

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Inteligencia artificial y ética

Entrenar, evaluar, predecir el siguiente token, y comprobar antes de creer

Gerson Garrido

Semana 15

Hoy

1. Programar reglas o aprender de los datos
2. Entrenar un modelo y evaluarlo con datos que no vio. Overfitting
3. Una red neuronal: pesos que se ajustan
4. Un LLM: predecir el siguiente token. Por qué responde con seguridad aunque esté equivocado
5. Por qué los fundamentos siguen importando
6. Ética: sesgo, privacidad y responsabilidad
7. Laboratorio: comprobar antes de creer. Repaso y qué sigue

Lectura

Apuntes de la semana 15. Código en `code/week-15`. Los programas de hoy no son para escribirlos: son para seguir las ideas a mano.

SECCIÓN 01

Reglas o datos

01

Separar el spam: dos formas

una persona escribe la regla

```
algoritmo es_spam_con_reglas(palabras)
    si palabras ≥ 2 entonces
        devolver 1
    fin si
    devolver 0
fin algoritmo
```

El 2 lo escogió alguien, por intuición.

Machine learning

Dar datos de ejemplo a un programa, para que busque en ellos la regla.

- **Modelo** (*(model)*): lo que decide. Una función con números ajustables.
- **Entrenamiento** (*(training)*): ajustar esos números con ejemplos.
- **Aprendizaje supervisado** (*(supervised learning)*): cada ejemplo trae su **etiqueta**.

Ninguna forma gana siempre

	Programar reglas	Machine learning
Quién fija la regla	Una persona	Un programa, a partir de ejemplos
Qué hace falta	Entender el problema hasta poder escribirlo	Muchos ejemplos buenos, con su respuesta
Cómo se corrige	Cambiando el código	Cambiando datos o modelo, y entrenando otra vez
Se puede explicar	Se lee la regla	Depende del modelo; a veces es muy difícil
Cuándo conviene	La regla se puede escribir, como el IGV	Es difícil escribirla, como reconocer una voz

Idea clave

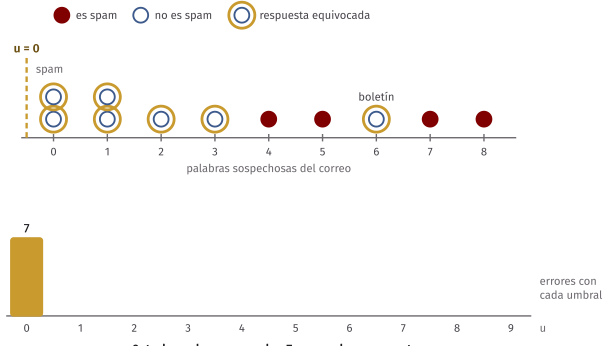
En los dos casos, el resultado es un programa que puede estar mal.

SECCIÓN 02

Entrenar y evaluar

02

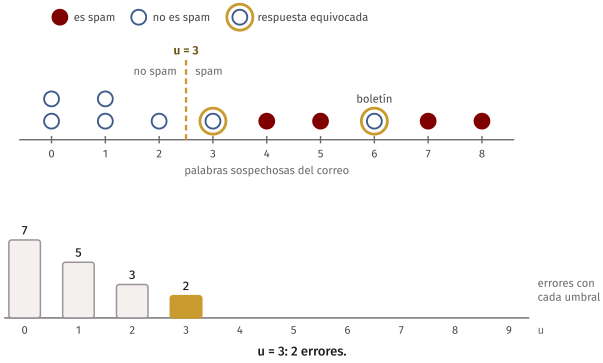
Entrenar un umbral



u = 0: todos salen spam, y los 7 que no lo son cuentan como error.

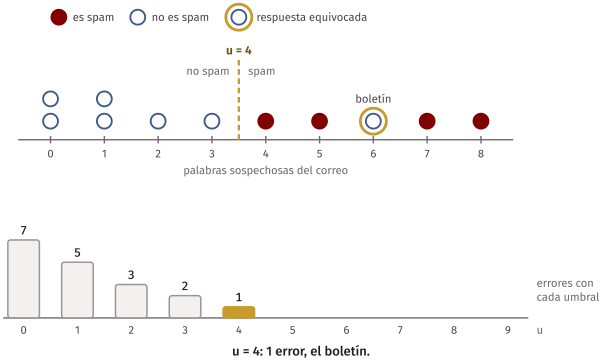
- Spam si tiene *u* palabras sospechosas o más. Son 11 correos inventados, con un boletín que alguien sí pidió.

Entrenar un umbral



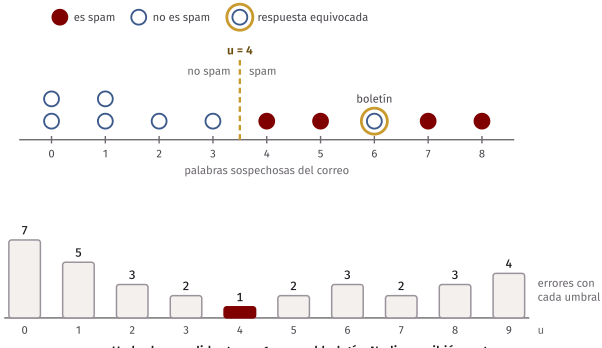
- Spam si tiene *u* palabras sospechosas o más. Son 11 correos inventados, con un boletín que alguien sí pidió.

Entrenar un umbral



- Spam si tiene u palabras sospechosas o más. Son 11 correos inventados, con un boletín que alguien sí pidió.
- Con $u = 4$, todos aciertan salvo el boletín: **1 error**. Con $u = 3$ o $u = 5$ hay 2.

Entrenar un umbral



Umbral aprendido: 4, con 1 error, el boletín. Nadie escribió ese 4.

- Spam si tiene u palabras sospechosas o más. Son 11 correos inventados, con un boletín que alguien sí pidió.
- Con $u = 4$, todos aciertan salvo el boletín: **1 error**. Con $u = 3$ o $u = 5$ hay 2.
- Entrenar es probar cada umbral, contar errores y quedarse con el mejor.

El algoritmo, y lo que sale

pseudocódigo

```
algoritmo entrenar_umbral(correos, n)
    mejor_u ← 0
    menos_errores ← contar_errores(correos, n, 0)
    para u desde 1 hasta 9 hacer
        errores ← contar_errores(correos, n, u)
        si errores < menos_errores entonces
            menos_errores ← errores
            mejor_u ← u
        fin si
    fin para
    devolver mejor_u
fin algoritmo
```

terminal

```
$ build/spam_models
Errores en entrenamiento según el umbral
u = 2: 3
u = 3: 2
u = 4: 1
u = 5: 2
u = 6: 3
Umbral aprendido: 4
```

Nadie escribió el 4: salió de los datos. Aprender es esto: ajustar números para cometer menos errores en los ejemplos.

El memorizador copia al más parecido

pseudocódigo

```
algoritmo memorizador(entrenamiento, n, palabras)
    mejor ← 0
    menor_distancia ← -1
    para i desde 0 hasta n - 1 hacer
        distancia ← valor absoluto de (entrenamiento[i].palabras -
     palabras)
        si menor_distancia = -1 o distancia < menor_distancia entonces
            menor_distancia ← distancia
            mejor ← i
        fin si
    fin para
    devolver entrenamiento[mejor].etiqueta
fin algoritmo
```

Guarda todos los correos. Para uno nuevo, busca el más parecido y copia su etiqueta.

Pregunta: ¿cuántos errores comete con sus propios datos de entrenamiento?

El memorizador copia al más parecido

pseudocódigo

```
algoritmo memorizador(entrenamiento, n, palabras)
    mejor ← 0
    menor_distancia ← -1
    para i desde 0 hasta n - 1 hacer
        distancia ← valor absoluto de (entrenamiento[i].palabras -
     palabras)
        si menor_distancia = -1 o distancia < menor_distancia entonces
            menor_distancia ← distancia
            mejor ← i
        fin si
    fin para
    devolver entrenamiento[mejor].etiqueta
fin algoritmo
```

Guarda todos los correos. Para uno nuevo, busca el más parecido y copia su etiqueta.

Pregunta: ¿cuántos errores comete con sus propios datos de entrenamiento?

Ninguno. Cada correo se encuentra a sí mismo. ¿Es entonces el mejor modelo?

Evaluar con datos que no vio

- Antes de entrenar, apartamos 8 correos: los **datos de prueba**.

terminal

```
$ build/spam_models
Modelo                  entrenamiento     prueba
Regla a mano (u = 2)             3/11        2/8
Umbral aprendido                 1/11        1/8
Memorizador                      0/11        3/8
```

Evaluar con datos que no vio

- Antes de entrenar, apartamos 8 correos: los **datos de prueba**.

terminal

```
$ build/spam_models
Modelo                  entrenamiento     prueba
Regla a mano (u = 2)             3/11        2/8
Umbral aprendido                 1/11        1/8
Memorizador                      0/11        3/8
```

- En prueba hay dos correos de 6 palabras que sí son spam. El memorizador recuerda al boletín, y falla los dos.

Evaluar con datos que no vio

- Antes de entrenar, apartamos 8 correos: los **datos de prueba**.

terminal

```
$ build/spam_models
Modelo                  entrenamiento     prueba
Regla a mano (u = 2)             3/11        2/8
Umbral aprendido                 1/11        1/8
Memorizador                      0/11        3/8
```

- En prueba hay dos correos de 6 palabras que sí son spam. El memorizador recuerda al boletín, y falla los dos.
- El mejor en entrenamiento resultó el peor en prueba.

Evaluar con datos que no vio

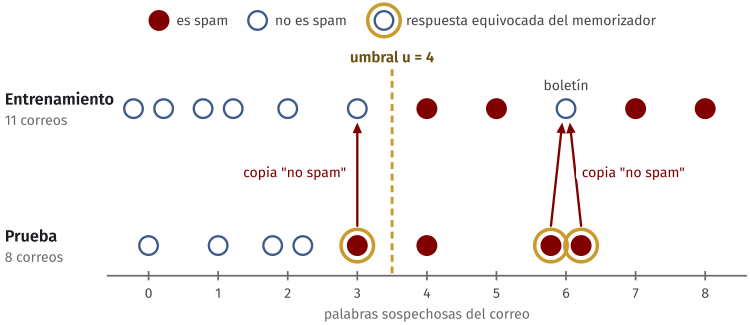
- Antes de entrenar, apartamos 8 correos: los **datos de prueba**.

terminal

```
$ build/spam_models
Modelo                  entrenamiento     prueba
Regla a mano (u = 2)             3/11        2/8
Umbral aprendido                 1/11        1/8
Memorizador                      0/11        3/8
```

- En prueba hay dos correos de 6 palabras que sí son spam. El memorizador recuerda al boletín, y falla los dos.
- El mejor en entrenamiento resultó el peor en prueba.
- Con 8 correos es poca evidencia: son pocos para seguirlos a mano. Una evaluación seria usa muchos más.

Por qué falla el memorizador



Memorizador
0 de 11 en entrenamiento, 3 de 8 en prueba

Umbral u = 4
1 de 11 en entrenamiento, 1 de 8 en prueba

Overfitting

Overfitting o sobreajuste

El modelo se ajusta tanto a sus datos de entrenamiento, incluidas sus rarezas, que le va peor con datos nuevos.

Es memorizar las respuestas de la práctica: con las mismas preguntas se saca 20, y esa nota no dice nada sobre lo que pasará con preguntas nuevas.

- **Error común:** probar, cambiar el modelo, volver a probar, hasta que salga bien. Así los datos de prueba influyen en el modelo y ya no son datos que no vio.
- **No es una garantía:** si los correos del futuro son distintos, el modelo puede empeorar.

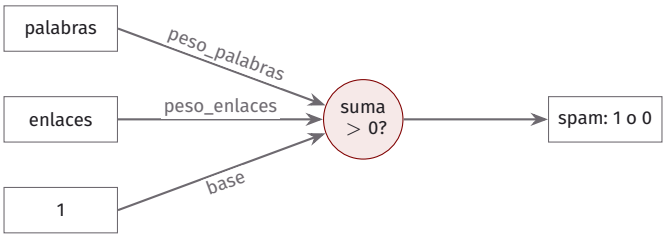
SECCIÓN 03

Redes neuronales

03



Una neurona artificial



- Multiplica cada entrada por su **peso** (*(weight)*), suma todo con una `base`, y responde 1 si el puntaje es mayor que 0.
- Un peso grande y positivo empuja hacia el spam. Uno negativo empuja en contra.
- Se inspiró de lejos en las neuronas del cerebro, pero no es un modelo de un cerebro: es aritmética.

Cada error mueve los pesos

una vuelta de entrenar_neurona

```
para i desde 0 hasta n - 1 hacer
    c ← correos[i]
    puntaje ← peso_palabras × c.palabras + peso_enlaces × c.enlaces + base
    cambio ← c.etiqueta - (puntaje > 0)
    si cambio ≠ 0 entonces
        errores ← errores + 1
    fin si
    peso_palabras ← peso_palabras + cambio × c.palabras
    peso_enlaces ← peso_enlaces + cambio × c.enlaces
    base ← base + cambio
fin para
```

- Pesos y `base` empiezan en 0. Se repite hasta una vuelta sin errores, que llega solo si existen pesos que aciertan todos los ejemplos. `neuron.c` agrega un límite de 10 vueltas.
- Acierta: `cambio` vale 0. Dijo 0 y era 1: los pesos suben. Dijo 1 y era 0: los pesos bajan.

La neurona aprende, paso a paso

terminal

```
$ build/neuron
vuelta correo  puntaje resp etiq  w_pal w_enl  base
     1 (3,2)        0    0    1      3     2     1
     1 (1,0)        4    1    0      2     2     0
     1 (2,2)        8    1    1      2     2     0
     1 (0,1)        2    1    0      2     1    -1
     2 (3,2)        7    1    1      2     1    -1
     2 (1,0)        1    1    0      1     1    -2
     2 (2,2)        2    1    1      1     1    -2
     2 (0,1)       -1    0    0      1     1    -2
Vueltas: 3. Errores en la última: 0
```

Cada fila: el puntaje y la respuesta con los pesos de **antes**, y los pesos **después**.

La neurona aprende, paso a paso

terminal

```
$ build/neuron
vuelta correo  puntaje resp etiq  w_pal w_enl  base
     1 (3,2)        0    0    1      3     2     1
     1 (1,0)        4    1    0      2     2     0
     1 (2,2)        8    1    1      2     2     0
     1 (0,1)        2    1    0      2     1    -1
     2 (3,2)        7    1    1      2     1    -1
     2 (1,0)        1    1    0      1     1    -2
     2 (2,2)        2    1    1      1     1    -2
     2 (0,1)       -1    0    0      1     1    -2
Vueltas: 3. Errores en la última: 0
```

Fila 1: $0 \cdot 3 + 0 \cdot 2 + 0 = 0$, responde 0 y era 1. `cambio` vale $+1$: los pesos quedan en 3, 2 y 1.

La neurona aprende, paso a paso

terminal

```
$ build/neuron
vuelta correo  puntaje resp etiq  w_pal w_enl  base
     1 (3,2)        0    0    1      3     2     1
     1 (1,0)        4    1    0      2     2     0
     1 (2,2)        8    1    1      2     2     0
     1 (0,1)        2    1    0      2     1    -1
     2 (3,2)        7    1    1      2     1    -1
     2 (1,0)        1    1    0      1     1    -2
     2 (2,2)        2    1    1      1     1    -2
     2 (0,1)       -1    0    0      1     1    -2
Vueltas: 3. Errores en la última: 0
```

Fila 2: $3 \cdot 1 + 2 \cdot 0 + 1 = 4$, responde 1 y era 0. `cambio` vale -1: quedan en 2, 2 y 0.

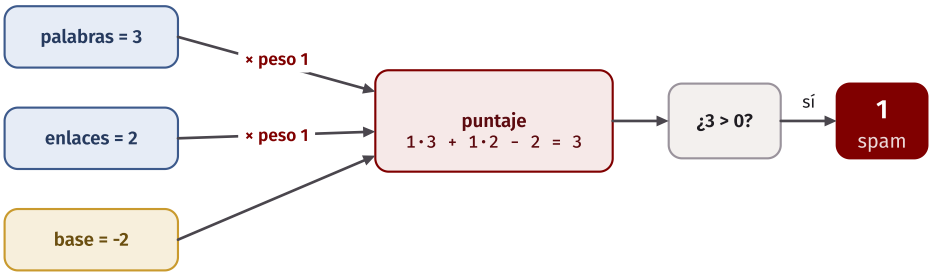
La neurona aprende, paso a paso

terminal

```
$ build/neuron
vuelta correo  puntaje resp etiq  w_pal w_enl  base
     1 (3,2)        0    0    1      3     2     1
     1 (1,0)        4    1    0      2     2     0
     1 (2,2)        8    1    1      2     2     0
     1 (0,1)        2    1    0      2     1    -1
     2 (3,2)        7    1    1      2     1    -1
     2 (1,0)        1    1    0      1     1    -2
     2 (2,2)        2    1    1      1     1    -2
     2 (0,1)       -1    0    0      1     1    -2
Vueltas: 3. Errores en la última: 0
```

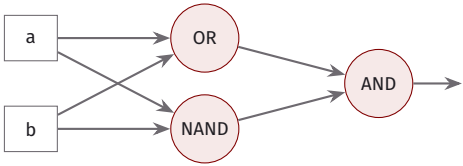
La vuelta 3 no tiene errores, y aquí no la mostramos. Al final: palabras y enlaces valen igual, y hacen falta más de 2 entre los dos. Nadie escribió esa regla.

Con los pesos finales: el correo (3, 2)



Entrenar es ajustar los dos pesos y la base. Las entradas vienen del correo.

Por qué capas: XOR



Una **red neuronal** (*(neural network)*): las salidas de unas neuronas son entradas de otras. Aquí, los pesos están escritos a mano.

```
terminal

$ build/network
a b  or nand  salida
0 0   0    1       0
0 1   1    1       1
1 0   1    1       1
1 1   1    0       0
```

Una sola neurona no puede: de los pesos enteros de −3 a 3, 0 de 343 combinaciones dan XOR.

Las redes de hoy son esta idea a una escala enorme. Lo que aprenden depende de sus datos.

SECCIÓN 04

Qué hace un LLM

04

Predecir el siguiente token

LLM (*(large language model)*)

Un modelo entrenado con muchísimo texto para esta tarea: **dado un texto, predecir qué viene después**.

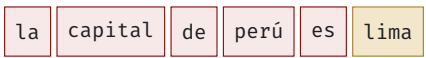
la	capital	de	perú	es

- **Token**: fragmento de texto. Una palabra corta, o un pedazo de una larga.

Predecir el siguiente token

LLM (*(large language model)*)

Un modelo entrenado con muchísimo texto para esta tarea: **dado un texto, predecir qué viene después**.

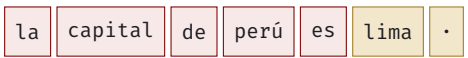


- **Token**: fragmento de texto. Una palabra corta, o un pedazo de una larga.
- Predice un token, lo agrega al texto, y predice el siguiente otra vez.

Predecir el siguiente token

LLM (*(large language model)*)

Un modelo entrenado con muchísimo texto para esta tarea: **dado un texto, predecir qué viene después**.



- **Token**: fragmento de texto. Una palabra corta, o un pedazo de una larga.
- Predice un token, lo agrega al texto, y predice el siguiente otra vez.
- Vamos a construir uno de juguete: tokens que son palabras, y solo se mira la última.

Entrenar es contar

terminal

```
$ cat corpus.txt
la capital de perú es lima .
la capital de chile es santiago .
la capital de colombia es bogotá .
la ciudad más poblada de perú es lima .
garcía publicó un artículo sobre redes en 2019 .
flores publicó un artículo sobre redes en 2019 .
quispe publicó un artículo sobre memoria en 2021 .
```

- Es todo su texto de entrenamiento. Las personas y los artículos son inventados.
- El modelo solo cuenta qué palabra viene después de cada una.

Entrenar es contar

terminal

```
$ cat corpus.txt
la capital de perú es lima .
la capital de chile es santiago .
la capital de colombia es bogotá .
la ciudad más poblada de perú es lima .
garcía publicó un artículo sobre redes en 2019 .
flores publicó un artículo sobre redes en 2019 .
quispe publicó un artículo sobre memoria en 2021 .
```

- Es todo su texto de entrenamiento. Las personas y los artículos son inventados.
- El modelo solo cuenta qué palabra viene después de cada una.
- Después de `es`: `lima` 2 veces, `santiago` 1 y `bogotá` 1.

Entrenar es contar

terminal

```
$ cat corpus.txt
la capital de perú es lima .
la capital de chile es santiago .
la capital de colombia es bogotá .
la ciudad más poblada de perú es lima .
garcía publicó un artículo sobre redes en 2019 .
flores publicó un artículo sobre redes en 2019 .
quispe publicó un artículo sobre memoria en 2021 .
```

- Es todo su texto de entrenamiento. Las personas y los artículos son inventados.
- El modelo solo cuenta qué palabra viene después de cada una.
- Después de `es`: `lima` 2 veces, `santiago` 1 y `bogotá` 1.
- Después de `de`: `perú` 2 veces, `chile` 1 y `colombia` 1.

Elige la más frecuente

pseudocódigo

```
algoritmo siguiente(pares, n, palabra)
    mejor ← -1
    para i desde 0 hasta n - 1 hacer
        si pares[i].primera = palabra entonces
            si mejor = -1 o pares[i].cuenta > pares[mejor].cuenta
     entonces
                mejor ← i
            fin si
        fin si
    fin para
    devolver mejor
fin algoritmo
```

En un empate, se queda con la que apareció primero.

Para continuar un texto, repite: elige la siguiente, la escribe, y sigue desde ella hasta el punto.

En C es `most_frequent`, en `next_token.c`, con `strcmp` para comparar.

La misma seguridad, verdadero o falso

terminal

```
$ printf 'la capital de perú es\nla capital de chile es\nla capital de ecuador es\n' | build/next_token
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de perú es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de chile es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de ecuador es lima .
```

La misma seguridad, verdadero o falso

terminal

```
$ printf 'la capital de perú es\nla capital de chile es\nla capital de ecuador es\n' | build/next_token
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de perú es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de chile es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de ecuador es lima .
```

- La primera es correcta. La segunda es falsa, aunque el dato sí está en su texto. La tercera es sobre un país que nunca vio.

La misma seguridad, verdadero o falso

terminal

```
$ printf 'la capital de perú es\nla capital de chile es\nla capital de ecuador es\n' | build/next_token
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de perú es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de chile es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de ecuador es lima .
```

- La primera es correcta. La segunda es falsa, aunque el dato sí está en su texto. La tercera es sobre un país que nunca vio.
- **Las tres salen igual**, por el mismo mecanismo y con las mismas cuentas.

La misma seguridad, verdadero o falso

terminal

```
$ printf 'la capital de perú es\nla capital de chile es\nla capital de ecuador es\n' | build/next_token
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de perú es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de chile es lima .
tras "es": lima 2/4 santiago 1/4 bogotá 1/4
la capital de ecuador es lima .
```

- La primera es correcta. La segunda es falsa, aunque el dato sí está en su texto. La tercera es sobre un país que nunca vio.
- **Las tres salen igual**, por el mismo mecanismo y con las mismas cuentas.
- El 2/4 no es la probabilidad de que la frase sea cierta. Dice cuán seguido venía `lima` después de `es`. El programa no tiene ninguna variable para “esto es verdad”.

Un LLM real es muchísimo mejor

Modelo de juguete	LLM real
Unas pocas líneas de texto	Una cantidad enorme de texto
Tokens que son palabras	Tokens que pueden ser pedazos de palabras
Una tabla de cuentas	Una red neuronal, con pesos ajustados para predecir el siguiente token
Solo mira la última palabra	Tiene en cuenta un contexto largo: distingue Chile de Perú
Siempre elige la más frecuente	Suele sortear entre las probables: una misma pregunta puede dar respuestas distintas

- Por eso responde bien muchas preguntas.
- Algunos productos agregan pasos, como buscar en documentos o correr código.
- El núcleo sigue siendo predecir el texto que viene.

Una cita que nadie escribió

terminal

```
$ printf 'quispe publicó\n' | build/next_token
tras "publicó": un 3/3
quispe publicó un artículo sobre redes en 2019 .
```

Una cita que nadie escribió

terminal

```
$ printf 'quispe publicó\n' | build/next_token
tras "publicó": un 3/3
quispe publicó un artículo sobre redes en 2019 .
```

- Esa frase **no está** en `corpus.txt`. Quispe escribió sobre memoria en 2021.

Una cita que nadie escribió

terminal

```
$ printf 'quispe publicó\n' | build/next_token
tras "publicó": un 3/3
quispe publicó un artículo sobre redes en 2019 .
```

- Esa frase **no está** en `corpus.txt`. Quispe escribió sobre memoria en 2021.
- Cada paso era el más probable. Tiene autor, tema y año, y se lee como una cita de verdad.

Una cita que nadie escribió

terminal

```
$ printf 'quispe publicó\n' | build/next_token
tras "publicó": un 3/3
quispe publicó un artículo sobre redes en 2019 .
```

- Esa frase **no está** en `corpus.txt`. Quispe escribió sobre memoria en 2021.
- Cada paso era el más probable. Tiene autor, tema y año, y se lee como una cita de verdad.
- Ningún paso del algoritmo revisa si el artículo existe.

Una cita que nadie escribió

terminal

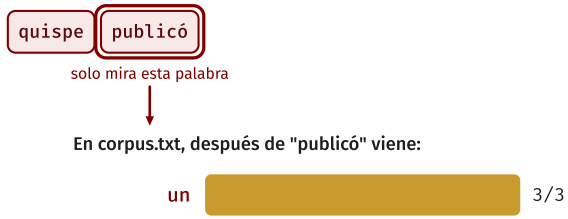
```
$ printf 'quispe publicó\n' | build/next_token
tras "publicó": un 3/3
quispe publicó un artículo sobre redes en 2019 .
```

- Esa frase **no está** en `corpus.txt`. Quispe escribió sobre memoria en 2021.
- Cada paso era el más probable. Tiene autor, tema y año, y se lee como una cita de verdad.
- Ningún paso del algoritmo revisa si el artículo existe.

Alucinaciones

Respuestas falsas dichas como ciertas. Es una metáfora: describe el resultado, sin afirmar nada sobre intenciones. Un texto probable no siempre es verdadero.

Cada paso elige lo más frecuente



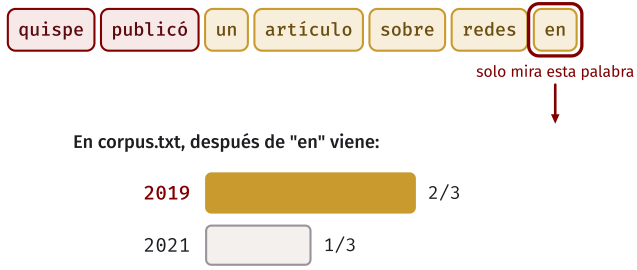
Elige la más frecuente, la agrega y repite.

Cada paso elige lo más frecuente



Elige la más frecuente, la agrega y repite.

Cada paso elige lo más frecuente



Elige la más frecuente, la agrega y repite.

Cada paso elige lo más frecuente

quispe | publicó | un | artículo | sobre | redes | en | 2019 | .

lo que armó el modelo, cada paso con la palabra más frecuente

lo que dice corpus.txt sobre Quispe

quispe | publicó | un | artículo | sobre | memoria | en | 2021 | .

Esa cita no está en el texto, y se lee igual que una verdadera.

Ningún paso del algoritmo revisa si el artículo existe.

El tono no dice si es correcta

- **Una razón del tono:** muchos textos afirman sin dudar. Un modelo que predice texto escribe así.
- **“No lo sé”** también es texto predicho: no garantiza que el modelo haya medido lo que sabe.
- **¿Entiende?** Es una pregunta abierta, y en parte depende de qué se llame entender. No hace falta resolverla: la respuesta se comprueba.

Sus límites

Se equivoca con seguridad: datos, citas, funciones de biblioteca, cuentas. Depende de sus datos: lo que falta, lo que está mal, lo que pasó después. Hereda sesgos. Una respuesta correcta hoy no asegura la de mañana.

Un caso real: *Mata contra Avianca*

- En un tribunal federal de Nueva York, dos abogados presentaron un escrito que citaba sentencias que no existían, generadas por un chatbot.
- Cuando la corte las cuestionó, siguieron defendiéndolas por un tiempo.
- El 22 de junio de 2023, el juez les impuso a ellos y a su estudio una multa de 5 000 dólares.
- Escribió que usar una herramienta de IA confiable no es impropio en sí mismo. El problema: los abogados siguen siendo responsables de que lo que presentan sea exacto.

Idea clave

Una respuesta segura, bien escrita y con citas todavía puede ser falsa. Comprobarla es trabajo de quien la usa. Aquí bastaba con buscar las sentencias.

SECCIÓN 05

Los fundamentos

05

Para juzgar una respuesta hay que saber el tema

errors/last_char.c

```c
char last_char(const char s[]) {
    int n = strlen(s);
    return s[n - 1];
}
```

Se ve bien, y con `"hola"` devuelve `a`.

Saben	Les permite
Seguir a mano	Saber qué hace, sin creer la explicación
`assert` y bordes	Encontrar el caso que falla
`gcc`, sanitizers	Notar lo que no se ve a simple vista
Overflow	Dudar de una cuenta
Contar pasos	Notar que no sirve con datos grandes

Para juzgar una respuesta hay que saber el tema

errors/last_char.c

```c
char last_char(const char s[]) {
    int n = strlen(s);
    return s[n - 1];
}
```

Se ve bien, y con `"hola"` devuelve `a`.

Con la cadena vacía, `n` vale 0, y `s[-1]` está fuera del arreglo. Es comportamiento indefinido.

Saben	Les permite
Seguir a mano	Saber qué hace, sin creer la explicación
`assert` y bordes	Encontrar el caso que falla
`gcc`, sanitizers	Notar lo que no se ve a simple vista
Overflow	Dudar de una cuenta
Contar pasos	Notar que no sirve con datos grandes

SECCIÓN 06

Ética

06

Sesgo

Sesgo (*(bias)*)

Un error sistemático que perjudica a ciertos casos o grupos. No hace falta mala intención.

- Un modelo aprende de los datos. Si reflejan decisiones injustas, o un grupo casi no aparece, lo repite.
- **Un caso real.** El 10 de octubre de 2018, Reuters informó que Amazon dejó de lado un sistema experimental para ordenar currículos. Se entrenó con currículos de 10 años, la mayoría de hombres, y bajaba el puntaje a los que decían “women’s” (de mujeres).
- Según ese reportaje, los reclutadores miraron sus recomendaciones sin depender solo de ellas, y la empresa dijo que nunca lo usaron para evaluar candidatos.

Sesgo

Sesgo (*(bias)*)

Un error sistemático que perjudica a ciertos casos o grupos. No hace falta mala intención.

- Un modelo aprende de los datos. Si reflejan decisiones injustas, o un grupo casi no aparece, lo repite.
- **Un caso real.** El 10 de octubre de 2018, Reuters informó que Amazon dejó de lado un sistema experimental para ordenar currículos. Se entrenó con currículos de 10 años, la mayoría de hombres, y bajaba el puntaje a los que decían “women’s” (de mujeres).
- Según ese reportaje, los reclutadores miraron sus recomendaciones sin depender solo de ellas, y la empresa dijo que nunca lo usaron para evaluar candidatos.
- Nadie programó esa regla: salió de los datos. Y quitar una columna no basta, porque el distrito o el trabajo pueden decir mucho de lo que se quiso quitar.

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?

Un sistema decide a quién darle un crédito. ¿Qué preguntarían?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.
- **Explicación.** ¿Se puede saber por qué rechazó a alguien?

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.
- **Explicación.** ¿Se puede saber por qué rechazó a alguien?
- **Reclamo.** ¿Puede una persona pedir que un humano revise la decisión?

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.
- **Explicación.** ¿Se puede saber por qué rechazó a alguien?
- **Reclamo.** ¿Puede una persona pedir que un humano revise la decisión?
- **Responsabilidad.** Si hace daño, ¿quién responde?

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.
- **Explicación.** ¿Se puede saber por qué rechazó a alguien?
- **Reclamo.** ¿Puede una persona pedir que un humano revise la decisión?
- **Responsabilidad.** Si hace daño, ¿quién responde?
- **Seguimiento.** ¿Alguien sigue midiendo los resultados cuando ya está en uso?

Un sistema decide a quién darle un crédito. ¿Qué preguntaría?

- **Datos.** ¿Con cuáles se entrenó? ¿De qué años y de quiénes? ¿Quiénes faltan?
- **Evaluación.** ¿Con datos que no vio? ¿Cuál es el error en cada grupo, y no solo en promedio?
- **Tipos de error.** Rechazar a quien sí iba a pagar no es lo mismo que aprobar a quien no iba a pagar.
- **Explicación.** ¿Se puede saber por qué rechazó a alguien?
- **Reclamo.** ¿Puede una persona pedir que un humano revise la decisión?
- **Responsabilidad.** Si hace daño, ¿quién responde?
- **Seguimiento.** ¿Alguien sigue midiendo los resultados cuando ya está en uso?

> **Idea clave**
>
> Ninguna tiene una respuesta automática. No basta con “el modelo tiene buena nota”.

Privacidad, autoría y responsabilidad

Privacidad

- Al entrenar: ¿de dónde salieron los datos?
- Al usarlo: lo que escriben viaja a otra organización. HTTPS protege el camino, no lo que el destino haga después.
- Un **prompt** es el texto que se le da al modelo. No pongan lo que no pondrían en un correo a un desconocido.

Autoría y responsabilidad

- Quien usa la herramienta responde por el resultado.
- Sean honestos sobre lo que usaron.
- Si no lo pueden explicar, no lo entreguen.
- Avisen cuando algo no está bien probado.

La regla de este curso

La IA generativa está **prohibida**, salvo en la actividad guiada de esta semana.

SECCIÓN 07

Laboratorio

07

Parte 1: tres funciones sospechosas

`lab/suspect.c`: `is_prime`, `max_of_array` y `copy_text`. Se escribieron para la actividad, y cada una tiene un bug a propósito. Compilan sin advertencias y pasan la prueba que traen.

1. Agreguen `assert` en `lab/tests.c` hasta que fallen.
2. Solo entonces, corrijan.
3. Corran `make test-lab`.

`make test-lab` prueba sus pruebas

Con el original deben **fallar**. Con su corrección deben **pasar**. Una prueba que siempre pasa, o que siempre falla, no sirve.

`copy_text`

El sanitizer solo avisa si se sale del arreglo. Usen un destino del tamaño exacto y un texto que no cabe.

La prueba del borde

terminal

```
$ build/passing_border
passing_border: errors/passing_border.c:21: main: Assertion ‘count_passing(border, 3) == 2’ failed.
```

- Debía contar notas de 11 o más, y dice `grades[i] > 11`. La prueba de la transcripción, un ejemplo escrito para esta actividad, pasa.
- `gcc` no dice nada: es C válido. El sanitizer tampoco: no hay error de memoria.
- Solo lo encontró una prueba que **sabía qué esperar**. Y hay que saber que el 11 es el borde.

Error común

Pedir también las pruebas al asistente y creer que pasarlas basta. Pueden tener el mismo error de razonamiento. Sus casos los piensan ustedes.

Parte 2: la actividad guiada

Con el asistente que permita la universidad:

1. **Antes**, escriban qué debe hacer y sus casos borde.
2. Pidan una función en C que ya escribieron, sin datos personales.
3. Léanla. ¿Qué línea no pueden explicar?
4. Compílenla con `-Wall -Wextra` y los sanitizers.
5. Pruébenla con **sus** pruebas `assert`.
6. Anoten en `lab/activity_sheet.md` lo que hizo bien, el error y cómo lo detectaron.

Sin asistente o sin internet

`lab/transcripts.md`: **ejemplos escritos para esta actividad, no salidas reales de un asistente**. Sin computadora, se siguen a mano.

Si no encuentran un error

También es un resultado: la función funciona en los casos que revisaron.

SECCIÓN 08

Repaso y qué sigue

08

Repaso para el examen final

Todo el curso, con énfasis en las semanas 9 a 15.

Semanas	Sin mirar, pueden…
1 a 7	Seguir a mano ciclos y funciones. Convertir entre bases. Explicar el overflow.
9 y 10	Dibujar un arreglo y una cadena. Explicar por qué una función puede cambiar un arreglo.
11 y 12	Usar `string.h`, recorrer una matriz, definir un `struct`, leer un archivo.
13	Seguir la búsqueda binaria y contar comparaciones. Comparar N, log N y N^2.
14	Explicar DNS y HTTP, y qué protege HTTPS.
15	Responder las preguntas de la guía, con los ejemplos a mano.

El método

Predecir **antes** de correr, y después comprobar.

Qué sigue en la carrera

Lo que vieron	Lo que se estudia después
Funciones y pruebas	Programación, a fondo, con recursión y abstracción
Arreglos, búsqueda, eficiencia	Estructuras de datos y algoritmos
Bits, memoria, `gdb`	Arquitectura de computadoras y sistemas operativos
Lógica y compuertas	Matemática discreta y teoría de la computación
Archivos y `struct`	Bases de datos
Internet y HTTP	Redes de computadoras
Neurona, entrenar y evaluar	Inteligencia artificial y machine learning
Pruebas, responsabilidad	Ingeniería de software

Los nombres y el orden los fija el plan de estudios. Las herramientas van a cambiar muchas veces. Entender un problema, partirlo, resolverlo, dudar y comprobar, no.

Para la próxima clase

- Hoy, en el laboratorio: `lab/suspect.c`, `lab/tests.c` y la actividad guiada.
- La próxima semana es el examen final.
- La IA generativa sigue prohibida fuera de esta actividad.

Preguntas

¿Por qué no se evalúa un modelo con los datos con los que se entrenó?
¿Por qué un LLM puede citar un artículo que no existe?
Una función pasa la prueba que le pidieron. ¿Qué más revisarían antes de usarla?