

Tokenización con BPE

Lucca Frachelle

septiembre 2026

Por qué existe el tokenizador

Un modelo de lenguaje no procesa texto. La primera capa de un Transformer es una **tabla de embeddings**: una matriz de `vocab_size` filas por `n_embd` columnas, donde cada fila es un vector aprendido. Lo que el modelo recibe es una secuencia de **enteros** que indexan esas filas.

Entonces, entre el texto que escribe una persona y el modelo hace falta una pieza que:

1. convierta un **str** arbitrario en una secuencia de enteros de un **conjunto finito y fijo** (el vocabulario);
2. pueda hacer el camino inverso sin pérdida, para que el modelo pueda *responder* en texto;
3. use **pocos** enteros por texto, porque la ventana de contexto es finita y el costo computacional de la atención crece con el cuadrado de la longitud de la secuencia.

Esa pieza es el tokenizador. Es un módulo **completamente separado** del modelo: se entrena aparte, con su propio corpus y su propio algoritmo (nada de descenso por gradiente), y una vez congelado condiciona todo lo que el modelo puede o no puede hacer.

Antes de construir uno, veamos uno real trabajando. `tiktoken` es la biblioteca de OpenAI con los tokenizadores de sus modelos:

```
import tiktoken

enc = tiktoken.get_encoding("o200k_base") # el tokenizador de GPT-4o

frase = "Los tokenizadores son la parte más subestimada de un LLM."
ids = enc.encode(frase)

print(f"{len(frase)} caracteres -> {len(ids)} tokens")
print(ids)
print([enc.decode([i]) for i in ids])

57 caracteres -> 15 tokens
[16593, 6602, 170058, 2391, 557, 7441, 3932, 1543, 125938, 1194, 334, 537, 451, 19641, 13]
['Los', ' token', 'izadores', ' son', ' la', ' parte', ' más', ' sub', 'estim', 'ada', ' de',

Fijate en la última línea: los tokens no son palabras ni letras. Son fragmentos, y el espacio viaja pegado al inicio de la palabra que sigue. Eso tiene una consecuencia inmediata:

for s in ["hola", " hola", "Hola", " Hola", "HOLA"]:
    print(f"{s!r:>8} -> {enc.encode(s)}")

'hola' -> [122845]
' hola' -> [171847]
'Hola' -> [49864]
' Hola' -> [157464]
'HOLA' -> [39, 87254]
```

Cinco formas de escribir la misma palabra, cinco secuencias de ids distintas. Para el modelo son entradas diferentes, y tiene que aprender por separado que significan casi lo mismo.

Y comparemos idiomas, con la misma oración:

```
oraciones = [
    ("inglés", "The tokenizer is the most underestimated part of a language model."),
    ("español", "El tokenizador es la parte más subestimada de un modelo de lenguaje."),
]
for idioma, texto in oraciones:
    n = len(enc.encode(texto))
    print(f"{idioma:>8}: {n:>3} tokens para {len(texto):>3} caracteres "
          f"({len(texto)/n:.2f} caracteres por token)")

    inglés: 12 tokens para 66 caracteres (5.50 caracteres por token)
    español: 16 tokens para 68 caracteres (4.25 caracteres por token)
```

El español entra menos texto en la misma cantidad de tokens. No es un accidente: es consecuencia directa del corpus con el que se entrenó el tokenizador, y lo vamos a ver aparecer una y otra vez.

Todas estas conductas de los LLMs, que parecen fallas del modelo, son en realidad consecuencias del tokenizador. Volvemos a la lista en la [sección 13](#), cuando tengamos las herramientas para explicarlas:

- ¿Por qué a un LLM le cuesta deletrear palabras o contar letras?
- ¿Por qué falla al invertir un string, pero acierta si le pedís que lo haga letra por letra?
- ¿Por qué se equivoca en aritmética con números largos?
- ¿Por qué el español (y cualquier idioma que no sea inglés) “cuesta” más caro?
- ¿Por qué es peor escribiendo Python que JavaScript?
- ¿Por qué un espacio de más al final del prompt empeora la respuesta?
- ¿Por qué existen palabras como `SolidGoldMagikarp` que rompen al modelo?
- ¿Por qué conviene YAML antes que JSON para datos estructurados?

Primer intento: Unicode y code points

La idea más obvia es: usemos los caracteres como vocabulario. En Python, un `str` es una secuencia de **code points** Unicode, y `ord` da el número de cada uno:

```
saludo = "      (hola en coreano)"

print(saludo)
print(len(saludo), "code points")
print([ord(c) for c in saludo])

(hola en coreano)
25 code points
[50504, 45397, 54616, 49464, 50836, 32, 128075, 32, 40, 104, 111, 108, 97, 32, 101, 110, 32, 9
```

El camino inverso es `chr`:

```
print(chr(50504), chr(128075), chr(241))

ñ
```

Unicode es un **estándar** que asigna a cada carácter de (idealmente) todos los sistemas de escritura del mundo un número entero, llamado code point, que se escribe en hexadecimal con el prefijo `U+`. Cuántos hay:

```
import sys
import unicodedata
from collections import Counter
```

```
def mil(n):
    """Formatea con punto como separador de miles, como se escribe en español."""
    return f"{n:,}".replace(",", ".")

posibles = sys.maxunicode + 1
categorias = Counter(unicodedata.category(chr(cp)) for cp in range(posibles))

sin_asignar = categorias["Cn"]
privados = categorias["Co"]      # areas de uso privado, sin significado estandarizado
surrogates = categorias["Cs"]    # reservados para la mecánica de UTF-16
caracteres = posibles - sin_asignar - privados - surrogates

print(f"code points posibles:           {mil(posibles):>9}")
print(f" sin asignar:                   {mil(sin_asignar):>9}")
print(f" de uso privado:                 {mil(privados):>9}")
print(f" surrogates (UTF-16):            {mil(surrogates):>9}")
print(f" caracteres de verdad:            {mil(caracteres):>9}")

code points posibles:           1.114.112
 sin asignar:                   824.718
 de uso privado:                 137.468
 surrogates (UTF-16):            2.048
 caracteres de verdad:            149.878
```

Y acá se cae el primer intento, por dos razones:

- **El vocabulario sería enorme.** Casi 150.000 filas en la tabla de embeddings, y la enorme mayoría corresponden a caracteres que casi nunca aparecen en el corpus. Son parámetros que ocupan memoria y no se entrenan nunca.
- **El vocabulario sería inestable.** Unicode publica una versión nueva cada año, con caracteres y emoji nuevos. El vocabulario de un modelo tiene que ser un artefacto fijo, no algo que cambia con la versión del estándar.

Hay un tercer problema, más de fondo: usar caracteres como tokens deja las secuencias muy largas. Un texto de 5.000 caracteres son 5.000 tokens, y la ventana de contexto se llena con muy poco.

Segundo intento: UTF-8 y bytes

Un code point es un número abstracto. Para guardarlo en un archivo o mandarlo por la red hay que **codificarlo** en bytes, y para eso existen UTF-8, UTF-16 y UTF-32.

```
print(list(saludo.encode("utf-8")))
print(len(saludo.encode("utf-8")), "bytes para", len(saludo), "code points")
```

[236, 149, 136, 235, 133, 149, 237, 149, 152, 236, 132, 184, 236, 154, 148, 32, 240, 159, 145, 38 bytes para 25 code points

UTF-8 usa **1 a 4 bytes por code point**, según el tamaño del número, con un sistema de prefijos binarios:

Bytes UTF-8 (binario)	Bits del code point	Rango
0xxxxxxx	xxxxxxx	U+0000 – U+007F
110xxxxx 10yyyyyy	xxxxxyyyyy	U+0080 – U+07FF
1110xxxx 10yyyyyy 10zzzzzz	xxxxxyyyyyzzzzzz	U+0800 – U+FFFF

Bytes UTF-8 (binario)	Bits del code point	Rango
11110xxx 10yyyyyy 10zzzzzz 10wwwww	xxxyyyyyyzzzzzzwwwww	U+10000 – U+10FFFF

Los bits en **negrita** del principio de cada byte dicen qué papel cumple: 0 significa “byte ASCII suelto”, 110/1110/11110 significan “acá empieza una secuencia de 2, 3 o 4 bytes”, y 10 significa “soy continuación del byte anterior”. Se ve mejor en binario:

```
def mostrar_utf8(caracter):
    bs = caracter.encode("utf-8")
    print(f"{caracter!r} U+{ord(caracter):04X} -> {len(bs)} byte(s)")
    for b in bs:
        print(f"      {b:>3} 0b{b:08b}")

for c in ["a", "ñ", " ", " "]:
    mostrar_utf8(c)

'a' U+0061 -> 1 byte(s)
    97 0b01100001
'ñ' U+00F1 -> 2 byte(s)
    195 0b11000011
    177 0b10110001
' ' U+3042 -> 3 byte(s)
    227 0b11100011
    129 0b10000001
    130 0b10000010
' ' U+1F44B -> 4 byte(s)
    240 0b11110000
    159 0b10011111
    145 0b10010001
    139 0b10001011
```

Ese diseño explica por qué UTF-8 ganó, y por qué es la elección natural para un tokenizador:

- **Es retrocompatible con ASCII.** Los primeros 128 code points se codifican en un byte, con el mismo valor que en ASCII. Un archivo ASCII ya es un archivo UTF-8 válido.
- **Ningún byte de una secuencia multi-byte cae en el rango ASCII** (todos son > 128). Así que buscar un \n, una coma o un byte nulo byte a byte nunca da un falso positivo dentro de un carácter. Todo el software que ya trataba strings como bytes sigue funcionando.
- **Es auto-sincronizante.** Mirando un byte cualquiera sabés si es un comienzo o una continuación, así que podés retroceder hasta el límite del carácter sin leer desde el inicio.

UTF-32, en cambio, gasta 4 bytes siempre (mucho desperdicio para texto en inglés), y UTF-16 gasta 2 como mínimo, introduce bytes nulos que rompen las convenciones de C y necesita el mecanismo de *surrogate pairs* para los code points altos.

Si trabajamos sobre **bytes**, el vocabulario base tiene exactamente **256** entradas, un número chico y fijo para siempre. Y nunca existe el token “desconocido”: cualquier texto en cualquier idioma, presente o futuro, se puede representar, porque cualquier texto es una secuencia de bytes.

Carguemos los dos corpus con los que vamos a trabajar todo el apunte:

```
from pathlib import Path

_datos = Path("datos")
if not (_datos / "unicode_article.txt").exists():
```

```

_datos = Path("posts/datos")

texto_en = (_datos / "unicode_article.txt").read_text(encoding="utf-8")
texto_es = (_datos / "texto_es.txt").read_text(encoding="utf-8")

for nombre, t in [("inglés", texto_en), ("español", texto_es)]:
    bs = t.encode("utf-8")
    print(f"{nombre:>8}: {len(t):>6} caracteres    {len(bs):>6} bytes    "
          f"{len(bs)/len(t):.3f} bytes por carácter")

    inglés:  23328 caracteres    24597 bytes    1.054 bytes por carácter
    español:  23717 caracteres    24266 bytes    1.023 bytes por carácter

```

Los dos archivos tienen casi los mismos bytes a propósito, para que las comparaciones que vengan después sean de igual a igual. Que el texto en inglés tenga *más* bytes por carácter que el español puede sorprender, pero tiene explicación: el corpus en inglés es un artículo sobre Unicode y está lleno de emoji, banderas y texto Zalgo, que gastan 3 y 4 bytes por carácter. El “impuesto” que paga el español no aparece a nivel de bytes, sino más adelante, a nivel de tokens.

Y acá está el problema del segundo intento:

```

ids_en = list(texto_en.encode("utf-8"))

print(f"{len(texto_en)} caracteres -> {len(ids_en)} tokens (bytes)")
print(ids_en[:40], "...")

23328 caracteres -> 24597 tokens (bytes)
[65, 32, 80, 114, 111, 103, 114, 97, 109, 109, 101, 114, 226, 128, 153, 115, 32, 73, 110, 116,

```

Vocabulario chiquito y sin excepciones, pero secuencias larguísimas: un token por byte. Con una ventana de 1.000 tokens no entra ni media página. **Necesitamos comprimir**, sin perder la propiedad de que todo se puede representar.

Byte Pair Encoding

La solución es un algoritmo de compresión viejo, de 1994, que Sennrich, Haddow y Birch trajeron a NLP en [2016](#). La idea, en una línea:

Buscar el par de símbolos consecutivos más frecuente, reemplazarlo por un símbolo nuevo, y repetir.

Cada repetición agrega una entrada al vocabulario y acorta la secuencia. Con 256 símbolos iniciales (los bytes) y unos miles de repeticiones se llega a un vocabulario que representa palabras enteras frecuentes con un solo token, y palabras raras con varios.

Paso 1: contar pares

```

def get_stats(ids, counts=None):
    """Cuenta cuántas veces aparece cada par de ids consecutivos.

    El argumento `counts` permite acumular sobre un diccionario existente, que nos va a
    hacer falta en la sección 8 para contar sobre varios trozos de texto.
    """
    counts = {} if counts is None else counts
    for par in zip(ids, ids[1:]):
        counts[par] = counts.get(par, 0) + 1
    return counts

```

```
stats = get_stats(ids_en)
print(len(stats), "pares distintos\n")

for (a, b), n in sorted(stats.items(), key=lambda kv: -kv[1])[0:8]:
    texto = bytes([a, b]).decode("utf-8", errors="replace")
    print(f"({a:>3}, {b:>3}) = {texto!r:>6} {n:>4} veces")
```

1381 pares distintos

```
(101, 32) = 'e '      646 veces
(105, 110) = 'in'     446 veces
(115, 32) = 's '      424 veces
( 32, 116) = ' t'      405 veces
( 32, 97) = ' a'       377 veces
(116, 104) = 'th'      337 veces
(101, 114) = 'er'      294 veces
( 99, 111) = 'co'      290 veces
```

`zip(ids, ids[1:])` es el truco de siempre para recorrer pares consecutivos: `ids[1:]` es la misma lista corrida un lugar, así que `zip` los va emparejando.

El par más frecuente es (101, 32), que en ASCII es 'e' seguido de un espacio. Tiene sentido en un texto en inglés: es el final de *the*, *more*, *code*, *are*.

Paso 2: fusionar

```
def merge(ids, par, nuevo_id):
    """Reemplaza toda aparición consecutiva de `par` en `ids` por `nuevo_id`."""
    salida = []
    i = 0
    while i < len(ids):
        if i < len(ids) - 1 and ids[i] == par[0] and ids[i + 1] == par[1]:
            salida.append(nuevo_id)
            i += 2 # consumimos los dos ids del par
        else:
            salida.append(ids[i])
            i += 1
    return salida
```

```
print(merge([5, 6, 6, 7, 9, 1], (6, 7), 99))
```

```
[5, 6, 99, 9, 1]
```

La condición `i < len(ids) - 1` evita mirar `ids[i + 1]` en la última posición. El recorrido es **greedy de izquierda a derecha**, lo que importa cuando el par se solapa consigo mismo:

```
print(merge([6, 6, 6], (6, 6), 99))
```

```
[99, 6]
```

Hay dos ocurrencias de (6, 6) en [6, 6, 6] pero se puede fusionar solo una: la de la izquierda gana. No es un problema, es simplemente la convención, y lo importante es que `encode` y `decode` usen la misma.

Paso 3: repetir

Ahora el bucle de entrenamiento. Un parámetro: `vocab_size`, el tamaño final del vocabulario. Como arrancamos con 256 bytes, la cantidad de fusiones es `vocab_size - 256`:

```
def entrenar_bpe(ids, vocab_size, verbose=False):
    """Entrena BPE sobre una lista de ids y devuelve (ids comprimidos, merges, vocab).

    `merges` mapea (id, id) -> id nuevo, en el orden en que se aprendieron.
    `vocab` mapea id -> los bytes que representa.
    """
    assert vocab_size >= 256, "el vocabulario arranca en los 256 bytes"

    merges = {}
    vocab = {i: bytes([i]) for i in range(256)}
    ids = list(ids)  # copia: no destruimos la entrada

    for i in range(vocab_size - 256):
        stats = get_stats(ids)
        if not stats:  # quedó un solo id: no hay más pares
            break
        par = max(stats, key=stats.get)  # el par más frecuente
        nuevo_id = 256 + i  # los ids nuevos arrancan después de los byte
        ids = merge(ids, par, nuevo_id)
        merges[par] = nuevo_id
        vocab[nuevo_id] = vocab[par[0]] + vocab[par[1]]

        if verbose:
            legible = vocab[nuevo_id].decode("utf-8", errors="replace")
            print(f"{i + 1:>3}. {str(par):>12} -> {nuevo_id} {legible!r:>8} "
                  f"({stats[par]} apariciones)")

    return ids, merges, vocab
```

```
ids_bpe, merges, vocab = entrenar_bpe(ids_en, vocab_size=276, verbose=True)
```

1.	(101, 32) -> 256	'e '	(646 apariciones)
2.	(105, 110) -> 257	'in'	(446 apariciones)
3.	(115, 32) -> 258	's '	(424 apariciones)
4.	(116, 104) -> 259	'th'	(337 apariciones)
5.	(101, 114) -> 260	'er'	(294 apariciones)
6.	(99, 111) -> 261	'co'	(290 apariciones)
7.	(116, 32) -> 262	't '	(285 apariciones)
8.	(226, 128) -> 263	' '	(254 apariciones)
9.	(44, 32) -> 264	','	(243 apariciones)
10.	(97, 110) -> 265	'an'	(229 apariciones)
11.	(111, 114) -> 266	'or'	(214 apariciones)
12.	(100, 32) -> 267	'd '	(213 apariciones)
13.	(97, 114) -> 268	'ar'	(181 apariciones)
14.	(101, 110) -> 269	'en'	(174 apariciones)
15.	(257, 103) -> 270	'ing'	(166 apariciones)
16.	(261, 100) -> 271	'cod'	(165 apariciones)
17.	(121, 32) -> 272	'y '	(154 apariciones)
18.	(46, 32) -> 273	'.'	(154 apariciones)

19. (97, 108) -> 274 'al' (146 apariciones)
20. (259, 256) -> 275 'the ' (144 apariciones)

Vale la pena leer esa lista despacio, porque muestra el algoritmo pensando:

- Las primeras fusiones son las de la sección 4.1: 'e ', 'in', 's ', 'th'.
- La fusión 8, (226, 128), no es texto legible: son los dos primeros bytes de la secuencia UTF-8 de las comillas tipográficas " " ', muy frecuentes en el artículo. BPE no sabe nada de caracteres; encuentra la regularidad igual.
- La fusión 15 es (257, 103): el token 257 ('in') más 'g', o sea 'ing'. **Fusiona un token aprendido con otro.** Es la primera vez que el algoritmo construye sobre sí mismo, y es la razón de que con pocos miles de fusiones se llegue a palabras enteras.
- La última, (259, 256), junta 'th' con 'e ': el token 'the ', palabra y espacio en un solo id.

Los ids nuevos arrancan en 256 porque los primeros 256 ya están tomados por los bytes. Y el vocabulario final es un **árbol**: cada token nuevo se define en términos de dos anteriores, hasta llegar a las hojas, que son bytes.

¿Cuánto comprimimos?

```
print(f"antes:      {len(ids_en):>6} tokens")
print(f"después:   {len(ids_bpe):>6} tokens")
print(f"ratio:      {len(ids_en) / len(ids_bpe):>6.2f}X")

antes:      24597 tokens
después:    19438 tokens
ratio:       1.27X
```

Un 1,27X con apenas 20 fusiones. La curva sigue subiendo con `vocab_size`, y la vamos a medir en la [sección 12](#).

`merges` sale de contar pares en un texto concreto. Entrenado en este artículo en inglés, el tokenizador va a ser eficiente en inglés y malo en español o en código. Los tokenizadores reales se entrenan sobre mezclas cuidadosamente elegidas de idiomas y lenguajes de programación, y esa mezcla —decidida antes de entrenar el modelo, y congelada para siempre— determina qué le va a salir barato y qué caro al LLM.

Decodificar: de ids a texto

Tenemos `merges`, que es lo que se aprende. Para decodificar hace falta `vocab`: qué bytes representa cada id. Se construye recorriendo `merges`:

```
def construir_vocab(merges):
    vocab = {i: bytes([i]) for i in range(256)}
    for (p0, p1), idx in merges.items():
        vocab[idx] = vocab[p0] + vocab[p1]
    return vocab
```

```
vocab = construir_vocab(merges)
```

```
for idx in [256, 270, 275]:
    print(f"{idx}: {vocab[idx]}")
```

```
256: b'e '
270: b'ing'
275: b'the '
```

Ese `for` depende de un detalle importante de Python: **los diccionarios preservan el orden de inserción**. El token 275 se define como `vocab[259] + vocab[256]`, así que 259 y 256 ya tienen que estar en el diccionario cuando llegamos a él. Como `merges` se llenó en orden creciente de id, el recorrido funciona. Si iteráramos en otro orden, tendríamos un `KeyError`.

Con `vocab`, decodificar es concatenar bytes y decodificar UTF-8:

```
def decode(ids, vocab):
    bs = b"".join(vocab[i] for i in ids)
    return bs.decode("utf-8", errors="replace")

print(decode([275, 271, 270], vocab))      # 'the ' + 'cod' + 'ing'
the coding
```

Por qué `errors="replace"` no es opcional

Un detalle fácil de pasar por alto: **no toda secuencia de ids corresponde a texto UTF-8 válido**. El id 128 es el byte `0b10000000`, que según la tabla de la sección 3 es un byte de *continuación*. Suelto, sin un byte de comienzo antes, es inválido:

```
try:
    bytes([128]).decode("utf-8")
except UnicodeDecodeError as error:
    print("UnicodeDecodeError:", error)
```

UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0: invalid start byte

Un tokenizador tiene que ser una función total: si el modelo genera el id 128, `decode` no puede explotar. Por eso `errors="replace"`, que sustituye los bytes inválidos por el carácter de reemplazo U+FFFD:

```
print(repr(decode([128], vocab)))
', ,
```

Esto no es un caso hipotético. El LLM genera un token por vez, y si le pedís un emoji va a emitir varios tokens que juntos forman una secuencia UTF-8 de 4 bytes. Mientras la genera, los prefijos son inválidos. Por eso al usar un LLM en modo *streaming* a veces aparece un `UnicodeDecodeError` que después se convierte en el carácter correcto: es exactamente este mecanismo.

Codificar: de texto a ids

Nos falta la dirección que más se usa. Podría parecer que basta con buscar en `vocab` los bytes más largos que coincidan, pero no: hay que **reproducir exactamente el mismo proceso del entrenamiento**, aplicando las fusiones en el orden en que se aprendieron.

```
def encode(texto, merges):
    ids = list(texto.encode("utf-8"))

    while len(ids) >= 2:
        stats = get_stats(ids)
        # De todos los pares presentes, el que se aprendió PRIMERO (menor id nuevo).
        # Los pares que no están en `merges` reciben infinito, así nunca ganan.
        par = min(stats, key=lambda p: merges.get(p, float("inf")))
        if par not in merges:
            break
        # no queda nada por fusionar
        ids = merge(ids, par, merges[par])
```

```
return ids
```

```
print(encode("the code", merges))
```

```
[275, 271, 101]
```

Hay tres decisiones ahí que merecen explicación.

Por qué min y no max. En el entrenamiento usamos **max** sobre *frecuencias*. Acá usamos **min** sobre *ids nuevos*, que funcionan como un **ranking**: el id 256 fue la primera fusión aprendida, el 275 la última. Y hay que aplicarlas en ese orden porque las fusiones tardías se definen sobre las tempranas. El token 275 ('the ') es la fusión de 259 ('th') con 256 ('e '). Si fusionáramos primero por otro criterio, podríamos llegar a un estado donde el par (259, 256) nunca aparece, y produciríamos una tokenización que el entrenamiento nunca genera. Codificar es *replay* del entrenamiento, no búsqueda del mejor recubrimiento.

Por qué el break. **min** siempre devuelve algo, incluso si ningún par está en **merges**: en ese caso devuelve un par cualquiera con valor infinito. Sin el chequeo **par not in merges**, la línea siguiente daría **KeyError**, o peor, el bucle no terminaría nunca.

Por qué while len(ids) >= 2. Con menos de dos ids no hay pares, y **min** sobre un diccionario vacío levanta **ValueError**. Es el caso del string vacío y del carácter único:

```
print(encode("", merges), encode("a", merges))
```

```
[] [97]
```

Verificar que cierra

La propiedad que tiene que valer es que decodificar lo codificado devuelva el texto original:

```
print(decode(encode("hola mundo", merges), vocab) == "hola mundo")
```

```
print(decode(encode(texto_en, merges), vocab) == texto_en)
```

```
True
```

```
True
```

Y sobre texto que el tokenizador **nunca vio** al entrenar, que es el caso real:

```
validacion = (  
    "Muchos caracteres comunes, incluidos los numerales y la puntuación, no se tratan "  
    "como específicos de ningún sistema de escritura en particular.      "  
)
```

```
print(decode(encode(validacion, merges), vocab) == validacion)
```

```
True
```

Funciona con cualquier texto, y eso es consecuencia directa de trabajar sobre bytes: no hay caracteres fuera del alcance, así que no hace falta un token UNK.

`decode(encode(t)) == t` siempre. Pero `encode(decode(ids)) == ids` **no**: hay secuencias de ids que decodifican bien y que `encode` nunca produciría, porque no están en forma canónica.

```
ids_raros = [259, 256] # 'th' + 'e ', sin fusionar
```

```
texto_raro = decode(ids_raros, vocab)
```

```
print(f"{ids_raros} -> {texto_raro!r} -> {encode(texto_raro, merges)}")
```

```
[259, 256] -> 'the ' -> [275]
```

Los dos representan 'the ', pero solo el segundo es lo que sale del entrenamiento. El modelo solo vio la forma canónica, así que si le pasás la otra estás mandando una entrada que no aparece nunca en sus datos de entrenamiento.

El truco del regex: pre-tokenización

Ya tenemos un tokenizador BPE completo y funcionando. El problema es que si lo dejamos correr con un vocabulario grande, empieza a aprender tokens que no queremos. Entrenemos con 512 y busquemos los casos raros:

```
ids_512, merges_512, vocab_512 = entrenar_bpe(ids_en, vocab_size=512)
```

```
def mezcla_categorias(s):  
    """¿Este token junta cosas de distinta naturaleza (letras, símbolos, espacios)?"""  
    tiene_letra = any(c.isalpha() for c in s)  
    tiene_simbolo = any(not c.isalpha() and not c.isspace() for c in s)  
    espacio_interno = " " in s.strip()  
    return (tiene_letra and tiene_simbolo) or espacio_interno
```

```
sospechosos = [  
    (idx, vocab_512[idx].decode("utf-8", errors="replace"))  
    for idx in range(256, 512)  
]  
sospechosos = [(idx, s) for idx, s in sospechosos if mezcla_categorias(s)]  
  
print(f"{len(sospechosos)} de 256 tokens aprendidos mezclan categorías:\n")  
print(", ".join(repr(s) for _, s in sospechosos[:24]))
```

26 de 256 tokens aprendidos mezclan categorías:

'code poin', 's, ', 'UTF-', 'U+', 'e, ', ' 's ', 'code points ', 'es, ', ' . Th', 'UTF-8', 'code

Mirá esos tokens. Cosas como 'e. ', ' ', 's, ', 'ing ': el algoritmo se comió la puntuación junto con las letras. Con un vocabulario de decenas de miles esto explota, y trae tres problemas concretos:

- **Duplicación del vocabulario.** Van a existir tokens separados para 'dog', 'dog.', 'dog!', 'dog?', ' dog', ' Dog'. Cada uno es una fila distinta de la tabla de embeddings, y el modelo tiene que aprender en cada una, por separado, que se trata de un perro.
- **Se desperdicia capacidad.** Cada token gastado en una variante de puntuación es un token que no se usó para algo útil, como un término técnico o una palabra en otro idioma.
- **Se destruye la estructura del código.** En Python la indentación es sintaxis. Si BPE puede fusionar espacios libremente, cuatro espacios seguidos, ocho, doce y dieciséis terminan siendo tokens distintos, y los niveles de anidación se representan de formas incompatibles entre sí.

La solución de GPT-2: partir antes de contar

La solución que usa OpenAI desde GPT-2 es no correr BPE sobre el texto entero, sino:

1. partir el texto en trozos con una expresión regular;
2. correr BPE **dentro de cada trozo por separado**;
3. concatenar los resultados.

La clave es el paso 2: si las estadísticas de pares nunca cruzan el límite entre trozos, entonces **ningún token puede contener partes de dos trozos distintos**. Es una restricción dura, impuesta a mano por el diseñador del tokenizador, no algo que el algoritmo aprenda.

Este es el patrón de GPT-2, tal como está en su código publicado:

```
import regex as re

GPT2_SPLIT_PATTERN = (
    r""""s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?(?:\s\p{L}\p{N})+|\s+(?!\\S)|\s+"""
)

print(re.findall(GPT2_SPLIT_PATTERN, "Hola mundo, ¿cómo andás? Son 42 grados."))

['Hola', ' ', 'mundo', ',', ' ', '¿', 'cómo', ' ', 'andás', '?', ' ', 'Son', ' ', '42', ' ', 'grados', '.']
```

Primero, dos detalles de implementación:

- Se importa `regex`, **no** el módulo `re` de la biblioteca estándar. `regex` es una extensión compatible que agrega, entre otras cosas, las clases Unicode `\p{...}`. Se importa como `re` por costumbre, lo que confunde bastante al leer el código original.
- `\p{L}` significa “cualquier carácter que Unicode clasifique como letra”, **en cualquier idioma**: a, ñ, é, , . Y `\p{N}` es “cualquier dígito”. Con `[a-zA-Z]` habría que enumerar los alfabetos del mundo a mano.

El patrón es una cadena de alternativas separadas por `|`. `findall` recorre el texto de izquierda a derecha y en cada posición prueba las alternativas **en orden**, quedándose con la primera que coincide:

Alternativa	Qué captura	Ejemplo
's, 't, 're, 've, 'm, 'll, 'd ?\p{L}+	contracciones del inglés una o más letras, con un espacio opcional adelante	it's → it, 's hola mundo → hola, mundo
?\p{N}+	uno o más dígitos, con espacio opcional	son 42 → son, 42
?(?:\s\p{L}\p{N})+	símbolos: todo lo que no sea espacio, letra ni dígito	!!! , (
\s+(?!\\S)	espacios en blanco que no son seguidos por texto	ver abajo
\s+	cualquier resto de espacios	fin de string

La alternativa clave, y la más difícil de leer, es `\s+(?!\\S)`. `(?!\\S)` es un *negative lookahead*: “siempre que lo que sigue **no** sea un carácter distinto de espacio”. Combinado con la codicia de `\s+`, el efecto es “tomá una corrida de espacios, pero dejá el último para la palabra que viene”:

```
for s in ["hola mundo", "hola ", "renglón\n\nsiguiente"]:
    print(f"{s!r:>22} -> {re.findall(GPT2_SPLIT_PATTERN, s)}")

'hola mundo' -> ['hola', ' ', ' mundo']
'hola ' -> ['hola', ' ']
'renglón\n\nsiguiente' -> ['renglón', '\n', '\n', 'siguiente']
```

En el primer caso, los cuatro espacios se parten en ' ' (tres) más ' mundo'. Ese último espacio se va con la palabra porque, como vimos en la sección 1, la convención es que el espacio viaja pegado al inicio de la palabra siguiente. En el segundo caso no hay palabra que siga, así que los cuatro espacios quedan juntos.

Por qué existe esa convención: ' mundo' y 'mundo' no son el mismo token, pero como casi siempre una palabra viene después de un espacio, meter el espacio adentro del token es prácticamente gratis y ahorra un token por palabra. El precio lo pagamos en la sección 13.

Los agujeros del patrón de GPT-2

El propio código de GPT-2 tiene un comentario admitiendo el primero: las contracciones están escritas en minúscula y sin IGNORECASE.

```
for s in ["it's", "IT'S", "It'S"]:  
    print(f"{s!r:>8} -> {re.findall(GPT2_SPLIT_PATTERN, s)}")  
  
    "it's" -> ['it', "'s"]  
    "IT'S" -> ['IT', "'", 'S']  
    "It'S" -> ['It', "'", 'S']
```

En IT'S, el 'S no coincide con la alternativa de contracciones (que espera 's en minúscula), así que cae en la rama de símbolos y se parte distinto que la versión en minúscula. El modelo ve dos estructuras diferentes para la misma contracción.

El segundo agujero son los números: `?\p{N}+` no tiene límite de largo, así que un número de diez dígitos es un solo trozo y BPE puede aprender tokens con cualquier cantidad de dígitos adentro.

```
print(re.findall(GPT2_SPLIT_PATTERN, "el resultado fue 1234567890 exactamente"))  
['el', ' resultado', ' fue', ' 1234567890', ' exactamente']
```

El patrón de GPT-4

GPT-4 (el encoding `cl100k_base`) corrige eso:

```
GPT4_SPLIT_PATTERN = (  
    r""""(?:[sdmt]|ll|ve|re)|[^\r\n\p{L}\p{N}]?+\p{L}+|\p{N}{1,3}"""  
    r""""| ?[^\s\p{L}\p{N}]+|[\r\n]*|s*[\r\n]|s+(?!S)|s+"""  
)
```

```
pruebas = [  
    "IT'S",  
    "el resultado fue 1234567890 exactamente",  
    "¿cómo andás?",  
    "hola mundo",  
    "def f():\n    return 1\n",  
]
```

```
for s in pruebas:  
    print(f"{s!r}")  
    print(f"    GPT-2: {re.findall(GPT2_SPLIT_PATTERN, s)}")  
    print(f"    GPT-4: {re.findall(GPT4_SPLIT_PATTERN, s)}\n")  
  
"IT'S"  
    GPT-2: ['IT', "'", 'S']  
    GPT-4: ['IT', "'S"]  
  
'el resultado fue 1234567890 exactamente'  
    GPT-2: ['el', ' resultado', ' fue', ' 1234567890', ' exactamente']  
    GPT-4: ['el', ' resultado', ' fue', ' ', '123', '456', '789', '0', ' exactamente']  
  
'¿cómo andás?'  
    GPT-2: ['¿', 'cómo', ' andás', ' ?']  
    GPT-4: ['¿cómo', ' andás', ' ?']  
  
'hola mundo'
```

```
GPT-2: ['hola', ' ', ' mundo']
```

```
GPT-4: ['hola', ' ', ' mundo']
```

```
'def f():\n    return 1\n'
```

```
GPT-2: ['def', ' f', '():', '\n ', ' return', ' 1', '\n']
```

```
GPT-4: ['def', ' f', '():\n', ' ', ' return', ' ', '1', '\n']
```

Los cuatro cambios, en orden de aparición:

- `'(?:[sdmt]|ll|ve|re) — el (?:i:...) hace la comparación insensible a mayúsculas solo en ese grupo. IT'S ahora se parte igual que it's.`
- `[^\r\n\u{L}\u{N}]?+\u{L}+` — en lugar de “espacio opcional más letras”, ahora es “**cualquier carácter** que no sea salto de línea, letra ni dígito, opcional, más letras”. Así ¿cómo queda en un solo trozo, y también (hola o "palabra. El `?` es *possessive*: una vez que consumió el carácter, no lo devuelve por *backtracking*.
- `\u{N}{1,3}` — los números se parten en grupos de **a lo sumo tres dígitos**, y sin espacio adelante (fíjate que `' 1'` pasa a ser `' '` y `'1'`). Es la razón por la que 1234567890 se convierte en 123, 456, 789, 0, y tiene todo que ver con la aritmética de los LLMs (sección 13).
- `\s*[\r\n]` — los saltos de línea se tratan aparte. Fíjate en la última prueba: GPT-2 mete el salto de línea junto con la indentación que le sigue en un mismo trozo (`'\n '`), mientras GPT-4 lo pega a la puntuación anterior (`'():\n'`) y deja la indentación como trozo propio (`' ', ' return'`). En GPT-2, cada nivel de anidación produce un trozo distinto que combina salto de línea con espacios; en GPT-4, la indentación se agrupa aparte y de forma consistente. Esa diferencia, sola, explica buena parte de por qué GPT-4 escribe Python bastante mejor que GPT-2.

El patrón está diseñado alrededor del inglés. Las contracciones (`'s`, `'ll`, `'ve`) no existen en español, así que esas alternativas son peso muerto; y ningún patrón contempla que en español los signos de apertura `¿` y `¡` abren, ni que las palabras llevan tilde. GPT-4 al menos logra pegar `¿` a la palabra que sigue gracias a `[^\r\n\u{L}\u{N}]?+`, algo que GPT-2 no hacía.

Pero el gasto real no está en el regex sino en el corpus de entrenamiento del tokenizador, que es mayoritariamente inglés: las palabras en español simplemente no aparecen lo suficiente como para ganarse un token propio, y quedan partidas en pedazos. Lo medimos en la sección 9.

Todo junto: RegexTokenizer

Ya tenemos todas las piezas. Falta empaquetarlas en algo reutilizable que aplique el regex antes de entrenar y antes de codificar:

```
class RegexTokenizer:
    """Tokenizador BPE con pre-tokenización por expresión regular."""

    def __init__(self, pattern=None):
        self.pattern = pattern or GPT4_SPLIT_PATTERN
        self._re = re.compile(self.pattern)
        self.merges = {} # (id, id) -> id nuevo
        self.vocab = {i: bytes([i]) for i in range(256)} # id -> bytes

    def train(self, texto, vocab_size, verbose=False):
        # Un trozo por coincidencia del regex; BPE nunca cruza estos límites.
        trozos = [list(t.encode("utf-8")) for t in self._re.findall(texto)]

        for i in range(vocab_size - 256):
            # Contamos pares en todos los trozos, acumulando en el mismo diccionario.
            stats = {}
```

```

    for trozo in trozos:
        get_stats(trozo, stats)
    if not stats:
        break

    par = max(stats, key=stats.get)
    nuevo_id = 256 + i
    trozos = [merge(t, par, nuevo_id) for t in trozos]
    self.merges[par] = nuevo_id
    self.vocab[nuevo_id] = self.vocab[par[0]] + self.vocab[par[1]]

    if verbose:
        legible = self.vocab[nuevo_id].decode("utf-8", errors="replace")
        print(f"{nuevo_id}: {legible!r} ({stats[par]} apariciones)")

def _encode_bytes(self, bs):
    """BPE sobre los bytes de un solo trozo. Es el `encode` de la sección 6."""
    ids = list(bs)
    while len(ids) >= 2:
        stats = get_stats(ids)
        par = min(stats, key=lambda p: self.merges.get(p, float("inf")))
        if par not in self.merges:
            break
        ids = merge(ids, par, self.merges[par])
    return ids

def encode(self, texto):
    ids = []
    for trozo in self._re.findall(texto):
        ids.extend(self._encode_bytes(trozo.encode("utf-8")))
    return ids

def decode(self, ids):
    bs = b"".join(self.vocab[i] for i in ids)
    return bs.decode("utf-8", errors="replace")

```

Dos cosas a notar. En `train`, el diccionario `stats` se comparte entre todos los trozos: por eso `get_stats` recibía ese segundo argumento opcional. Y en `encode`, el bucle de fusiones corre por trozo, no sobre el texto completo, que es la mitad de codificación que le corresponde a la restricción impuesta en el entrenamiento.

Entrenemos las dos variantes de patrón y comparémoslas contra la versión sin regex, todas con el mismo vocabulario de 512:

```

tok_gpt2 = RegexTokenizer(GPT2_SPLIT_PATTERN)
tok_gpt2.train(texto_en, vocab_size=512)

tok = RegexTokenizer(GPT4_SPLIT_PATTERN)
tok.train(texto_en, vocab_size=512)

```

```

def tokens_mezclados(vocab):
    return [
        vocab[idx].decode("utf-8", errors="replace")
        for idx in range(256, 512)
    ]

```

```

        if mezcla_categorias(vocab[idx].decode("utf-8", errors="replace"))
    ]

bytes_totales = len(ids_en)
variantes = [
    ("sin regex", len(ids_512), tokens_mezclados(vocab_512)),
    ("regex GPT-2", len(tok_gpt2.encode(texto_en)), tokens_mezclados(tok_gpt2.vocab)),
    ("regex GPT-4", len(tok.encode(texto_en)), tokens_mezclados(tok.vocab)),
]

print(f"bytes originales: {bytes_totales}\n")
for nombre, n_tokens, mezclados in variantes:
    print(f"{nombre:>12}: {n_tokens:>6} tokens ({bytes_totales / n_tokens:.2f}X) "
          f"{len(mezclados):>2} tokens que mezclan categorías")

bytes originales: 24597

```

sin regex:	11442 tokens	(2.15X)	26 tokens que mezclan categorías
regex GPT-2:	11699 tokens	(2.10X)	0 tokens que mezclan categorías
regex GPT-4:	11694 tokens	(2.10X)	4 tokens que mezclan categorías

Dos lecturas de esa tabla.

La compresión baja. Con regex comprimimos *menos*, y era esperable: le prohibimos al algoritmo justamente las fusiones más rentables, las que cruzan categorías. Es el precio que se paga a cambio de un vocabulario más limpio, y es una decisión de diseño deliberada.

El regex hace lo que promete, con una excepción interesante. Con el patrón de GPT-2 no queda ni un token mezclado. Con el de GPT-4 quedan unos pocos:

```

print(tokens_mezclados(tok.vocab))

[''s', '-b', ''t', '-bit']

```

No es un error: es exactamente la rama `[^r\np{L}p{N}]?+\p{L}+` que agregó GPT-4, la que permite pegar **un** símbolo delante de una palabra. Es la misma que hace que `¿cómo` quede entero, y trae de arrastre el guion de `-bit` y el apóstrofo tipográfico de `'s`. GPT-4 relajó a propósito la restricción de GPT-2 para manejar mejor los idiomas que no son inglés.

Y así se ven los tokens que aprendió el tokenizador con el patrón de GPT-4:

```

aprendidos = [tok.vocab[idx].decode("utf-8", errors="replace") for idx in range(256, 512)]
print(", ".join(repr(s) for s in aprendidos[:40]))
print("...")
print(", ".join(repr(s) for s in aprendidos[-20:]))

'in', ' t', ' a', 'er', 'co', ' th', ' ', ' s', ' o', 'de', 're', 'it', ' co', ' the', 'ing', '
...
' r', 'ight', 'age', ' do', ' not', 'nder', 'pe', 'quen', ' E', ' j', ' li', 'am', ' character
', ' +', 'composed', ' _', ' '

```

Los primeros son bigramas del inglés; los últimos, palabras enteras con su espacio adelante. Se ve el algoritmo escalando de a poco de letras a palabras. Y el viaje redondo sigue valiendo:

```

print(tok.decode(tok.encode(texto_en)) == texto_en)
print(tok.decode(tok.encode(validacion)) == validacion)

True
True

```

Comparación contra el tokenizador de verdad

tiktoken trae los tokenizadores reales de los modelos de OpenAI. Son tres generaciones:

```
encodings = {
    "gpt2": tiktoken.get_encoding("gpt2"),           # GPT-2, GPT-3
    "cl100k_base": tiktoken.get_encoding("cl100k_base"), # GPT-3.5, GPT-4
    "o200k_base": tiktoken.get_encoding("o200k_base"), # GPT-4o y posteriores
}
```

```
for nombre, e in encodings.items():
    print(f"{nombre:>12}: vocabulario de {mil(e.n_vocab)} tokens")
```

```
gpt2: vocabulario de 50.257 tokens
cl100k_base: vocabulario de 100.277 tokens
o200k_base: vocabulario de 200.019 tokens
```

Y así rinden sobre nuestros dos corpus, que recordemos tienen prácticamente los mismos bytes:

```
print(f"{'encoding':>12} {'inglés':>10} {'español':>10} {'sobrecosto':>12}")
for nombre, e in encodings.items():
    n_en = len(e.encode(texto_en))
    n_es = len(e.encode(texto_es))
    print(f"{nombre:>12} {n_en:>10} {n_es:>10} {100 * (n_es / n_en - 1):>11.0f}%")
```

```
print()
print(f"{'nuestro (512)':>12} {len(tok.encode(texto_en)):>10} "
      f"{len(tok.encode(texto_es)):>10}")
```

encoding	inglés	español	sobrecosto
gpt2	7019	9104	30%
cl100k_base	6564	7380	12%
o200k_base	6447	6510	1%

```
nuestro (512)      11694      15583
```

Ahí está, medido, el **impuesto del español**: con el tokenizador de GPT-2, el mismo contenido en español consume un 30 % más de tokens que en inglés. Eso significa 30 % más de costo por API, 30 % menos de texto en la ventana de contexto y 30 % más de cómputo por la misma información. Con `cl100k_base` la brecha baja a 12 %, y con `o200k_base` a casi nada: cada generación de tokenizadores mejoró la cobertura multilingüe, agrandando el vocabulario y balanceando mejor el corpus de entrenamiento.

Nuestro tokenizador de 512 tokens, entrenado sobre 24 KB de inglés, obviamente pierde por mucho. Lo interesante es que la diferencia es de escala, no de algoritmo: `cl100k_base` es el mismo código que escribimos, con 100.256 fusiones en lugar de 256 y un corpus de entrenamiento gigantesco.

Los bytes están permutados

Si intentamos leer las fusiones de GPT-4 nos topamos con dos sorpresas. La primera:

```
cl = encodings["cl100k_base"]
byte_shuffle = {i: cl._mergeable_ranks[bytes([i])] for i in range(256)}

print("¿el byte i tiene id i?", all(k == v for k, v in byte_shuffle.items()))
print("primeros:", list(byte_shuffle.items())[:6])
print("el espacio (byte 32) tiene id", byte_shuffle[32])

¿el byte i tiene id i? False
```

```
primeros: [(0, 188), (1, 189), (2, 190), (3, 191), (4, 192), (5, 193)]
el espacio (byte 32) tiene id 220
```

Los 256 bytes iniciales **no** están en el orden obvio: el byte 0 es el id 188, el espacio es el 220. Es una permutación arbitraria, heredada de una decisión de implementación de GPT-2, sin ningún significado. Cualquier implementación que quiera producir los mismos ids que OpenAI tiene que aplicar esta permutación al codificar y la inversa al decodificar.

Recuperar las fusiones

La segunda sorpresa: tiktoken no guarda merges. Guarda `_mergeable_ranks`, que mapea **bytes** → **rango**, sin decir de qué dos padres salió cada token. Pero se puede reconstruir: para un token cualquiera, sus dos padres son los que resultan de aplicarle BPE usando solo las fusiones **anteriores** a él.

```
def bpe(ranks, token, max_rank):
    """Aplica BPE a `token` usando solo fusiones de rango menor a `max_rank`."""
    partes = [bytes([b]) for b in token]
    while True:
        mejor_i, mejor_rango = None, None
        for i, (a, b) in enumerate(zip(partes, partes[1:])):
            rango = ranks.get(a + b)
            if rango is not None and (mejor_rango is None or rango < mejor_rango):
                mejor_i, mejor_rango = i, rango
        if mejor_rango is None or mejor_rango >= max_rank:
            break
        partes = (partes[:mejor_i]
                  + [partes[mejor_i] + partes[mejor_i + 1]]
                  + partes[mejor_i + 2:])
    return partes

def recuperar_merges(ranks):
    merges = {}
    for token, rango in ranks.items():
        if len(token) == 1:
            continue # los bytes sueltos no son fusiones
        p0, p1 = bpe(ranks, token, max_rank=rango)
        merges[(ranks[p0], ranks[p1])] = rango
    return merges
```

```
merges_gpt4 = recuperar_merges(cl._mergeable_ranks)
print(f"{mil(len(merges_gpt4))} fusiones recuperadas")
```

100.000 fusiones recuperadas

Y ahora sí podemos ver las **primeras fusiones que aprendió GPT-4**, que es una de las cosas más reveladoras de todo el apunte:

```
inverso = {rango: token for token, rango in cl._mergeable_ranks.items()}

for (a, b), rango in sorted(merges_gpt4.items(), key=lambda kv: kv[1][:8]):
    print(f"{rango}: {inverso[a]!r} + {inverso[b]!r} = {inverso[rango]!r}")

256: b' ' + b' ' = b' '
257: b' ' + b' ' = b' '

```

```

258: b'i' + b'n' = b'in'
259: b' ' + b't' = b' t'
260: b'   ' + b'   ' = b'   '
261: b'e' + b'r' = b'er'
262: b'  ' + b'  ' = b'  '
263: b'o' + b'n' = b'on'

```

La primera fusión de GPT-4 es **dos espacios**. La segunda, cuatro espacios. La quinta, ocho. El tokenizador de GPT-4 dedica sus primerísimas fusiones —las más valiosas, las que se aplican antes que todo lo demás— a comprimir corridas de espacios. Es la huella de un corpus con muchísimo código fuente, y la contracara de aquel `\s*[\r\n]` que agregaron al patrón. Comparado con GPT-2, que gastaba un token por cada espacio de indentación, es una mejora enorme para programar.

Tokens especiales

Un tokenizador que solo hace BPE alcanza para representar texto, pero un LLM necesita además marcar **estructura**: dónde termina un documento, dónde empieza el turno del usuario, dónde va a ir el relleno en un autocompletado. Para eso existen los **tokens especiales**: ids que no salen de ninguna fusión y que no corresponden a ningún texto.

```

for nombre, e in encodings.items():
    print(f"{nombre}:")
    for token, idx in e._special_tokens.items():
        print(f"    {idx} {token}")

```

```

gpt2:
    50256 <|endoftext|>
cl100k_base:
    100257 <|endoftext|>
    100258 <|fim_prefix|>
    100259 <|fim_middle|>
    100260 <|fim_suffix|>
    100276 <|endofprompt|>
o200k_base:
    199999 <|endoftext|>
    200018 <|endofprompt|>

```

`<|endoftext|>` es el más importante: es lo que se intercala entre documentos no relacionados al armar el corpus de entrenamiento. Sin él, el modelo aprendería que después de una receta de tortas viene el código fuente de Linux. Es la señal de “borrá el contexto, esto es otra cosa”.

Los `<|fim_*|>` son de *fill in the middle*: permiten pedirle al modelo que complete en el medio de un texto dándole el prefijo y el sufijo, que es exactamente lo que hace un autocompletado de código en un editor. Y los modelos de chat agregan más (`<|im_start|>`, `<|im_end|>`) para delimitar los turnos de la conversación: el “formato de chat” no es más que texto con tokens especiales adentro.

Dado que no salen de BPE, hay que manejarlos aparte: se busca el token especial en el texto, se lo saca, y a los pedazos que quedan se les aplica BPE normalmente. Agreguemos eso a nuestra clase:

```

class TokenizerConEspeciales(RegexTokenizer):

    def __init__(self, pattern=None):
        super().__init__(pattern)
        self.especiales = {}
        self.especiales_inv = {}

    def registrar_especiales(self, especiales):

```

```

"""`especiales` es un dict texto -> id. Los ids van por encima del vocabulario."""
self.especiales = dict(especiales)
self.especiales_inv = {idx: txt for txt, idx in especiales.items()}

def encode(self, texto, permitir_especiales=True):
    if not permitir_especiales or not self.especiales:
        return super().encode(texto)

    # Partimos por los tokens especiales, conservándolos (el grupo de captura hace
# que `split` los devuelva como elementos de la lista).
    patron = "(" + "|".join(re.escape(t) for t in self.especiales) + ")"

    ids = []
    for parte in re.split(patron, texto):
        if parte in self.especiales:
            ids.append(self.especiales[parte])
        elif parte:
            ids.extend(super().encode(parte))
    return ids

def decode(self, ids):
    piezas = []
    for idx in ids:
        if idx in self.vocab:
            piezas.append(self.vocab[idx])
        elif idx in self.especiales_inv:
            piezas.append(self.especiales_inv[idx].encode("utf-8"))
        else:
            raise ValueError(f"id fuera del vocabulario: {idx}")
    return b"".join(piezas).decode("utf-8", errors="replace")

```

```

tok_esp = TokenizerConEspeciales()
tok_esp.merges = tok.merges # reusamos lo aprendido en la sección 8
tok_esp.vocab = tok.vocab
tok_esp.registrar_especiales({"<|endoftext|>": 512, "<|usuario|>": 513})

```

```

mensaje = "<|usuario|>the code<|endoftext|>"
ids = tok_esp.encode(mensaje)

```

```

print(ids)
print(tok_esp.decode(ids))

```

```

[513, 325, 101, 298, 512]
<|usuario|>the code<|endoftext|>

```

Comparemos con lo que pasa si no los tratamos como especiales:

```

print(tok_esp.encode(mensaje, permitir_especiales=False))

```

```

[60, 124, 316, 117, 280, 105, 111, 124, 62, 325, 101, 298, 60, 124, 271, 100, 111, 102, 116, 3

```

Sin el tratamiento especial, <|usuario|> se convierte en una docena de tokens comunes: es texto como cualquier otro. Y ahí está el problema de seguridad: si tu aplicación pega la entrada del usuario en un prompt y **sí** habilita los tokens especiales, un usuario puede escribir literalmente <|im_end|> y cerrar su propio turno, haciéndose pasar por el sistema. Por eso **tiktoken** obliga a pedirlo explícitamente:

```
try:
    cl.encode("<|endoftext|> hola")
except ValueError as error:
    print(str(error)[:180], "...")

print(cl.encode("<|endoftext|> hola", allowed_special="all"))
```

Encountered text corresponding to disallowed special token '<|endoftext|>'.
 If you want this text to be encoded as a special token, pass it to `allowed\_special`, e.g. `al

[100257, 305, 8083]

Por defecto levanta excepción; hay que decir `allowed_special` a propósito. La regla práctica es tratar el texto del usuario siempre con los especiales **deshabilitados**, y construir la estructura del prompt vos, desde el código.

SentencePiece: lo que usan Llama y Mistral

Fuera de OpenAI, la biblioteca más usada es [SentencePiece](#) (Llama, Mistral, Gemma). Hace BPE, igual que nosotros, pero con una diferencia de fondo:

tiktoken codifica el texto a UTF-8 y corre BPE **sobre los bytes**. **SentencePiece** corre BPE **sobre los code points Unicode**, y recién al final decide qué hacer con los caracteres raros.

Ese orden invertido explica toda la lista de opciones raras que hay que configurar:

- **character_coverage** (típicamente 0.9995): qué proporción de los caracteres del corpus entran al vocabulario base. Los que quedan afuera son “caracteres raros”, y hay que decidir qué hacer con ellos.
- **byte_fallback**: si está en `True`, los caracteres que quedaron afuera se descomponen en sus bytes UTF-8, que sí están en el vocabulario. Es el parche que hace que SentencePiece se comporte parecido a un tokenizador de bytes. Llama 2 lo usa.
- **unk_id**: si **byte_fallback** está en `False`, los caracteres raros se mapean todos al token `<unk>`, y ahí la información **se pierde**: `decode(encode(t)) != t`. Nada de eso puede pasar en un tokenizador de bytes.
- **add_dummy_prefix**: agrega un espacio al principio de todo texto, para que "hola" y " hola" se tokenicen igual. Es una decisión distinta a la de OpenAI, que prefiere que sean tokens diferentes.
- La normalización Unicode (NFKC por defecto) puede **modificar el texto** antes de tokenizar.

También cambia la notación: SentencePiece representa el espacio con el carácter (U+2581), así que los tokens se ven como `hola, mundo`.

`sentencepiece` no tiene wheel para Python 3.13 y compilarlo requiere `cmake`, así que este bloque está marcado para no ejecutarse al renderizar. El código es correcto: si lo corrés en un entorno con la biblioteca instalada (Colab la trae), funciona tal cual.

```
import sentencepiece as spm

spm.SentencePieceTrainer.train(
    input="datos/texto_es.txt",
    model_prefix="tok_es",
    model_type="bpe",
    vocab_size=512,
    character_coverage=0.9995,
    byte_fallback=True,          # sin esto, los caracteres raros se pierden en <unk>
    add_dummy_prefix=True,
    normalization_rule_name="identity", # sin normalización, para no alterar el texto
)
```

```

sp = spm.SentencePieceProcessor(model_file="tok_es.model")

print(sp.encode("hola mundo", out_type=str))
# -> ['hola', 'mun', 'do'] (el es el espacio)
print(sp.encode(" ", out_type=str))
# con byte_fallback=True se parte en bytes UTF-8: ['<0xEC>', '<0x95>', ...]

```

El resumen práctico: SentencePiece tiene muchas más perillas, arrastra decisiones históricas y puede perder información. Trabajar sobre bytes desde el principio, como hicimos nosotros, evita todos esos problemas de raíz.

Cuánto vocabulario conviene

vocab_size es el único hiperparámetro real del tokenizador, y no se puede elegir mirando una sola cosa. Del lado de las ventajas, más vocabulario comprime más. Midámoslo: entrenemos hasta 2048 registrando cuántos tokens quedan después de cada fusión.

```

def curva_compresion(texto, vocab_max, pattern=None):
    """Entrena BPE registrando el largo de la secuencia después de cada fusión."""
    compilado = re.compile(pattern or GPT4_SPLIT_PATTERN)
    trozos = [list(t.encode("utf-8")) for t in compilado.findall(texto)]

    largo = lambda: sum(len(t) for t in trozos)
    curva = [(256, largo())]

    for i in range(vocab_max - 256):
        stats = {}
        for trozo in trozos:
            get_stats(trozo, stats)
        if not stats:
            break
        par = max(stats, key=stats.get)
        trozos = [merge(t, par, 256 + i) for t in trozos]
        curva.append((257 + i, largo()))

    return curva

curva_en = curva_compresion(texto_en, 2048)
curva_es = curva_compresion(texto_es, 2048)

for v, n in curva_en[::384] + [curva_en[-1]]:
    print(f"vocab {v:>5}: {n:>6} tokens ({len(ids_en) / n:.2f}X)")

vocab 256: 24597 tokens (1.00X)
vocab 640: 10382 tokens (2.37X)
vocab 1024: 8357 tokens (2.94X)
vocab 1408: 7367 tokens (3.34X)
vocab 1792: 6662 tokens (3.69X)
vocab 2048: 6406 tokens (3.84X)

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(7, 4))

```

```

for curva, etiqueta, texto_fuente in [
    (curva_en, "inglés", texto_en),
    (curva_es, "español", texto_es),
]:
    bytes_totales = len(texto_fuente.encode("utf-8"))
    vs = [v for v, _ in curva]
    ratios = [bytes_totales / n for _, n in curva]
    ax.plot(vs, ratios, label=etiqueta)

ax.set_xlabel("tamaño del vocabulario")
ax.set_ylabel("compresión (bytes por token)")
ax.legend()
ax.grid(alpha=0.3)
plt.show()

```

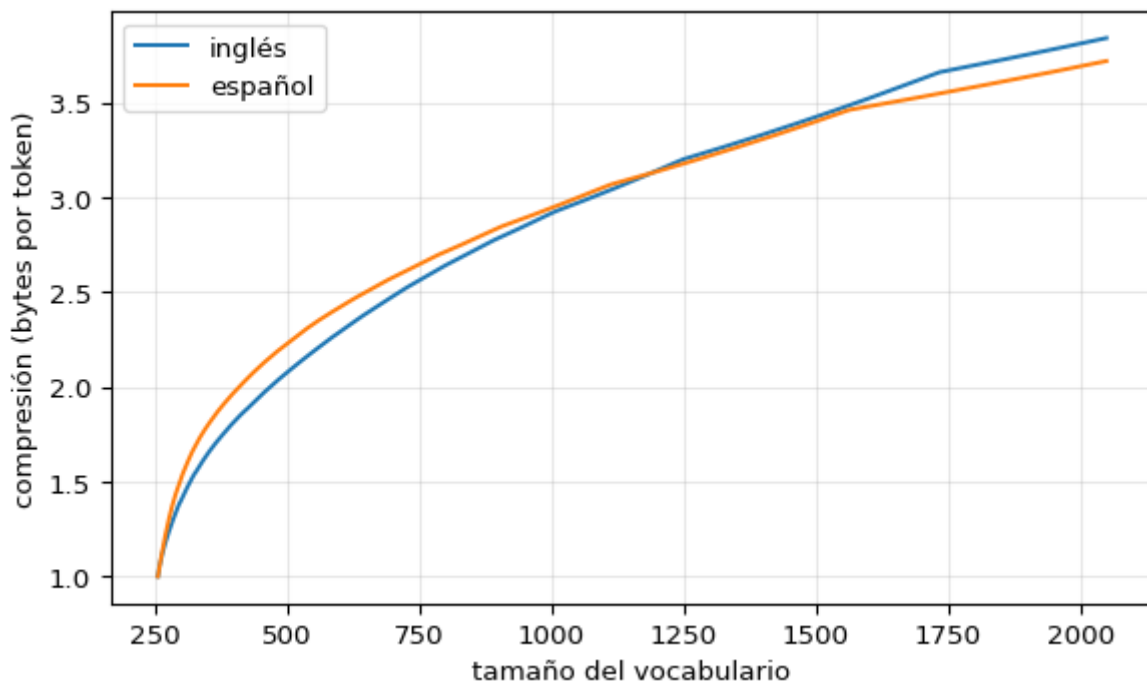


Figura 1: Compresión en función del tamaño del vocabulario, para los dos corpus (24 KB cada uno). Las dos curvas se aplanan: cada fusión nueva rinde menos que la anterior.

La curva se aplana: las primeras fusiones capturan los pares realmente frecuentes y las últimas apenas rascan. **Rendimientos decrecientes.**

Y hay algo más en ese gráfico que conviene señalar: las dos curvas van prácticamente pegadas. Un tokenizador entrenado **en español** comprime español igual de bien que uno entrenado en inglés comprime inglés. El idioma no tiene nada de intrínsecamente caro; lo caro es usar un tokenizador entrenado en otro idioma. Es exactamente el sobre costo que medimos en la sección 9.

Del otro lado del balance hay tres costos.

Cuesta parámetros. La tabla de embeddings tiene $\text{vocab_size} \times \text{n_embd}$ parámetros, y en la salida hace falta otra matriz del mismo tamaño para proyectar a logits:

```

n_embd = 768 # el ancho de GPT-2 small
for v in [1_024, 50_257, 100_277, 200_019]:
    tabla = v * n_embd
    print(f"vocab {mil(v):>9}: {tabla / 1e6:>5.1f}M parámetros por tabla ")

```

```
f"({100 * tabla / 124e6:>4.1f})% de los 124M de GPT-2 small)"
```

```
vocab    1.024:    0.8M parámetros por tabla ( 0.6% de los 124M de GPT-2 small)
vocab    50.257:   38.6M parámetros por tabla (31.1% de los 124M de GPT-2 small)
vocab   100.277:   77.0M parámetros por tabla (62.1% de los 124M de GPT-2 small)
vocab   200.019:  153.6M parámetros por tabla (123.9% de los 124M de GPT-2 small)
```

GPT-2 small tiene 124M parámetros en total, y con su vocabulario de 50.257 tokens son 38,6M —casi un tercio del modelo— los que se van en la tabla de embeddings. GPT-2 comparte esa matriz con la proyección de salida (*weight tying*), así que la paga una sola vez; los modelos que no lo hacen la pagan dos veces. En cualquier caso, duplicar el vocabulario no es gratis.

Cuesta cómputo en la salida. Cada paso de generación calcula un softmax sobre todo el vocabulario. Con 200.000 tokens es una operación considerable, que además hay que hacer una vez por token generado.

Los tokens raros quedan mal entrenados. Cuanto más grande el vocabulario, menos veces aparece cada token en el corpus. Un token que apareció tres veces en todo el entrenamiento tiene un embedding que es prácticamente su inicialización aleatoria. Volvemos a esto en la sección siguiente, con el caso de SolidGoldMagikarp.

Se puede agregar tokens a un modelo ya entrenado: se agranda la tabla de embeddings, se inicializan las filas nuevas (idealmente con el promedio de los embeddings de los tokens que ese texto usaba antes) y se hace fine-tuning. Es lo habitual al adaptar un modelo a un idioma nuevo o a un dominio técnico, y también la base de técnicas de compresión de prompts, donde se entrenan tokens nuevos que resumen instrucciones enteras.

Las rarezas, ya explicadas

Volvamos a la lista de la [sección 1](#). Ahora tenemos todo para responderla.

Por qué no sabe deletrear ni contar letras

```
o200 = encodings["o200k_base"]

for palabra in ["constitución", "ubiquitous", "DefaultCellStyle"]:
    trozos = [o200.decode([i]) for i in o200.encode(palabra)]
    print(f"{palabra:>18} -> {trozos}")

    constitución -> ['constit', 'ución']
    ubiquitous -> ['ub', 'iqu', 'it', 'ous']
    DefaultCellStyle -> ['Default', 'Cell', 'Style']
```

Si le preguntás cuántas letras `t` tiene `constitución`, el modelo no ve letras: ve dos tokens, `'constit'` y `'ución'`. Contar letras adentro de un token requiere que el modelo haya memorizado, durante el entrenamiento, la composición de cada token en caracteres. Algunos la aprenden parcialmente, y por eso a veces acierta. Pero no está *mirando* las letras: está recordando de memoria.

Por la misma razón, si le pedís que invierta un string falla, y si le pedís “primero separa las letras con guiones y después invertí la lista” acierta: al separarlas con guiones las convertís en tokens individuales, y recién ahí las puede manipular.

Por qué falla en aritmética

```
gpt2, o200 = encodings["gpt2"], encodings["o200k_base"]

for numero in ["1234", "1257", "9999999"]:
    print(f"{numero:>9}: gpt2 {[gpt2.decode([i]) for i in gpt2.encode(numero)]}")
    f" o200k {[o200.decode([i]) for i in o200.encode(numero)]}")
```

```
1234:  gpt2 ['12', '34']    o200k ['123', '4']
1257:  gpt2 ['12', '57']    o200k ['125', '7']
9999999: gpt2 ['9999', '999']  o200k ['999', '999', '9']
```

1234 se parte en $123 + 4$, y 1257 en $125 + 7$. La partición no tiene nada que ver con la estructura decimal del número: no separa unidades, decenas ni centenas, y dos números parecidos se parten distinto. Para sumar, el modelo tiene que aprender aritmética sobre representaciones fragmentadas de forma arbitraria. Que funcione tan bien como funciona es notable; que se equivoque con números largos, esperable.

Por qué el español cuesta más

Ya lo medimos en la sección 9: mismo contenido, 30 % más de tokens con GPT-2. La causa es doble. Los caracteres con tilde ocupan dos bytes en UTF-8, así que arrancan costando el doble. Y sobre todo, las palabras en español aparecen mucho menos en el corpus del tokenizador, así que casi ninguna se ganó un token propio y quedan partidas en pedazos.

Por qué es peor escribiendo Python

```
codigo = "def f(x):\n    if x > 0:\n        return x\n    return -x\n"
```

```
for nombre, e in encodings.items():
    print(f"{nombre:>12}: {len(e.encode(codigo)):>3} tokens")
```

```
print()
print("gpt2  :", [gpt2.decode([i]) for i in gpt2.encode(codigo)][:14])
print("cl100k:", [cl.decode([i]) for i in cl.encode(codigo)][:14])
```

```

gpt2: 32 tokens
cl100k_base: 20 tokens
o200k base: 20 tokens

```

```
gpt2 : ['def', ' f', '(', 'x', ')', ':', '\\n', ' ', ' ', ' ', ' ', ' if', ' x', ' >', ' 0', ':']
cl100k: ['def', ' f', '(x', '):\\n', ' ', ' ', ' if', ' x', ' >', ' ', '0', ':\\n', ' ', ' ', ' re
```

El mismo código: 32 tokens con GPT-2, 20 con GPT-4. Mirá los trozos: GPT-2 gasta **un token por cada espacio** de indentación, mientras GPT-4 agrupa la indentación en un solo token. En Python la indentación es sintaxis, así que GPT-2 desperdiciaba una parte enorme de su contexto en espacios, y encima tenía que aprender la estructura del código a partir de secuencias larguísimas de tokens idénticos. En JavaScript, donde los espacios son opcionales, el problema era mucho menor. De ahí la fama.

Por qué molesta un espacio al final del prompt

```
for s in ["hola", "hola ", "hola "]:
    ids = o200.encode(s)
    print(f"{s!r:>9} -> {ids} {[o200.decode([i]) for i in ids]}")
```

```
'hola' -> [122845] ['hola']
'hola ' -> [122845, 220] ['hola', ' ']
'hola  ' -> [122845, 256] ['hola', '  ']
```

'hola ' no es 'hola' más un espacio en el mismo token: es 'hola' seguido de un token de espacio suelto. Y como vimos en la sección 7, la convención es que los espacios viajan **pegados a la palabra que sigue**. Un espacio solo, huérfano, es una secuencia que casi no aparece en los datos de entrenamiento. El modelo queda parado en una parte rarísima de su distribución, y encima tiene que

“adivinar” que la próxima palabra ya no debería empezar con espacio. De ahí que las APIs adviertan sobre los espacios finales en el prompt.

Por qué existen tokens que rompen el modelo

```
print("gpt2 ' SolidGoldMagikarp' ->", gpt2.encode(" SolidGoldMagikarp"))
print("cl100k ' SolidGoldMagikarp' ->", cl100k.encode(" SolidGoldMagikarp"))

gpt2 ' SolidGoldMagikarp' -> [43453]
cl100k ' SolidGoldMagikarp' -> [22925, 26509, 34015, 1609, 8035]
```

En GPT-2, ' SolidGoldMagikarp' es **un solo token**. Era el nombre de un usuario de Reddit muy activo, presente miles de veces en el corpus con el que se entrenó el *tokenizador*. Pero ese corpus no era el mismo con el que se entrenó el *modelo*: en los datos del modelo, ese token aparecía casi nunca. Resultado: una fila de la tabla de embeddings que quedó prácticamente en su inicialización aleatoria.

Cuando se le pedía a GPT-2 o GPT-3 repetir esa palabra, el modelo hacía cosas erráticas: insultaba, cambiaba de tema, alucinaba. Son los **tokens no entrenados** (*undertrained tokens*), y aparecen cuando el corpus del tokenizador y el del modelo no coinciden. En `cl100k_base` ya no existe: se parte en cinco tokens comunes.

Por qué conviene YAML antes que JSON

```
import json

datos = {"nombre": "Ana", "edad": 31, "ciudad": "Montevideo", "activo": True}

como_json = json.dumps(datos, indent=2, ensure_ascii=False)
como_yaml = "nombre: Ana\nedad: 31\nciudad: Montevideo\nactivo: true\n"

print(f"JSON: {len(o200.encode(como_json)):>3} tokens")
print(f"YAML: {len(o200.encode(como_yaml)):>3} tokens")

JSON:  31 tokens
YAML:  19 tokens
```

Los mismos datos, 60 % más de tokens en JSON. Las llaves, los corchetes, las comillas y la indentación son todos tokens, y no aportan información. Cuando el contexto o el costo importan, la elección del formato es una decisión de ingeniería con impacto medible.

Para seguir

Lo que construimos son unas 60 líneas de Python, y es esencialmente el mismo algoritmo que corre adentro de GPT-4. Lo que cambia es la escala del corpus, el tamaño del vocabulario y una capa de ingeniería para que sea rápido.

Vale la pena quedarse con la idea de fondo: el tokenizador es una etapa **separada del modelo, sin aprendizaje por gradiente, y congelada para siempre** una vez que se empieza a entrenar. Todo lo que decida —qué idiomas comprime bien, cómo parte los números, si la indentación cuesta caro— se convierte en una restricción permanente del modelo. Es la parte menos glamorosa de un LLM y la que explica una fracción sorprendente de sus fallas.

Para practicar, el recorrido de Karpathy en [minbpe](#) cubre lo mismo de forma ejecutable y termina con ejercicios: entrenar un tokenizador propio, barrer `vocab_size` sobre otro corpus, y reproducir los ids de un tokenizador de referencia a partir de las fusiones.

Recursos

- Karpathy, A. [Let's build the GPT Tokenizer](#) — la clase en la que se basa este apunte.

- Karpathy, A. [minbpe](#) — implementación de referencia y los ejercicios originales.
- [tiktokenizer](#) — pegás texto y ves los tokens de cada modelo. Insustituible para desarrollar intuición.
- [tiktoken](#) — la biblioteca de OpenAI. Vale la pena leer `tiktoken/_educational.py`.
- Sennrich, R., Haddow, B. y Birch, A. (2016). *Neural Machine Translation of Rare Words with Subword Units* — el paper que trajo BPE a NLP.
- Beta, R. *A Programmer's Introduction to Unicode* — para la parte de Unicode y UTF-8.
- [SentencePiece](#) — la alternativa que usan Llama, Mistral y Gemma.