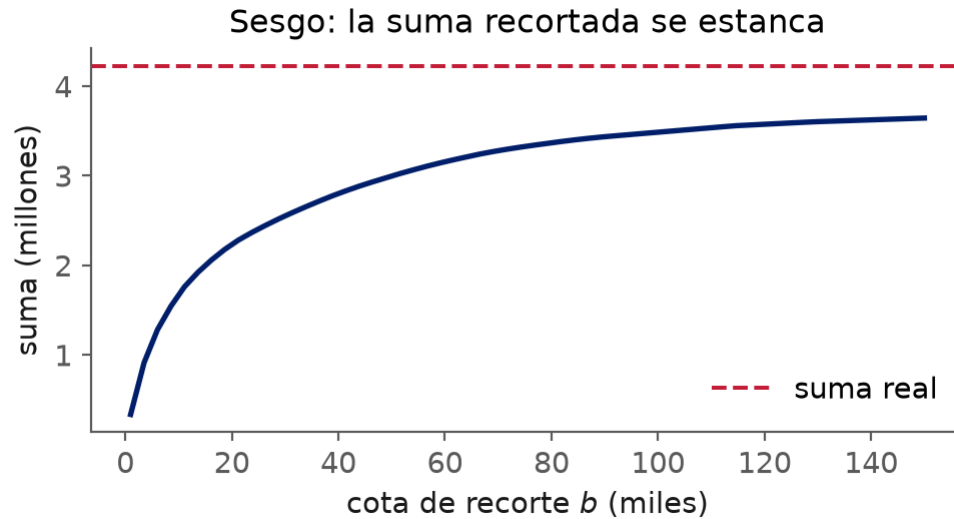
Forzamos una cota. Si ningún valor puede exceder b , entonces $\Delta f = b$ por construcción.

```
def suma_recortada(serie, b):  
    return serie.clip(upper=b).sum()  
  
print(f"{'b':>9} | {'suma recortada':>15} | {'sesgo':>7} | {'ruido':>8}")  
print("-" * 50)  
for b in [5_000, 20_000, 100_000, 500_000]:  
    real = suma_recortada(censo.ganancia_capital, b)  
    desvios = [abs(mecanismo_laplace(real, b, 1.0) - real) for _ in range(300)]  
    sesgo = (real - suma_real) / suma_real  
    print(f"{'b':>9,} | {'real':>15,.0f} | {'sesgo':>6.1%} | "  
          f"{'np.mean(desvios)/suma_real':>7.1%}")
```

b	suma recortada	sesgo	ruido
5,000	1,146,931	-72.8%	0.1%
20,000	2,233,632	-47.1%	0.5%
100,000	3,485,208	-17.5%	2.4%
500,000	4,178,390	-1.1%	12.4%

El dilema del recorte: b pequeño sesga la suma hacia abajo; b grande exige más ruido. Hay un óptimo intermedio.

Encontrar el codo



A la izquierda, el sesgo desaparece al crecer b . A la derecha, el ruido lo compensa: el **error total** tiene un mínimo interior.

Elegir b sin mirar los datos

Problema circular: el b óptimo depende de los datos, y mirarlos cuesta presupuesto. La salida del libro: **buscar el codo con consultas privadas**.

```
def cota_privada(serie, epsilon, candidatos):
    """Sube b hasta que la suma recortada deja de crecer. Gasta epsilon."""
    eps_i = epsilon / len(candidatos)    # composición secuencial
    anterior = None
    for b in candidatos:
        actual = mecanismo_laplace(suma_recortada(serie, b), b, eps_i)
        if anterior is not None and actual <= anterior * 1.02:
            return b, actual
        anterior = actual
    return candidatos[-1], anterior

b_elegido, _ = cota_privada(censo.ganancia_capital, 0.5,
                           [2_000 * 2**i for i in range(8)])
print(f"Cota elegida de forma privada: b = {b_elegido:,}")
print(f"Costo: eps = 0.5, pagado ANTES de publicar la suma")
```

```
Cota elegida de forma privada: b = 32,000
Costo: eps = 0.5, pagado ANTES de publicar la suma
```

Honestidad contable: el presupuesto gastado en elegir b **cuenta**. Es composición secuencial, no un preprocesamiento gratuito.

Promedios: dos consultas, no una

Un promedio no es una consulta lineal, así que no tiene sensibilidad simple. La descomponemos y usamos composición secuencial.

```
def media_privada(serie, cota, epsilon):
    suma = mecanismo_laplace(serie.clip(upper=cota).sum(), cota, epsilon / 2)
    conteo = mecanismo_laplace(len(serie), 1, epsilon / 2)
    return suma / conteo

print(f"Edad media real:      {censo.edad.mean():.4f}")
for _ in range(3):
    m = media_privada(censo.edad, 90, 1.0)
    print(f"Edad media privada: {m:.4f}    (eps = 1.0)")
```

```
Edad media real:      37.6352
Edad media privada:  37.6372    (eps = 1.0)
Edad media privada:  37.6700    (eps = 1.0)
Edad media privada:  37.6075    (eps = 1.0)
```

El presupuesto se parte: $\epsilon/2$ para la suma (sensibilidad **90**, la edad máxima) y $\epsilon/2$ para el conteo (sensibilidad **1**).

El espejismo de la sensibilidad local

Tentación: usar la sensibilidad **en la base que tenemos**, no en el peor caso. Sería mucho más chica.

Y rompe la garantía. La sensibilidad local depende de los datos; si la escala del ruido depende de los datos, la escala misma **filtra información**. Un atacante que observa un ruido pequeño aprende que la base no tiene valores extremos.

Las alternativas legítimas son más elaboradas:

- **Smooth sensitivity** — suaviza la sensibilidad local sobre bases cercanas
- **Sample and aggregate** — particiona, resuelve por partes, agrega con un mecanismo cuya sensibilidad sí está acotada
- **Propose-test-release** — verifica *privadamente* que la cota propuesta sirve

Todas comparten el principio: **nada que dependa de los datos puede escapar al presupuesto.**

4 • Privacidad diferencial aproximada

Por qué relajar la definición

La ε -DP pura es exigente: prohíbe **cualquier** salida que sea más de e^ε veces más probable bajo D que bajo D' , incluso las de probabilidad ínfima.

Relajar la definición con un término aditivo δ compra tres cosas:

- acceso al **mecanismo Gaussiano** y a la sensibilidad L_2
- **composición avanzada**, que crece como $\sqrt{k}$ en vez de k
- mejor comportamiento en **consultas de alta dimensión**

La definición

(ϵ, δ) -privacidad diferencial. Para todo par de bases vecinas D, D' y todo conjunto de salidas S :

$$\Pr[M(D) \in S] \leq e^\epsilon \Pr[M(D') \in S] + \delta$$

El término δ es una **probabilidad de falla del mecanismo**: con probabilidad $\leq \delta$ la garantía de ϵ -DP simplemente no aplica.

La regla práctica sobre δ

Elegir $\delta \ll 1/n$. Con $\delta \geq 1/n$ el mecanismo “publicar los datos de una persona al azar” satisface técnicamente la definición — y es evidentemente inaceptable.

La convención habitual es $\delta \approx 1/n^2$ o, directamente, $\delta = 10^{-5}$ para bases del orden de decenas de miles.

Diferencia conceptual clave:

- ϵ acota la **cantidad** de información filtrada, siempre
- δ acota la **probabilidad** de que esa cota no valga

Son dos ejes distintos, no dos formas del mismo parámetro.

Sensibilidad L_1 y L_2

Cuando la consulta devuelve un **vector**, hay dos formas de medir cuánto lo mueve una persona.

$$\Delta_1 f = \max_{D, D'} \|f(D) - f(D')\|_1 \quad \Delta_2 f = \max_{D, D'} \|f(D) - f(D')\|_2$$

dimension k	L1 = k	L2 = sqrt(k)	razon
1	1	1.00	1.00x
4	4	2.00	2.00x
16	16	4.00	4.00x
100	100	10.00	10.00x
1000	1000	31.62	31.62x

Para una persona que afecta k componentes en **1** cada una: $\Delta_1 = k$ pero $\Delta_2 = \sqrt{k}$. **Laplace necesita Δ_1** ; Gauss se conforma con Δ_2 . Ahí está la ganancia.

El mecanismo Gaussiano: definición

Misma receta que Laplace — sumar ruido calibrado a la sensibilidad — pero con ruido **normal** y calibrado a Δ_2 .

Mecanismo Gaussiano. Sea f una consulta con valores en $\mathbb{R}^k$ y sensibilidad $\Delta_2 f$. Para $\varepsilon \in (0, 1)$ y $\delta > 0$, el mecanismo

$$M(D) = f(D) + Z, \quad Z \sim \mathcal{N}(0, \sigma^2 I_k), \quad \sigma = \frac{\Delta_2 f \sqrt{2 \ln(1, 25/\delta)}}{\varepsilon}$$

satisface (ε, δ) -privacidad diferencial.

Frente a Laplace, $M(D) = f(D) + \text{Lap}(\Delta_1 f / \varepsilon)$ en cada componente:

- **Sensibilidad:** usa $\Delta_2 f$ en vez de $\Delta_1 f$ — la ventaja en alta dimensión
- **Precio:** la garantía es (ε, δ) -DP, no ε -DP pura
- **Rango de validez:** este análisis clásico (Dwork y Roth, 2014) exige $\varepsilon < 1$

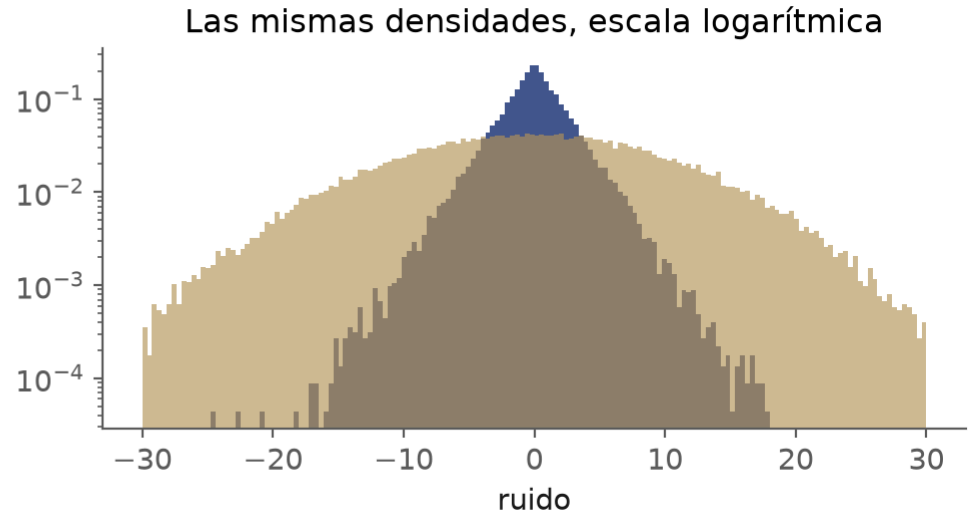
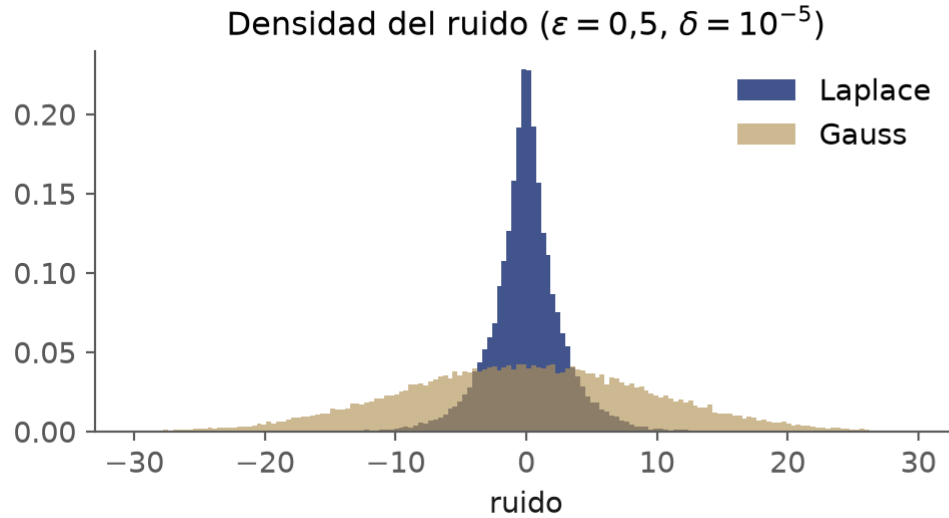
El mecanismo Gaussiano en código

```
def mecanismo_gaussiano(valor, sens_l2, epsilon, delta):  
    """Valido para epsilon < 1 (analisis clasico de Dwork & Roth)."""  
    sigma = np.sqrt(2 * np.log(1.25 / delta)) * sens_l2 / epsilon  
    return valor + np.random.normal(0.0, sigma, size=np.shape(valor))  
  
print(f"Conteo real: {total}")  
for _ in range(3):  
    print(f"Gaussiano (eps=0.5, delta=1e-5): "  
          f"{mecanismo_gaussiano(float(total), 1.0, 0.5, 1e-5):.2f}")
```

```
Conteo real: 2738  
Gaussiano (eps=0.5, delta=1e-5): 2749.89  
Gaussiano (eps=0.5, delta=1e-5): 2727.40  
Gaussiano (eps=0.5, delta=1e-5): 2727.91
```

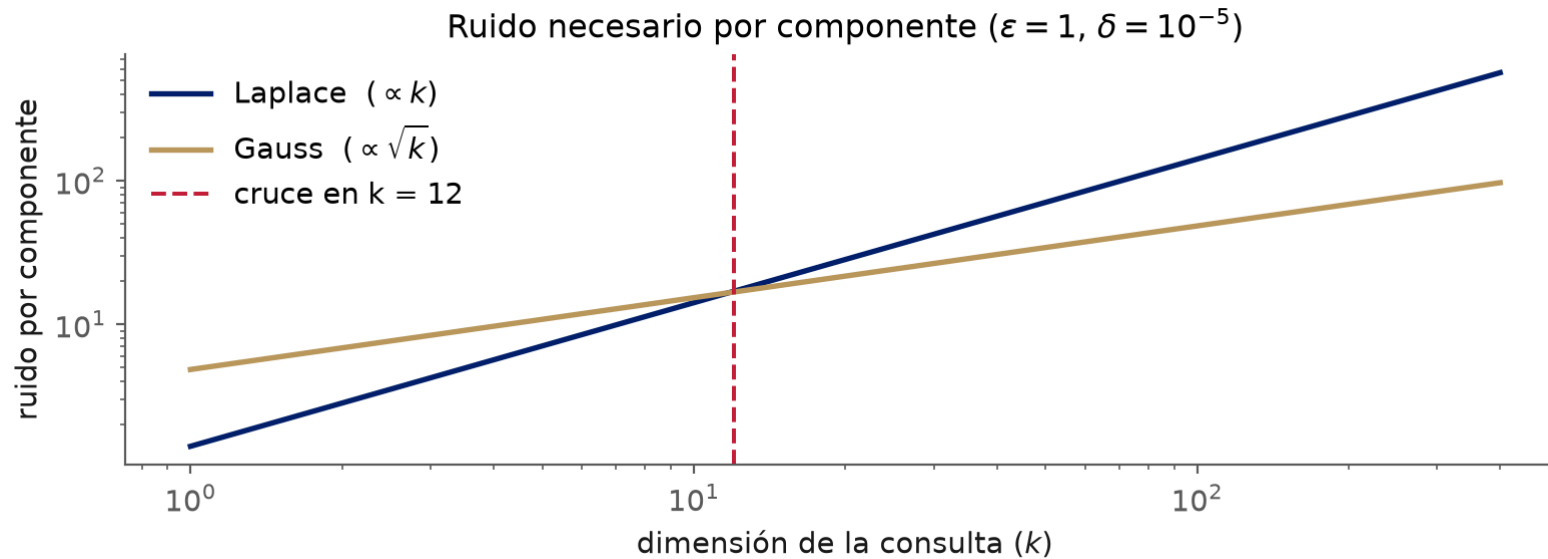
Nótese el precio: para una consulta escalar, σ incorpora el factor $\sqrt{2\ln(1,25/\delta)} \approx 4,8$ cuando $\delta = 10^{-5}$.

Laplace contra Gauss: escalar



En una consulta escalar **Laplace gana**: con el mismo ϵ concentra mucha más masa cerca de cero, mientras Gauss paga el precio de δ sin recibir nada a cambio.

Laplace contra Gauss: vectorial



A partir de $k \approx 12$ componentes, Gauss domina — y sigue mejorando. **Por eso el aprendizaje automático privado usa Gauss:** un gradiente tiene tantas componentes como parámetros el modelo.

Composición avanzada

La relajación también mejora la contabilidad del presupuesto.

k mecanismos $(\varepsilon, 0)$ -DP componen a (ε', δ') -DP, para cualquier $\delta' > 0$ que elijamos, con

$$\varepsilon' = 2\varepsilon \sqrt{2k \ln(1/\delta')}$$

k	secuencial	avanzada	mejor
1	0.10	0.96	secuencial
10	1.00	3.03	secuencial
100	10.00	9.60	avanzada
1000	100.00	30.35	avanzada
10000	1000.00	95.97	avanzada

Crece como $\sqrt{k}$ en vez de k , pero con un factor constante alto: **solo conviene cuando k** es grande. Para pocas consultas, la composición simple gana.

Variantes

La composición avanzada sigue siendo mediocre para el aprendizaje automático, donde k son miles de iteraciones. De ahí nacieron definiciones diseñadas para **componer bien**.

- **Rényi DP** — divergencia de Rényi de orden α ; composición: suma exacta, sin pérdida

Rényi DP: componer sin pérdida

El problema de fondo: (ϵ, δ) -DP acota la **cola** de la pérdida de privacidad. Componer cotas de cola es intrínsecamente costoso — cada composición vuelve a aplicar un argumento de peor caso sobre otro.

$(\alpha, \bar{\epsilon})$ -privacidad diferencial de Rényi. M la satisface si para todo par de bases vecinas D, D' :

$$D_{\alpha}(M(D) \parallel M(D')) \leq \bar{\epsilon}$$

donde D_{α} es la divergencia de Rényi de orden $\alpha > 1$.

En vez de acotar una cola, acota una **divergencia**. Y las divergencias se suman:

- **Composición exacta:** $(\alpha, \bar{\epsilon}_1)$ y $(\alpha, \bar{\epsilon}_2)$ componen a $(\alpha, \bar{\epsilon}_1 + \bar{\epsilon}_2)$. Sin pérdida y sin δ .
- **El mecanismo Gaussiano encaja perfecto:** con sensibilidad Δ y ruido σ satisface $(\alpha, \alpha\Delta^2/2\sigma^2)$ -RDP para *todo* α a la vez.
- **zCDP** es el caso particular en que la cota crece lineal en α : un único parámetro resume toda la familia.

Las tres contabilidades, lado a lado

Mismo mecanismo Gaussiano repetido k veces ($\sigma = 20, \Delta = 1, \delta = 10^{-5}$). Solo cambia **cómo llevamos la cuenta**.

```
sigma, delta = 20.0, 1e-5
L = np.log(1 / delta)

def eps_secuencial(k):      # sumar los epsilon, uno por iteracion
    return k * np.sqrt(2 * np.log(1.25 / (delta / k))) / sigma

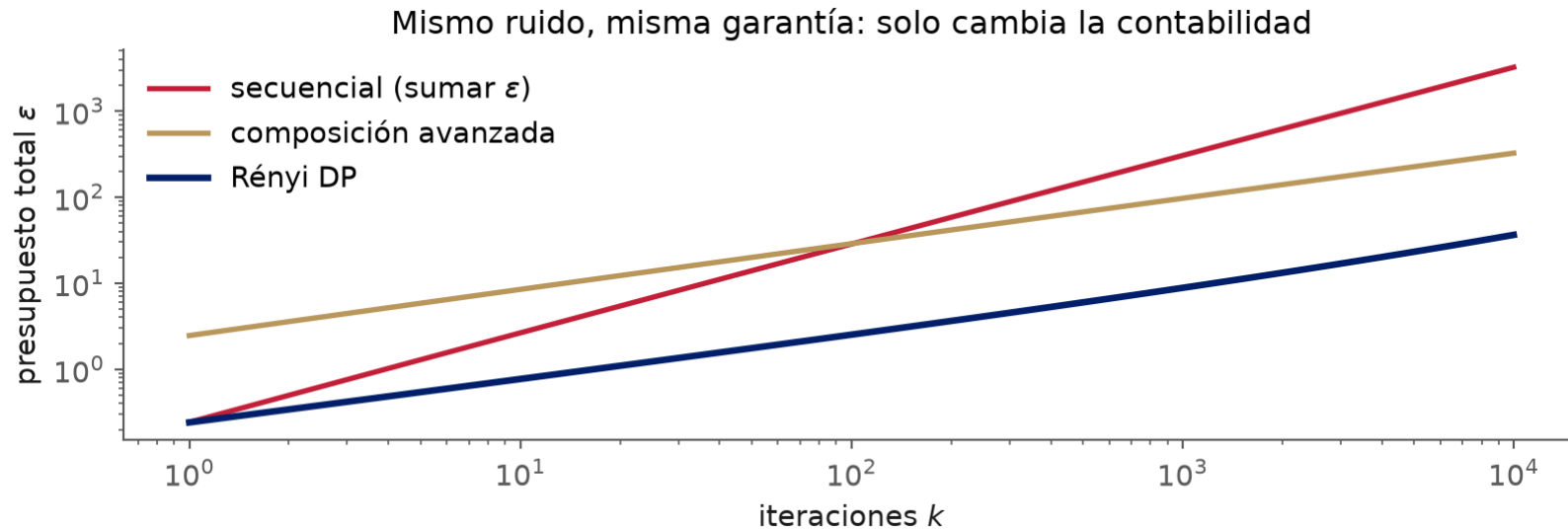
def eps_avanzada(k):        # composicion avanzada sobre los mismos k
    e_i = np.sqrt(2 * np.log(1.25 / (delta / (2 * k)))) / sigma
    return 2 * e_i * np.sqrt(2 * k * np.log(2 / delta))

def eps_rdp(k):             # componer en RDP y convertir una sola vez
    return k / (2 * sigma**2) + np.sqrt(2 * k * L) / sigma

for k in [1, 100, 10_000]:
    print(f"k = {k:>6} | secuencial {eps_secuencial(k):>8.1f} | "
          f"avanzada {eps_avanzada(k):>7.1f} | RDP {eps_rdp(k):>6.1f}")
```

k =	1		secuencial	0.2		avanzada	2.5		RDP	0.2
k =	100		secuencial	28.6		avanzada	28.8		RDP	2.5
k =	10000		secuencial	3236.2		avanzada	325.0		RDP	36.5

Cuánto se gana



La conversión final es $(\alpha, \bar{\epsilon})$ -RDP $\Rightarrow (\bar{\epsilon} + \ln(1/\delta)/(\alpha - 1), \delta)$ -DP: se compone en RDP y se optimiza sobre α **al final**, convirtiendo una sola vez.

Con $k = 10^4$ el presupuesto cae de $\epsilon \approx 3236$ a ≈ 36 **sin tocar el ruido**. Esto es lo que hace viable el DP-SGD del capítulo siguiente.

5 · Aprendizaje automático

El problema

Queremos entrenar un clasificador que prediga ingreso alto a partir de edad y educación — **sin que el modelo memorice a nadie.**

```
X_bruto = censo[["edad", "educacion"]].values.astype(float)
X = (X_bruto - X_bruto.mean(0)) / X_bruto.std(0)
X = np.hstack([X, np.ones((len(X), 1))]) # término constante
y = np.where(censo.ingreso_alto.values == 1, 1, -1)

corte = 4_000
X_ent, y_ent, X_test, y_test = X[:corte], y[:corte], X[corte:], y[corte:]

def exactitud(theta):
    return np.mean(np.sign(X_test @ theta) == y_test)

print(f"Entrenamiento: {len(X_ent)} filas | Prueba: {len(X_test)} filas")
piso_trivial = max(np.mean(y_test == 1), np.mean(y_test == -1))
print(f"Clase mayoritaria (piso trivial): {piso_trivial:.3f}")
```

```
Entrenamiento: 4000 filas | Prueba: 1000 filas
Clase mayoritaria (piso trivial): 0.546
```

Dónde vive la información privada

Regresión logística por descenso de gradiente. El gradiente es una **suma sobre personas** — y cada término depende de una sola.

```
def gradientes(theta, X, y):  
    """Matriz de gradientes: una fila por persona."""  
    margen = y * (X @ theta)  
    return (-y * expit(-margen))[:, None] * X  
  
g = gradientes(np.zeros(3), X_ent, y_ent)  
print(f"Forma de la matriz de gradientes: {g.shape} (una fila por persona)")  
print(f"Gradiente de Karen: {g[0].round(4)}")  
print(f"Gradiente agregado (lo que usamos): {g.sum(axis=0).round(2)}")
```

```
Forma de la matriz de gradientes: (4000, 3) (una fila por persona)  
Gradiente de Karen: [ 0.305  0.2961 -0.5  ]  
Gradiente agregado (lo que usamos): [-481.78 -566.81 -192.  ]
```

Publicar el gradiente agregado en cada iteración es publicar una consulta de suma. Ya sabemos protegerla: **recortar y sumar ruido**.

Recortar gradientes

La sensibilidad de la suma de gradientes no está acotada — igual que la suma de ganancias de capital. Misma solución: recortar, ahora en norma L_2 .

```
def recortar_filas(G, b):  
    """Escala cada fila para que su norma L2 no exceda b."""  
    normas = np.linalg.norm(G, axis=1, keepdims=True)  
    return G * np.minimum(1.0, b / np.maximum(normas, 1e-12))  
  
normas = np.linalg.norm(g, axis=1)  
print(f"Norma L2 de los gradientes: mediana {np.median(normas):.3f}, "  
      f"maxima {normas.max():.3f}")  
tras = np.linalg.norm(recortar_filas(g, 1.0), axis=1).max()  
print(f"Tras recortar con b = 1:   maxima {tras:.3f}")  
print(f"Sensibilidad L2 de la suma recortada = b = 1 (por construccion)")
```

```
Norma L2 de los gradientes: mediana 0.779, maxima 2.135  
Tras recortar con b = 1:   maxima 1.000  
Sensibilidad L2 de la suma recortada = b = 1 (por construccion)
```

Descenso de gradiente privado

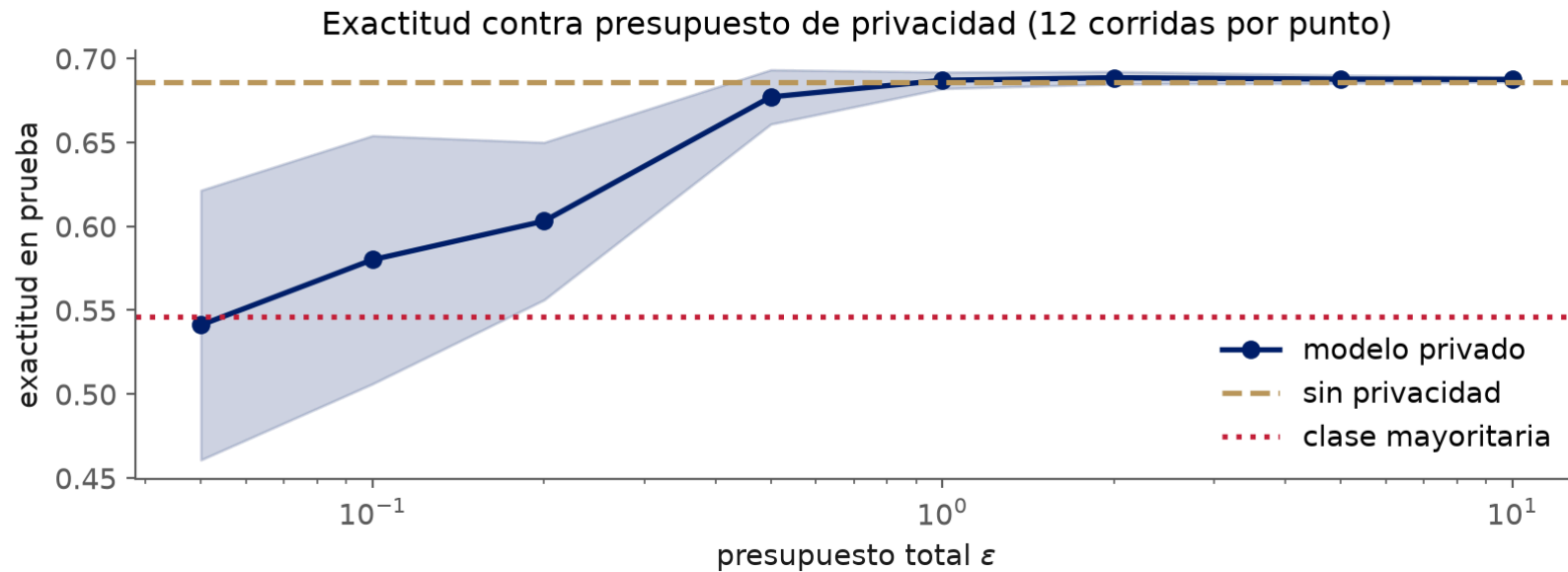
Juntamos todo: recorte por persona, ruido Gaussiano sobre la suma, y composición secuencial a lo largo de las iteraciones.

```
def descenso_privado(X, y, iteraciones=30, epsilon=1.0, delta=1e-5,
                    b=1.0, eta=2.0):
    theta = np.zeros(X.shape[1])
    for _ in range(iteraciones):
        G = recortar_filas(gradientes(theta, X, y), b)
        suma = G.sum(axis=0)
        if epsilon is not None:          # el presupuesto se reparte
            suma = mecanismo_gaussiano(suma, b, epsilon / iteraciones,
                                       delta / iteraciones)
        theta = theta - eta * suma / len(X)
    return theta

theta_np = descenso_privado(X_ent, y_ent, epsilon=None) # referencia sin ruido
print(f"Exactitud sin privacidad:      {exactitud(theta_np):.4f}")
for eps in [10.0, 1.0, 0.1]:
    acc = exactitud(descenso_privado(X_ent, y_ent, epsilon=eps))
    print(f"Exactitud con eps = {eps:>5}:    {acc:.4f}")
```

```
Exactitud sin privacidad:      0.6860
Exactitud con eps = 10.0:      0.6880
Exactitud con eps = 1.0:       0.6760
Exactitud con eps = 0.1:       0.6380
```

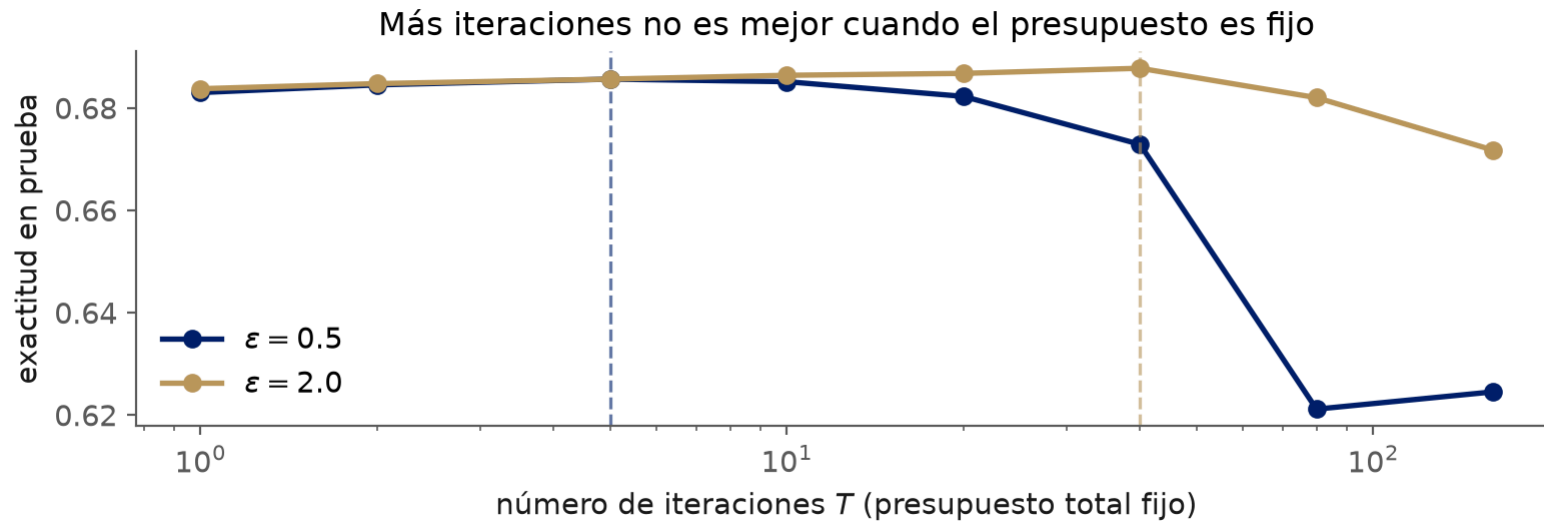
El compromiso privacidad-utilidad



La curva es la respuesta honesta a “¿cuánto cuesta la privacidad?”: **depende del régimen**. Arriba de $\epsilon \approx 1$ el costo es casi nulo; abajo de $0,2$ el modelo colapsa hacia la clase mayoritaria.

El costo de iterar

Más iteraciones convergen mejor, pero cada una gasta ε/T : el ruido por paso **crece** con T . Hay un óptimo interior.



Con presupuesto fijo, el número de iteraciones pasa de ser un detalle de implementación a ser un **hiperparámetro de privacidad**. Y ajustarlo mirando los datos también cuesta presupuesto.

DP-SGD en la práctica

Lo que implementamos es *noisy gradient descent* de lote completo. El algoritmo estándar, **DP-SGD**, agrega dos piezas:

- **Muestreo de Poisson** de lotes, que aporta *privacy amplification by subsampling*: muestrear una fracción q de los datos reduce el ϵ efectivo aproximadamente en un factor q
- **Contabilidad por momentos** (*moments accountant* / Rényi DP) en lugar de composición secuencial, lo que recorta el presupuesto de forma drástica en miles de iteraciones

Ambas piezas atacan lo mismo: el presupuesto por iteración. Con ellas, entrenar redes profundas con $\epsilon \approx 1$ deja de ser imposible.

Las cinco ideas en una página

Capítulo	Idea central	Qué se lleva a la práctica
1. Privacidad diferencial	Indistinguibilidad entre bases vecinas, en el peor caso	ϵ es la perilla única entre privacidad y utilidad
2. Propiedades	Post-procesamiento gratis; composición secuencial y paralela	El presupuesto es un recurso que se asigna y se agota
3. Sensibilidad	Cuánto mueve el resultado una persona	Sin cota no hay privacidad: recortar es obligatorio
4. DP aproximada	δ compra Gauss, L_2 y mejor composición	Indispensable en alta dimensión
5. Aprendizaje	El gradiente es una consulta de suma	Recortar por persona, ruido, contabilidad

Lo que no cubrimos

- **Mecanismo exponencial** — para salidas no numéricas (elegir una categoría)
- **Privacidad diferencial local** — sin curador de confianza; es el modelo de RAPPOR en Chrome y de la telemetría de Apple
- **Datos sintéticos privados** — publicar una base entera, no una estadística
- **Aprendizaje profundo** — DP-SGD sobre redes, *moments accountant* en detalle
- **Unidad de privacidad** — proteger a una persona con múltiples registros (*user-level* frente a *record-level*) cambia toda la contabilidad

Referencias

Near, J. P. y Abueh, C. (2021). *Programming Differential Privacy*. Capítulos 3 (Differential Privacy), 4 (Properties of Differential Privacy), 5 (Sensitivity), 6 (Approximate Differential Privacy) y 11 (Machine Learning). <https://programming-dp.com/>

Dwork, C. y Roth, A. (2014). *The Algorithmic Foundations of Differential Privacy*. Foundations and Trends in Theoretical Computer Science, 9(3-4).

Abadi, M. et al. (2016). Deep Learning with Differential Privacy. *CCS 2016*.