Facsímil 08

La ciencia de los datos

Capítulo 04

Features y representaciones: de tablas a embeddings

686f6c61 · 2026

Capítulo 04: Features y representaciones: de tablas a embeddings

Entrando en el tema

En el capítulo anterior congelamos una partición honesta. Ahora toca una pregunta igual de importante: ¿qué entra realmente al modelo? Un dataset no se convierte solo en inteligencia por estar en una tabla. Hay que transformar fechas, categorías, texto y metadatos en números que una función pueda operar.

Ese paso se llama representación. A veces es tan sencillo como convertir `email` en una columna binaria. A veces implica normalizar una fecha. A veces convierte un texto completo en un vector de cientos o miles de dimensiones. En todos los casos hay una decisión de ingeniería: qué información permitimos, qué información prohibimos, cuándo se ajusta la transformación y qué versión queda registrada.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Distinguir dato crudo, feature y representación.	No llamas “dato” a cualquier columna sin mirar cómo se transforma.
Diseñar un contrato de features.	Declaras columnas permitidas, prohibidas, tipos y `fit_scope`.
Explicar one-hot, normalización, TF-IDF y embeddings.	Puedes escribir sus fórmulas y decir qué aprende cada pieza.
Evitar leakage de representación.	Ajustas vocabulario, escalas e IDF solo con train.
Leer una dimensión de embedding como coste y contrato.	No tratas `d=64`, `d=768` o `d=3072` como decoración.
Evaluar una búsqueda vectorial mínima.	Miras top-k, términos fuera de vocabulario, metadata y slices.
Detectar feature skew.	Preguntas si offline y online calculan la misma feature con el mismo tiempo de observación.
Diferenciar proxy de métrica formal.	No llamas recall@k a una cobertura top-k sin relevancia anotada.

La idea común detrás de todos esos objetivos es que representar no es comprimir por comodidad. Representar es decidir qué señales entran, qué señales quedan fuera, cómo se ajustan las transformaciones y qué contrato garantiza que producción verá lo mismo que vimos al evaluar. Una feature puede parecer una columna inocente y ser un atajo; un embedding puede parecer semántica y ser una cercanía sin evidencia.

La frase central del capítulo:

> Una feature no es una columna: es una promesa sobre cómo una columna se convierte en señal.

La escena: el modelo no sabe leer tu tabla

Imagina una tabla con casos de soporte académico. Tiene columnas como `created_at`, `product`, `channel`, `student_id`, `source_id`, `label` y `text`. Una persona puede leerla y entender bastante: esto va de becas, aquello de matrícula, esto llegó por email, aquello por portal.

El modelo, en cambio, no ve "becas" como idea ni "email" como canal humano. Ve números. Si le das texto, alguien debe tokenizarlo o vectorizarlo. Si le das fechas, alguien debe convertirlas en una escala útil. Si le das categorías, alguien debe decidir si se codifican como one-hot, índices, embeddings aprendidos o metadata para filtrar.

Aquí aparece una parte muy de ingeniería de datos e IA: no basta con tener datos. Hay que construir una representación coherente, reproducible y alineada con el uso real. Si esa representación se construye mal, el modelo puede aprender una señal falsa, olvidar una señal importante o medir con información que no debería haber visto.

Qué no es una feature

Una feature no es cualquier columna que tengamos a mano. `student_id` existe en la tabla, pero quizá no debería entrar al modelo. Si entra, el sistema podría aprender estilos de estudiantes concretos en vez de patrones generales. `source_id` existe, pero puede actuar como atajo si cada documento está demasiado asociado a una etiqueta. `label` existe, pero usarla como entrada sería convertir la respuesta en pista.

Tampoco es una transformación hecha "porque mejora la métrica". Si una feature mejora mucho el resultado, hay que preguntar por qué. Puede ser una señal legítima, como la antigüedad de un caso. Puede ser una fuga, como una fecha de resolución disponible solo después de cerrar el ticket. Puede ser una variable de identidad, como un código interno que en train coincide con la etiqueta pero en producción cambia.

Y un embedding no es conocimiento puro. Es una representación numérica aprendida o calculada. Puede capturar parecido semántico, pero no garantiza verdad, actualidad, permisos ni relación exacta. Si un vector store devuelve un fragmento cercano, todavía necesitamos comprobar si ese fragmento contiene evidencia suficiente para la pregunta.

Qué sí es una representación

Una representación es la forma en que describimos un objeto para que un algoritmo pueda trabajar con él. En aprendizaje estadístico, el punto de partida suele ser una matriz X, donde cada fila representa un ejemplo y cada columna representa una feature.[1] La etiqueta o valor que queremos predecir suele separarse como y.

$$X \in \mathbb{R}^{n \times d}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
X	Matriz de features.	24 casos por 49 features.
n	Número de ejemplos.	24 filas.
d	Número de features.	49 columnas generadas por el cuaderno.
$x_{i,j}$	Valor de la feature j para el ejemplo i.	`product__becas = 1`.

En palabras: X es la tabla ya convertida a números útiles para el modelo. No incluye todo lo que existe en el dataset; incluye solo las señales que el contrato permite usar como entrada.

En un proyecto de IA moderna, X puede mezclar varias familias de señal:

FAMILIA	EJEMPLO	QUÉ APORTA
Numérica	Días desde la primera fecha de train.	Orden temporal, recencia, duración.
Categórica	Producto, canal, país, tipo de usuario.	Segmentos y contexto operativo.
Texto disperso	TF-IDF, bolsa de palabras, BM25.	Coincidencia lexical y términos técnicos.
Texto denso	Embeddings de frases o documentos.	Parecido semántico y paráfrasis.
Metadata	`case_id`, permisos, versión de documento.	Trazabilidad, filtros y auditoría.

La clave es no mezclar funciones. Metadata puede ser imprescindible para auditar o filtrar, pero no necesariamente debe ser feature del modelo. El contrato decide esa frontera.

Familias de features: separar antes de transformar

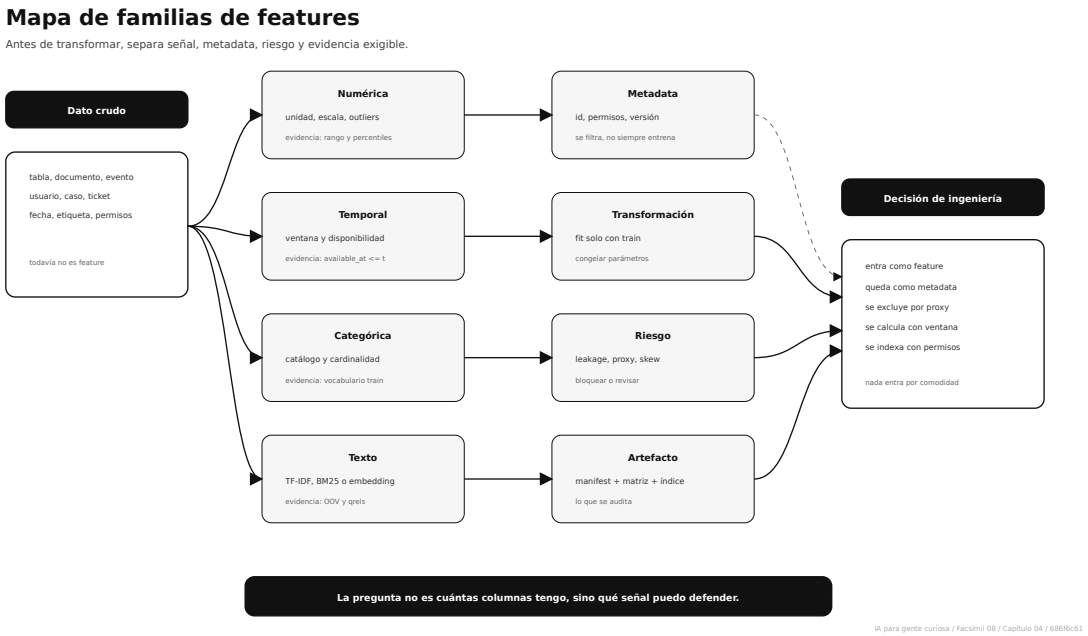
Una representación profesional empieza antes del código. Empieza clasificando qué tipo de señal tienes delante. No se transforma igual una fecha, una categoría con seis valores, una categoría con miles de valores, una descripción de texto, una etiqueta humana, un identificador técnico o una fecha de disponibilidad. Si todo entra en el mismo saco, el pipeline puede funcionar por accidente, pero será difícil de explicar, difícil de reproducir y peligroso de llevar a producción.

Para un ingeniero de datos, esta separación es una forma de proteger el sistema. Una feature numérica exige rango, unidad y política de nulos. Una feature temporal exige ventana y fecha de observación. Una feature categórica exige catálogo, tratamiento de desconocidos y decisión sobre cardinalidad. Un texto exige vocabulario, tokenización, idioma, normalización y quizá embeddings. La metadata exige otra pregunta: ¿sirve para filtrar y auditar, o realmente debe entrar como señal del modelo?

FAMILIA	EJEMPLO REALISTA	DECISIÓN TÉCNICA	RIESGO TÍPICO	EVIDENCIA QUE PEDIRÍA
Numérica	`importe_pendiente`, `dias_desde_ticket`	Escala, unidad, rango, outliers.	Que una magnitud grande domine sin aportar más información.	Histograma, percentiles, regla de nulos y transformación.
Temporal	`tickets_7d_as_of_created_at`	Ventana, granularidad y fecha de disponibilidad.	Usar datos que todavía no existían en el momento de decidir.	`event_timestamp`, `available_at` y test point-in-time.
Categórica baja	`channel`, `product`, `country`	One-hot, catálogo y desconocidos.	Crear columnas nuevas silenciosamente en producción.	Vocabulario ajustado con train y política `unknown`.
Categórica alta	`student_id`, `center_id`, `course_id`	Agrupar, excluir, usar metadata o embeddings aprendidos.	Memorizar identidades o crear miles de columnas poco útiles.	Cardinalidad, frecuencia mínima y revisión de proxy.
Texto lexical	`text`, `title`, `subject`	TF-IDF, BM25, stopwords, idioma y vocabulario.	Ajustar IDF con test o perder siglas y códigos.	Vocabulario train, OOV y baseline lexical.
Texto denso	Chunk, pregunta, caso completo	Encoder, dimensión, normalización, métrica.	Confundir cercanía semántica con evidencia válida.	Modelo, versión, dimensión, qrels y métricas de ranking.
Metadata	`case_id`, `source_id`, permisos, versión	Filtrar, auditar, unir y explicar.	Meter identificadores como atajo predictivo.	Lista de columnas prohibidas y uso explícito en filtros.

La alta cardinalidad merece atención propia. Un one-hot de `product` con seis categorías es razonable. Un one-hot de `student_id` con 200.000 valores casi nunca lo es: sube dimensión, memoria y riesgo de memorizar personas. Si el identificador es necesario para permisos, deduplicación o trazabilidad, se conserva como metadata. Si se quiere usar como señal, hay que justificar por qué no es un proxy peligroso y cómo se comporta cuando aparece una identidad nueva.

El *target encoding* (codificación por objetivo) es todavía más delicado. La idea es codificar una categoría usando una estadística del objetivo, por ejemplo la tasa histórica de reclamación de un centro. Puede ser útil en problemas tabulares, pero también puede filtrar la respuesta si se calcula con todo el dataset o si una categoría aparece muy pocas veces. La advertencia general coincide con la literatura sobre leakage en minería de datos: una transformación puede introducir información que no estaría disponible en el momento real de predicción.[2] Si se usa una codificación supervisada, debe calcularse dentro de cada fold o solo con train, suavizar categorías raras y documentar la política para categorías nuevas.



Separar familias de features evita que metadata, identidad, tiempo y texto acaben mezclados sin contrato.

Cómo se construye una feature por dentro

Construir una representación es una cadena de decisiones. Primero elegimos columnas permitidas. Después definimos transformaciones. Luego ajustamos lo que tenga que aprenderse usando solo train. Finalmente transformamos train, validation y test con los parámetros ya congelados.

Una feature numérica puede normalizarse con una escala min-max estándar. La idea es llevar un valor a una escala común usando los extremos observados en el conjunto con el que se ajusta la transformación. La documentación de `MinMaxScaler` expresa esta misma transformación para escalar cada feature dentro de un rango dado.[3] En un pipeline profesional, ese ajuste pertenece a train; scikit-learn insiste en esta separación porque ajustar preprocesado con datos reservados es una fuente clásica de fuga de información.[4]

$$\tilde{x} = \frac{x - x_{min}^{train}}{x_{max}^{train} - x_{min}^{train}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Valor original.	Día 10 desde el inicio.
x_{min}^{train}	Mínimo observado en train.	Día 0.
x_{max}^{train}	Máximo observado en train.	Día 15.
$\tilde{x}$	Valor normalizado.	$10/15 = 0.67$.

En palabras: el valor se reubica dentro de una escala común usando extremos aprendidos en train. Si validation o test participan en esos extremos, la evaluación deja de ser limpia.

La parte importante está en los superíndices: mínimo y máximo salen de train. Si usas todo el dataset, test participa en la escala. Puede parecer poco, pero en pipelines grandes esa pequeña comodidad se acumula con imputación, selección de variables, vocabularios e índices. Por eso las librerías maduras recomiendan encapsular preprocesado y modelo dentro de un `Pipeline`: no por estética, sino para que cada fold, split o experimento ajuste transformaciones en el sitio correcto.[5]

Una categoría puede codificarse con one-hot, una codificación que crea una columna binaria por categoría del vocabulario ajustado.[6]

$$onehot(c)_k = \begin{cases} 1 & \text{si } c = k \\ 0 & \text{si } c \neq k \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
c	Categoría de la fila.	`becas`.
k	Categoría concreta del vocabulario.	`matricula`.
$onehot(c)_k$	Columna binaria para k.	0 si el producto es `becas`.

En palabras: cada categoría congelada se convierte en una columna de sí/no. La categoría no se convierte en un número ordinal que invente distancia entre `becas` y `matricula`.

Si en train vemos `becas`, `horarios`, `matricula`, `pagos`, `practicas` y `titulos`, esas son las columnas que crea el contrato. Si mañana aparece `doctorado`, el sistema no debería improvisar silenciosamente una columna nueva en producción. Debe registrarlo como categoría desconocida, decidir cómo tratarla y actualizar el contrato si procede.

Para texto lexical, TF-IDF pondera términos usando frecuencia local y rareza global. Es una representación clásica en recuperación de información y clasificación textual; scikit-learn la implementa como transformación de recuentos a una matriz de features TF-IDF, con

referencias a Manning, Raghavan y Schütze.[7] Una forma habitual es:

$$tfidf(t,d) = tf(t,d) \cdot \left(\log \frac{1+N}{1+df(t)} + 1\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t	Término.	`beca` .
d	Documento o texto de una fila.	“requisitos de beca”.
$tf(t,d)$	Veces que aparece t en d.	1.
N	Número de textos de train.	15.
$df(t)$	Número de textos de train que contienen t.	2.
$tfidf(t,d)$	Peso final del término.	Mayor si el término es informativo.

En palabras: TF-IDF sube términos frecuentes dentro de un texto, pero baja términos que aparecen en muchos textos. Por eso ayuda a rescatar palabras específicas sin confundirlas con vocabulario común del corpus.

BM25, una familia clásica de ranking lexical, parte de una intuición parecida pero ajusta saturación de frecuencia y longitud del documento.[8] En una formulación habitual, la puntuación de un documento D para una consulta Q se expresa así:

$$\text{BM25}(D,Q) = \sum_{q_i \in Q} IDF(q_i) \frac{f(q_i,D)(k_1+1)}{f(q_i,D) + k_1\left(1 - b + b\frac{|D|}{avgdl}\right)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q_i	Término de la consulta.	`matricula`.
$f(q_i, D)$	Frecuencia del término en el documento.	Cuántas veces aparece `matricula`.
$IDF(q_i)$	Peso de rareza global del término.	Sube si aparece en pocos documentos.
$	D	$
$avgdl$	Longitud media del corpus.	Media de tokens por documento.
k_1, b	Parámetros de saturación y normalización por longitud.	Controlan cuánto premia repetición y documento largo.

En palabras: BM25 premia que el documento contenga términos de la consulta, pero no deja que repetir una palabra cien veces dispare la puntuación sin límite. También corrige el efecto de documentos más largos.

En RAG híbrido, estas señales lexicales siguen siendo muy útiles: los embeddings toleran paráfrasis; lo lexical rescata términos exactos, códigos, siglas y nombres propios. Si un alumno pregunta por `EXP-2026-41`, un embedding puede encontrar documentos parecidos, pero BM25 suele ser la primera defensa para no perder el identificador literal.

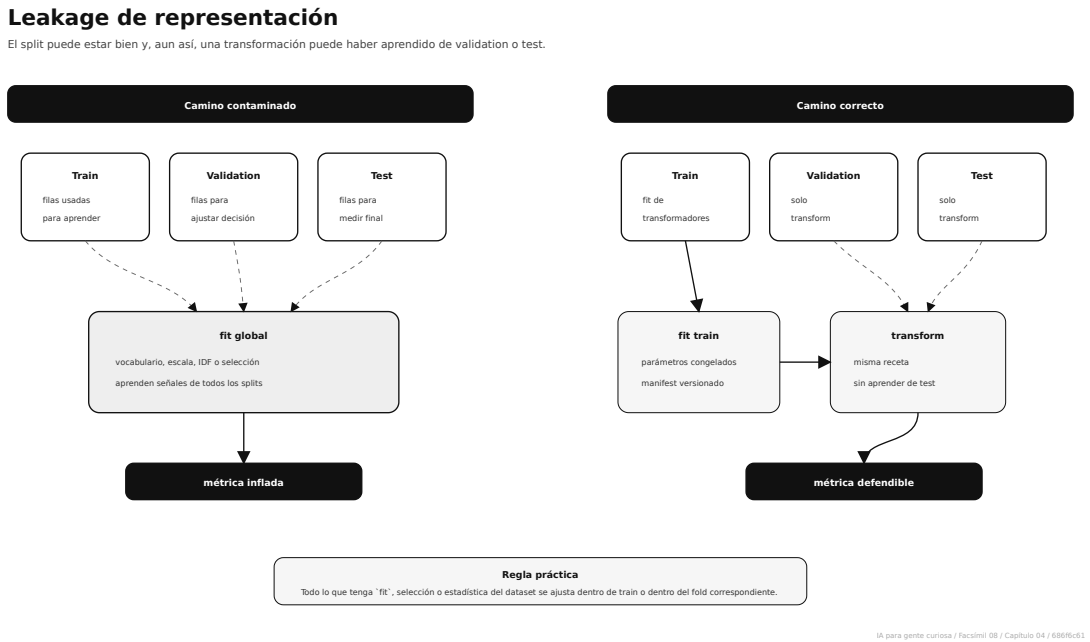
Leakage de representación: cuando la fuga no está en el split

En el capítulo 03 hablamos de splits y leakage. Aquí aparece una versión más sutil: el split puede estar bien, pero la transformación puede haber mirado donde no debía. Si calculas el vocabulario TF-IDF con todos los textos, validation y test ya influyeron en los pesos. Si eliges las mejores features mirando la métrica sobre todo el dataset, test ya participó en el diseño. Si normalizas con mínimos y máximos globales, los extremos de test modifican la escala de train. La fuga no siempre está en una columna llamada `label`; a veces está en una decisión cómoda de preprocesado.

Kaufman, Rosset, Perlich y Stitelman describen el leakage como información que no estaría disponible en el momento real de predicción y que, aun así, entra durante entrenamiento o evaluación.[9] En representación, esa información puede colarse por tres puertas: columnas posteriores al evento, transformaciones ajustadas fuera de train y selección de features hecha con el conjunto reservado.

Un caso muy realista: quieres predecir si una consulta de matrícula será prioritaria. Tienes `created_at`, `text`, `channel`, `status`, `resolved_at`, `agent_notes` y `label`. `resolved_at` y `agent_notes` son tentadores porque correlacionan mucho con el resultado. Pero si se rellenan

después de atender el caso, no son señales disponibles al decidir. Otro caso: haces target encoding de `center_id` usando toda la tabla. Si un centro aparece solo en test, acabas usando su tasa real de test para codificarlo. La métrica sube; el sistema no ha aprendido mejor, solo ha recibido una pista.



> El leakage de representación aparece cuando un transformador aprende de filas que deberían servir solo para validar o medir.

Selección y reducción de features: menos columnas puede ser mejor ingeniería

Añadir features no es gratis. Aumenta memoria, tiempo de entrenamiento, riesgo de overfitting, coste de serving, dificultad de explicación y superficie de errores. En datos tabulares pequeños, diez features bien entendidas pueden ser más defendibles que quinientas columnas generadas sin criterio. En texto y embeddings, lo mismo: subir dimensión o vocabulario puede mejorar señales, pero también introduce ruido, coste y dificultad de evaluación.

La selección de features busca responder una pregunta concreta: ¿qué variables aportan señal fuera de la casualidad y del atajo? Hay técnicas filtradas, como mirar varianza, correlación o información mutua; técnicas embebidas, como regularización; técnicas wrapper, como evaluar subconjuntos; y técnicas de inspección posterior, como permutation importance. scikit-learn documenta permutation importance como una forma de medir cuánto empeora una métrica al permutar una feature, siempre sobre un conjunto de validación o test adecuado, no sobre los mismos datos usados para ajustar.[10]

La reducción de dimensionalidad responde a otro problema: representar muchos valores en menos coordenadas conservando estructura útil. PCA, SVD y otras técnicas aparecen en textos clásicos de aprendizaje estadístico como herramientas para proyectar datos a espacios de menor dimensión.[11] En este capítulo no necesitamos convertirlo en un curso de álgebra lineal, pero sí entender su contrato: si la proyección se ajusta con train, validation y test se transforman con esa proyección congelada. Si la ajustas con todo, vuelves a contaminar.

DECISIÓN	SIRVE PARA	CUIDADO PRINCIPAL	EJEMPLO ÚTIL
Eliminar varianza cero	Quitar columnas constantes.	Puede cambiar con producción si el catálogo crece.	Una categoría que nunca aparece en train.
Revisar correlación	Detectar duplicación fuerte entre features.	Correlación no implica causalidad ni utilidad.	`dias_abierto` y `horas_abierto`.
Permutation importance	Medir impacto de una feature en una métrica.	Debe hacerse con validación honesta.	Ver si `source_id` domina por atajo.
Regularización	Penalizar complejidad en modelos lineales.	Depende de escala y del modelo.	Muchas variables dispersas de texto.
PCA/SVD	Reducir dimensión conservando estructura.	Menos explicación directa por columna.	Vectores dispersos de alta dimensión.
Filtro por dominio	Bloquear proxies o variables post-evento.	Exige criterio humano y documentación.	Excluir `resolved_at` de un modelo de priorización.

La regla que me gusta usar es sencilla: si una feature no se puede explicar, versionar, calcular en el momento correcto y medir en validación honesta, todavía no está lista para producción. Puede quedarse en exploración, pero no en el contrato publicable.

Una selección de features bien hecha no busca adelgazar por estética. Busca reducir atajos, coste y fragilidad. Si una columna domina la predicción, el siguiente paso no es celebrarlo: es preguntarse si esa columna existirá en producción, si estaba disponible en el momento correcto, si es un proxy delicado y si la métrica sigue aguantando cuando la revisas por slices. Esa conversación es menos vistosa que subir un punto de accuracy, pero es mucho más cercana a cómo se evita deuda técnica en IA.

Embeddings: vectores densos con contrato

Un embedding es un vector denso. La notación siguiente es la forma habitual de escribir que un encoder parametrizado transforma un objeto en un vector dentro de un espacio de dimensión m, idea que aparece en modelos neuronales de lenguaje, Word2Vec, BERT y

Sentence-BERT.[12][13]

$$e = f_\theta(objeto) \in \mathbb{R}^m$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$objeto$	Lo que queremos representar.	Texto de un caso, imagen, usuario, producto.
f_θ	Encoder o función de representación.	Modelo de embeddings o encoder local.
θ	Parámetros del encoder.	Pesos entrenados o reglas congeladas.
m	Dimensión del vector.	64 en el cuaderno; 768 en muchos modelos BERT.
e	Vector resultante.	`[0.0, -0.12, ...]`.

En palabras: el encoder transforma un objeto en coordenadas. Esas coordenadas solo son comparables si query, documentos y evaluación usan el mismo encoder, versión, dimensión y normalización.

Para un ingeniero, la dimensión m no es un número decorativo. Afecta memoria, coste de indexación, latencia de búsqueda, tamaño del índice, elección de base vectorial y facilidad para mantener varias versiones. También afecta a la evaluación: cambiar de encoder o de dimensión puede reordenar vecinos aunque el corpus sea el mismo. Por eso un embedding debe venir con ficha técnica igual que un dataset: modelo, versión, dimensión, normalización, métrica de similitud, fecha de corte y uso permitido.

La historia moderna de embeddings no empieza con los LLM actuales. Bengio y colaboradores ya propusieron representar palabras mediante vectores aprendidos dentro de modelos de lenguaje neuronales.[14] Word2Vec popularizó embeddings preentrenados eficientes para palabras.[15] BERT llevó la idea a representaciones contextuales: el vector de una palabra depende de la frase completa.[16] Sentence-BERT adaptó arquitecturas BERT para producir embeddings de frases útiles en similitud semántica.[17]

La similitud coseno compara direcciones y es una medida clásica en el modelo vectorial de recuperación de información.[18]

$$\cos(a, b) = \frac{a \cdot b}{\|a\|\|b\|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a, b	Dos vectores.	Query y caso indexado.
$a \cdot b$	Producto escalar.	Suma de productos dimensión a dimensión.
$\|a\|$	Norma de a.	Longitud del vector.
$\cos(a, b)$	Similitud por dirección.	1 si apuntan igual; 0 si son ortogonales.

En palabras: el coseno compara orientación más que tamaño. Dos vectores pueden ser cercanos si apuntan en dirección parecida, aunque sus magnitudes originales fueran distintas.

Si el sistema normaliza los embeddings a norma 1 antes de guardarlos, el cálculo se simplifica: comparar por coseno queda muy cerca de comparar por producto escalar. La normalización L2 es una operación estándar de preprocesado vectorial.[19] Se escribe así:

$$\hat{e} = \frac{e}{\|e\|_2}$$

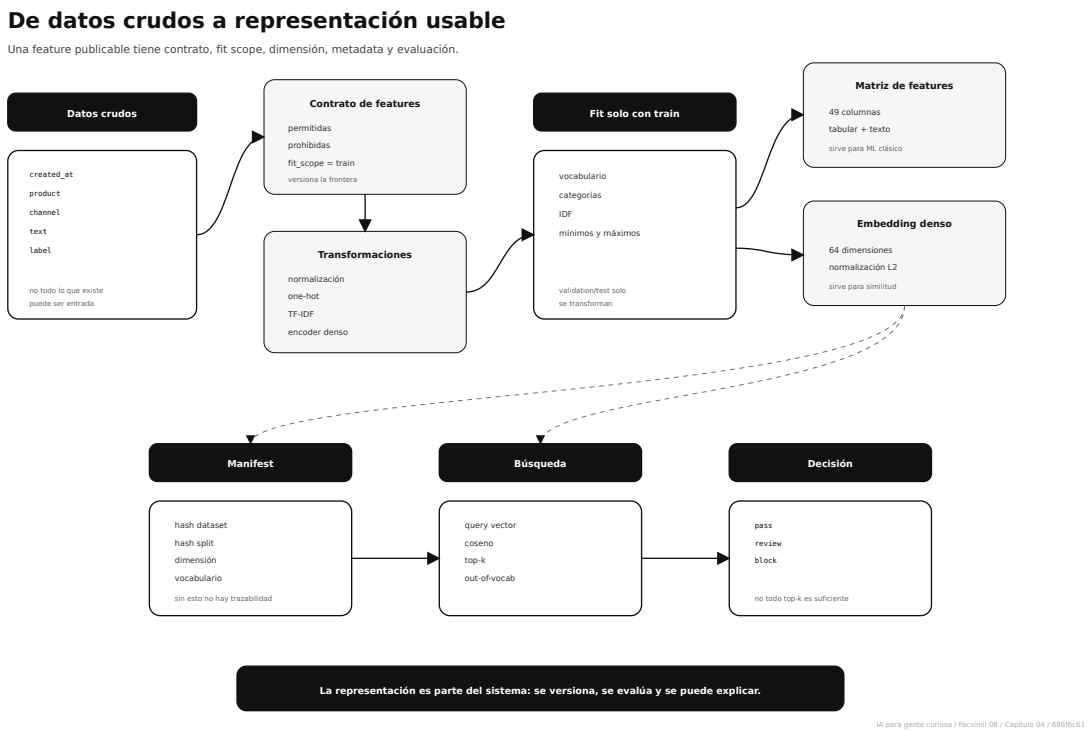
SÍMBOLO	SIGNIFICADO	EJEMPLO
e	Vector original.	Embedding antes de indexar.
$\|e\|_2$	Norma euclídea del vector.	Longitud geométrica.
$\hat{e}$	Vector normalizado.	El que se guarda si el índice espera norma 1.

En palabras: normalizar divide el vector por su longitud para dejarlo con norma 1. No añade conocimiento; prepara la geometría para comparar de forma coherente.

Esta fórmula no dice que el embedding sea “mejor”. Dice que lo dejas en una geometría consistente para que la comparación sea estable. En producción conviene registrar si el índice usa coseno, producto escalar o distancia euclídea, porque cambiar la métrica sin reindexar y reevaluar puede alterar el ranking aunque el texto sea el mismo.

Dimensión no significa “inteligencia”. Significa longitud del vector y, por tanto, memoria, coste de cálculo, coste de índice y capacidad de representar matices. Un embedding de 64 dimensiones cabe barato y sirve como baseline local. Uno de 768 o más puede capturar mucha más estructura, pero cuesta más guardar, comparar y servir.

La anatomía de una representación publicable



Una representación publicable conecta contrato, transformación, fit con train, manifiesto y evaluación.

Evaluar representaciones de recuperación

Una búsqueda vectorial no se valida mirando si “parece razonable”. Necesitas consultas, documentos relevantes anotados y una métrica que responda a la decisión real. En recuperación de información, si $Rel(q)$ es el conjunto de documentos relevantes para la consulta q, y $Ret_k(q)$ son los k documentos recuperados por el sistema, dos métricas básicas son precision@k y recall@k.[20]

$$\text{Precision@k}(q) = \frac{|Rel(q) \cap Ret_k(q)|}{k}$$

$$\text{Recall@k}(q) = \frac{|Rel(q) \cap Ret_k(q)|}{|Rel(q)|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Consulta evaluada.	“justificante de pago pendiente”.
$Rel(q)$	Conjunto anotado de documentos relevantes.	Casos o chunks que deberían responder a esa consulta.
$Ret_k(q)$	Resultados devueltos entre las primeras k posiciones.	Top 3 del índice.
Precision@k	Qué proporción del top-k es relevante.	Evita llenar contexto de ruido.
Recall@k	Qué proporción de lo relevante apareció en top-k.	Evita dejar fuera evidencia necesaria.

En palabras: precision@k mira la limpieza de los primeros resultados; recall@k mira cuánta evidencia relevante has conseguido traer. Si $k = 3$, recuperas tres documentos y solo uno está anotado como relevante, precision@3 vale $1/3$. Si para esa consulta había cuatro documentos relevantes y solo recuperaste uno, recall@3 vale $1/4$. No es una métrica decorativa: en RAG, esos tres documentos suelen ser los que pasan al prompt, al reranker o a la revisión humana.

Pero precision y recall no dicen dónde aparece el primer resultado bueno. Para eso se usa MRR, *Mean Reciprocal Rank*, una métrica clásica en recuperación de información cuando importa mucho que el primer resultado relevante aparezca pronto.[21]

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Q	Conjunto de consultas evaluadas.	30 preguntas reales de soporte.
q	Una consulta concreta.	“justificante de pago pendiente”.
$rank_q$	Posición del primer resultado relevante para q.	1 si el primero ya vale; 3 si aparece tercero.
MRR	Media del recíproco de esas posiciones.	Penaliza que lo relevante aparezca tarde.

En palabras: si el primer resultado relevante aparece en posición 1, aporta 1. Si aparece en posición 2, aporta 0,5. Si aparece en posición 5, aporta 0,2. Es muy útil para buscadores internos, asistentes de soporte y sistemas donde el usuario o el modelo suelen mirar pocos resultados.

Cuando no todas las evidencias relevantes valen lo mismo, necesitamos grados. Un fragmento puede responder exactamente, otro puede ser parcialmente útil y otro puede estar relacionado pero no resolver la pregunta. nDCG@k, propuesto dentro de la evaluación basada en ganancia acumulada por Järvelin y Kekäläinen, mide ranking con relevancia graduada y descuenta posiciones bajas.[22]

$$\mathrm{DCG@k}(q) = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$$\mathrm{nDCG@k}(q) = \frac{\mathrm{DCG@k}(q)}{\mathrm{IDCG@k}(q)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
i	Posición del resultado dentro del ranking.	1, 2, 3...
rel_i	Grado de relevancia del resultado en posición i.	0 irrelevante, 1 parcial, 2 relevante, 3 excelente.
$\mathrm{DCG@k}$	Ganancia acumulada descontada hasta k.	Premia relevancia arriba.
$\mathrm{IDCG@k}$	Ganancia ideal si ordenaras perfecto.	El mejor ranking posible con esos juicios.
$\mathrm{nDCG@k}$	DCG normalizado entre 0 y 1.	Permite comparar consultas con distinta dificultad.

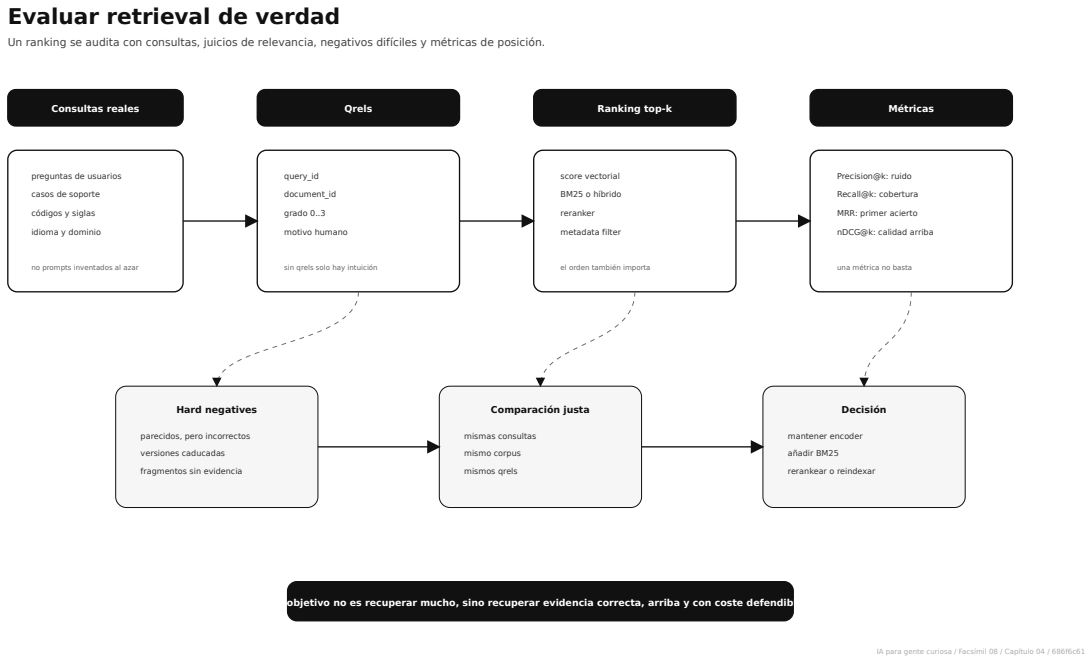
En palabras: nDCG@k castiga dos cosas a la vez: traer documentos poco relevantes y poner documentos buenos demasiado abajo. Para RAG es una métrica especialmente útil porque no basta con “algún chunk relevante en top-k”: quieres que los chunks más fuertes aparezcan arriba, antes de llenar contexto con ruido.

La diferencia no es académica de salón. Si haces un asistente de normativa universitaria, precision@k bajo significa que metes fragmentos irrelevantes en el prompt y aumentas el riesgo de respuesta confusa. Recall@k bajo significa que existe evidencia importante, pero el sistema no la trae. En un RAG de soporte, ambos errores se sienten: o respondes con ruido, o respondes sin la pieza que hacía falta.

El cuaderno de este capítulo trae un `qrels` mínimo, no una evaluación industrial completa. Eso significa que ya no miramos solo si el producto esperado aparece en el top-k: anotamos qué casos son relevantes para cada consulta, con qué grado y por qué. La cobertura por producto sigue siendo una señal barata para detectar errores groseros, pero precision@k, recall@k, MRR y nDCG@k se calculan con relevancia anotada. Para cerrar un sistema real

habría que ampliar ese `qrels` con más consultas, doble revisión, desacuerdos entre revisores, *hard negatives* y slices de dominio. Aun así, el mecanismo ya permite comparar encoder local, BM25, búsqueda híbrida, reranker y cambios de chunking sin discutir a ojo.

Un *hard negative* es un documento que se parece mucho a la respuesta correcta, pero no sirve para contestar. Por ejemplo, una política de becas del curso anterior puede ser semánticamente cercana a la consulta actual, pero estar caducada. Si el sistema la recupera arriba, el embedding no está “fallando de forma absurda”; está mostrando que la evaluación necesita distinguir parecido de evidencia válida. En sistemas de IA aplicada, esos casos son oro: obligan a introducir fecha, versión, permisos, reranking o reglas de filtrado.



> Una evaluación útil combina consultas reales, qrels, negativos difíciles y métricas que miran limpieza, cobertura y posición.

De quién viene cada fórmula y por qué está aquí

Hay una costumbre peligrosa en libros técnicos: poner fórmulas para que el texto parezca más serio. Aquí queremos justo lo contrario. Una fórmula entra solo si ayuda a tomar una decisión de ingeniería: cómo representar, cómo evitar leakage, cómo indexar, cómo comparar o cómo medir.

También conviene ser honestos con la palabra “de quién”. Algunas fórmulas tienen autores o familias muy reconocibles, como BM25. Otras no pertenecen a una persona concreta, sino a una tradición matemática o de ingeniería muy asentada, como escribir una matriz de datos

como $X \in \mathbb{R}^{n \times d}$ o normalizar un vector por su norma. En esos casos no buscamos un “inventor” decorativo: buscamos la referencia académica o técnica que justifica su uso.

FÓRMULA	DE DÓNDE VIENE	POR QUÉ SE USA AQUÍ	DECISIÓN DE INGENIERÍA QUE PERMITE
$X \in \mathbb{R}^{n \times d}$	Notación estándar de aprendizaje estadístico; Hastie, Tibshirani y Friedman la usan para razonar sobre matrices de datos, features y modelos.	Sitúa el capítulo: una tabla se convierte en matriz y cada columna pasa a tener significado operacional.	Separar filas, features, target y metadata antes de entrenar o evaluar.
Escalado min-max	Transformación clásica de preprocesado; scikit-learn la documenta en `MinMaxScaler`.	Explica por qué una feature numérica no debería entrar “tal cual” si su escala domina a otras.	Ajustar mínimos y máximos solo con train y evitar que test participe en el preprocesado.
One-hot encoding	Codificación categórica estándar; scikit-learn la documenta en `OneHotEncoder`.	Enseña que una categoría no es un número ordinal salvo que el dominio lo justifique.	Congelar vocabularios de categorías y decidir qué hacer con valores nuevos en producción.
TF-IDF	Recuperación de información y vectorización textual; Manning, Raghavan y Schütze la explican, y scikit-learn la implementa en `TfidfTransformer`.	Muestra cómo un texto puede pasar a features dispersas sin usar todavía un embedding neural.	Ajustar vocabulario e IDF solo con train; detectar términos fuera de vocabulario.
BM25	Robertson y Zaragoza, dentro del marco probabilístico de recuperación de información.	Da una base lexical fuerte para búsqueda y RAG híbrido, especialmente con códigos, siglas y nombres propios.	Comparar búsqueda lexical, vectorial e híbrida antes de elegir un índice.
$e = f_\theta(objeto) \in \mathbb{R}^m$	Notación habitual para encoders parametrizados; conecta con modelos neuronales de lenguaje, Word2Vec, BERT y Sentence-BERT.	Evita tratar “embedding” como magia: es una función que transforma un objeto en un vector.	Registrar modelo, versión, dimensión, normalización y tipo de objeto embebido.
Similitud coseno	Modelo vectorial de recuperación de información; Manning, Raghavan y Schütze la presentan como forma de comparar vectores.	Permite explicar por qué dos textos pueden estar cerca por dirección aunque no tengan la misma longitud.	Elegir métrica de índice y comprobar si query y documentos usan el mismo espacio vectorial.
Normalización L2	Operación estándar de álgebra lineal y preprocesado vectorial; scikit-learn la documenta en `Normalizer`.	Explica por qué muchos sistemas guardan embeddings con norma 1 antes de comparar.	Saber cuándo coseno y producto escalar son comparables y cuándo reindexar al cambiar métrica.

FÓRMULA	DE DÓNDE VIENE	POR QUÉ SE USA AQUÍ	DECISIÓN DE INGENIERÍA QUE PERMITE
$\mathrm{Precision@k}$	Métrica clásica de recuperación de información; Manning, Raghavan y Schütze.	Mide cuánto ruido metes en los primeros resultados que luego verá el modelo o el usuario.	Decidir si el top-k tiene suficiente limpieza para alimentar un RAG.
$\mathrm{Recall@k}$	Métrica clásica de recuperación de información; Manning, Raghavan y Schütze.	Mide si el sistema trae suficiente evidencia relevante entre los primeros k resultados.	Decidir si hace falta cambiar encoder, chunking, BM25, reranker o anotación de relevancia.
MRR	Métrica clásica de ranking usada en recuperación de información; Manning, Raghavan y Schütze la tratan dentro de evaluación de sistemas IR.	Mide cuánto tarda en aparecer el primer resultado relevante.	Decidir si el sistema sirve cuando el usuario o el LLM mira solo los primeros resultados.
$\mathrm{nDCG@k}$	Evaluación por ganancia acumulada descontada; Järvelin y Kekäläinen la formalizan para relevancia graduada.	Mide si los documentos más útiles aparecen arriba, no solo dentro del top-k.	Comparar encoders, BM25, híbridos, rerankers o cuantización cuando hay grados de relevancia.

La lectura práctica es esta: las fórmulas no están para memorizar símbolos. Están para que, cuando alguien proponga “metamos embeddings y ya”, puedas preguntar con precisión: ¿qué matriz estamos creando?, ¿qué se ajustó con train?, ¿qué encoder produjo el vector?, ¿qué dimensión tiene?, ¿qué métrica usa el índice?, ¿qué documentos eran relevantes y qué parte del top-k los recuperó?

Herramientas externas sin perder criterio

La forma superficial de leer este tema es quedarse con una frase tipo “usa un vector database” o “usa embeddings”. La forma de ingeniería es separar capas. Primero está el **preprocesado reproducible**: qué columnas entran, cómo se transforman y dónde se ajustan los parámetros. Después está el **contrato de datos**: qué tipo, rango, null, categoría o formato se acepta. Luego aparece la **capa de features** si hay que reutilizar definiciones offline y online. Y, si hay texto, documentos o chunks, aparece la **capa de embeddings e índice vectorial**. Cada herramienta externa debería entrar en una de esas capas, no en todas a la vez.

Para tabular clásico, scikit-learn sigue siendo una base muy buena porque obliga a pensar en `Pipeline` y transformadores por columna. `ColumnTransformer`, por ejemplo, permite aplicar normalización a numéricas, one-hot a categóricas y TF-IDF a texto, concatenando después todo en una sola matriz de features.[23] La pregunta de ingeniería no es “¿uso scikit-learn?”, sino “¿mi preprocesado queda dentro del pipeline o está repartido en celdas de notebook que nadie podrá reproducir?”.

Para contratos de datos, herramientas como Pandera o Great Expectations ayudan a declarar expectativas sobre tablas: tipos, columnas requeridas, rangos, nulls, categorías o reglas de negocio.[24][25] No sustituyen entender el dominio. Si declaras mal el contrato, la herramienta validará basura perfectamente. Su valor está en convertir una conversación difusa en una barrera ejecutable: “esta feature no entra si falta fecha de observación”, “esta categoría nueva bloquea”, “este porcentaje de nulos exige revisión”.

Cuando una feature debe existir tanto en entrenamiento como en producción, el problema cambia. Ya no basta con un pipeline local: necesitas una disciplina de feature store. Feast y Tecton son dos referencias útiles para entender esa arquitectura: definiciones versionadas, fuentes offline, serving online, entidades, ventanas temporales y joins point-in-time.[26][27] En un proyecto pequeño quizá no necesitas desplegar una plataforma completa; pero sí necesitas copiar la disciplina: definición única, timestamp de disponibilidad, manifest, tests y comparación offline/online.

En entornos cloud, la misma idea aparece integrada en plataformas de datos. Databricks documenta feature tables gobernadas en Unity Catalog, con claves primarias, namespace de catálogo, esquema y tabla; Google describe Vertex AI Feature Store como una capa de metadata y serving online sobre fuentes como BigQuery.[28][29] La decisión aquí no es “montar cloud porque sí”. Es preguntar si tu equipo ya vive en un lakehouse, warehouse o plataforma gestionada, y si te conviene aprovechar permisos, linaje, catálogo y serving que ya existen.

Con embeddings ocurre otra tentación: elegir proveedor por moda. OpenAI documenta los embeddings como vectores de números reales donde la distancia entre vectores mide relación entre textos, y permite trabajar con dimensión como parámetro en modelos compatibles.[30] Eso no te libera de evaluar. Un embedding más grande puede mejorar algunas tareas, pero también sube coste de almacenamiento, latencia y tamaño del índice. En ingeniería se registra siempre: modelo, versión, dimensión, normalización, fecha de generación, corpus usado, idioma, política de reindexado y métrica de evaluación.

Para guardar y consultar vectores, la elección depende más de tu sistema que del brillo de la herramienta. Si ya tienes Postgres y el caso es pequeño o mediano, `pgvector` puede ser suficiente porque guarda vectores junto con datos relacionales, joins, backups y permisos existentes.[31] Si necesitas un motor especializado con colecciones, payloads, filtros, cuantización o más control de búsqueda, Qdrant, Weaviate, Milvus o Pinecone entran en la conversación.[32][33][34][35]

La comparación útil no es “cuál es mejor”, sino “qué decisión protege”. Qdrant habla en términos de colecciones, vectores y payloads; eso te obliga a pensar en dimensión, métrica y metadata. Weaviate hace explícita la búsqueda híbrida, mezclando keyword y vector, útil cuando necesitas que siglas, códigos y nombres propios no desaparezcan detrás de la semántica. Milvus pone el foco en índices, métricas y balance entre rendimiento y corrección de la búsqueda. Pinecone insiste en metadata y filtrado porque muchos sistemas reales no pueden buscar “en todo”: tienen permisos, tenant, fecha, producto, país o tipo documental.

CAPA	HERRAMIENTAS POSIBLES	ÚSALAS CUANDO...	QUÉ EXIGIR ANTES DE CONFIAR
Preprocesado reproducible	scikit-learn `Pipeline`, `ColumnTransformer`	Tienes numéricas, categorías y texto en una misma matriz.	El `fit` ocurre solo con train y el pipeline se serializa o versiona.
Contrato de datos	Pandera, Great Expectations	Necesitas bloquear columnas, tipos, nulls, rangos o categorías inválidas.	Las reglas vienen del dominio y fallan con ejemplos negativos reales.
Feature store	Feast, Tecton, Databricks Feature Engineering, Vertex AI Feature Store	La misma feature vive offline y online, o necesitas joins temporales.	Hay `event_timestamp`, `created_timestamp`, point-in-time join y comparación offline/online.
Embeddings	OpenAI, Voyage, Cohere, Gemini, modelos open weights	El objeto es texto, chunk, imagen o consulta semántica.	Quedan registrados modelo, dimensión, normalización, coste, idioma, corpus y fecha de generación.
Índice vectorial	pgvector, Qdrant, Weaviate, Milvus, Pinecone, FAISS	Necesitas top-k, filtros, metadata, reindexado o búsqueda aproximada.	Se mide precision@k, recall@k, MRR y nDCG@k con qrels, y se prueban filtros de permisos.
Híbrido y reranking	BM25 + vector, Weaviate hybrid, rerankers externos	Hay códigos, siglas, nombres propios o consultas largas.	Se compara contra baseline lexical, vectorial e híbrido con las mismas consultas.

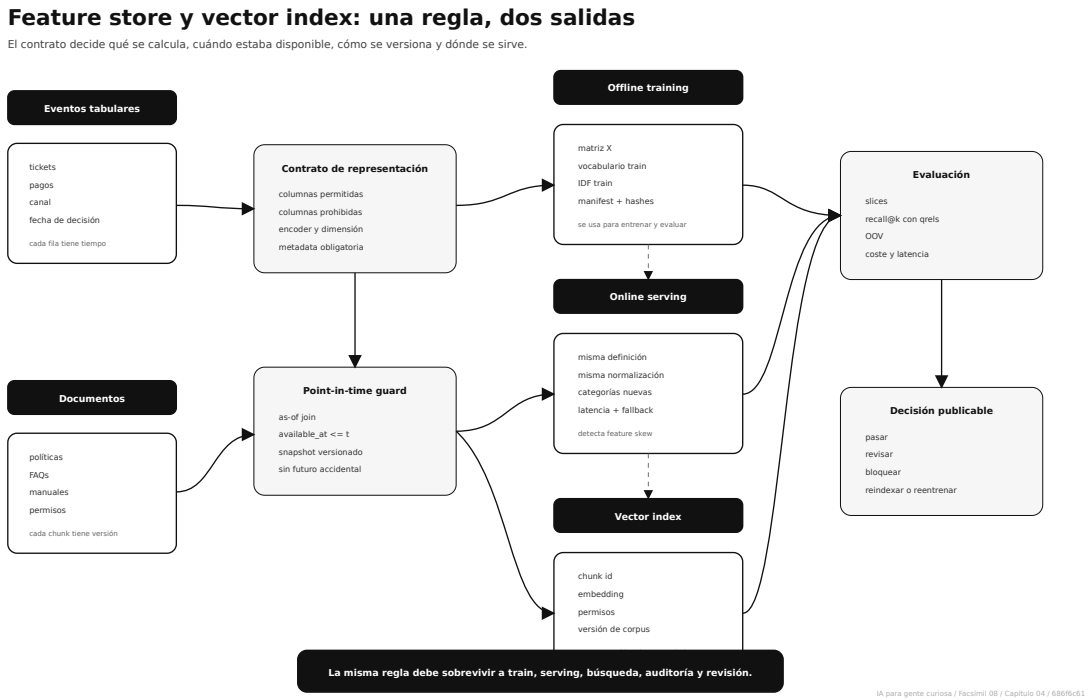
Un ejemplo concreto: tienes un asistente para normativa universitaria. Para la parte tabular de tickets usaría `Pipeline` y contratos de datos. Para features de producción, antes de montar una plataforma, exigiría una tabla con `entity_id`, `event_timestamp`, `available_at` y definición versionada. Para documentos, generaría embeddings con manifest, guardaría `document_id`, `chunk_id`, `version`, `permissions`, `created_at` y `embedding_model`, y empezaría con pgvector si el volumen cabe en Postgres. Solo saltaría a una base vectorial dedicada si necesito filtros complejos a escala, multi-tenant, cuantización, throughput alto o administración específica de índices. Y antes de celebrar nada, construiría un qrels de 30-50 consultas reales: ahí se ve si el sistema recupera evidencia o solo devuelve vecinos bonitos.

Paridad offline/online: el enemigo silencioso

Una de las formas más serias de romper un sistema de features es entrenar con una receta y servir con otra. En entrenamiento calculas `casos_abiertos_ultimos_7_dias` con una query sobre un snapshot histórico; en producción la calculas con una API que tiene retraso, redondea fechas de otra manera o incluye estados que el entrenamiento no veía. La métrica offline puede ser buena y, aun así, el sistema en vivo comportarse distinto. A esta diferencia se la suele llamar *training-serving skew* o *feature skew*.

El otro problema es temporal. Una feature puede ser correcta como cálculo y ser incorrecta como evidencia. Si un ticket se decide el 10 de junio, no puedes usar una columna que se rellenó el 12 de junio. Para evitarlo se usa *point-in-time correctness*: cada fila de entrenamiento solo puede unirse con valores de feature disponibles en el momento de decisión de esa fila. Feast lo documenta como *point-in-time joins*; Tecton lo trata como parte de la construcción de datos de entrenamiento para evitar que features futuras contaminen el dataset.[3637]

Un ejemplo cercano: quieres priorizar incidencias académicas. Podrías crear una feature `pagos_pendientes_al_crear_ticket`. Si la calculas mirando la base de pagos actual, quizá estás usando pagos resueltos después del ticket. La versión correcta necesita una fecha de observación: “qué sabía el sistema cuando se creó el ticket”. Esa frase parece burocrática, pero separa un modelo honesto de un modelo que aprende el futuro.



Un contrato de representación no termina en una matriz: también gobierna serving online, índice vectorial, evaluación y decisión de publicación.

Anatomía de un índice vectorial: no es una carpeta de embeddings

Un índice vectorial es una pieza de ingeniería, no un disco donde tiramos arrays. Recibe vectores, metadata, una métrica de distancia y una política de búsqueda. Devuelve vecinos aproximados o exactos, normalmente con filtros de payload, tenant, permisos o fecha. Si lo usamos en RAG, ese ranking puede decidir qué evidencia entra en el prompt. Por eso la pregunta correcta no es “¿qué vector database está de moda?”, sino “¿qué contrato de recuperación puedo medir y operar?”.

En volúmenes pequeños, una búsqueda exacta puede comparar la consulta contra todos los vectores. En volúmenes grandes, eso se vuelve caro. FAISS popularizó infraestructuras de búsqueda de similitud a gran escala, especialmente en GPU, y HNSW se convirtió en una familia muy usada de índices aproximados basados en grafos jerárquicos.[38][39] Aproximado significa esto: a cambio de velocidad y memoria razonable, mides si sigues recuperando los vecinos que importan.

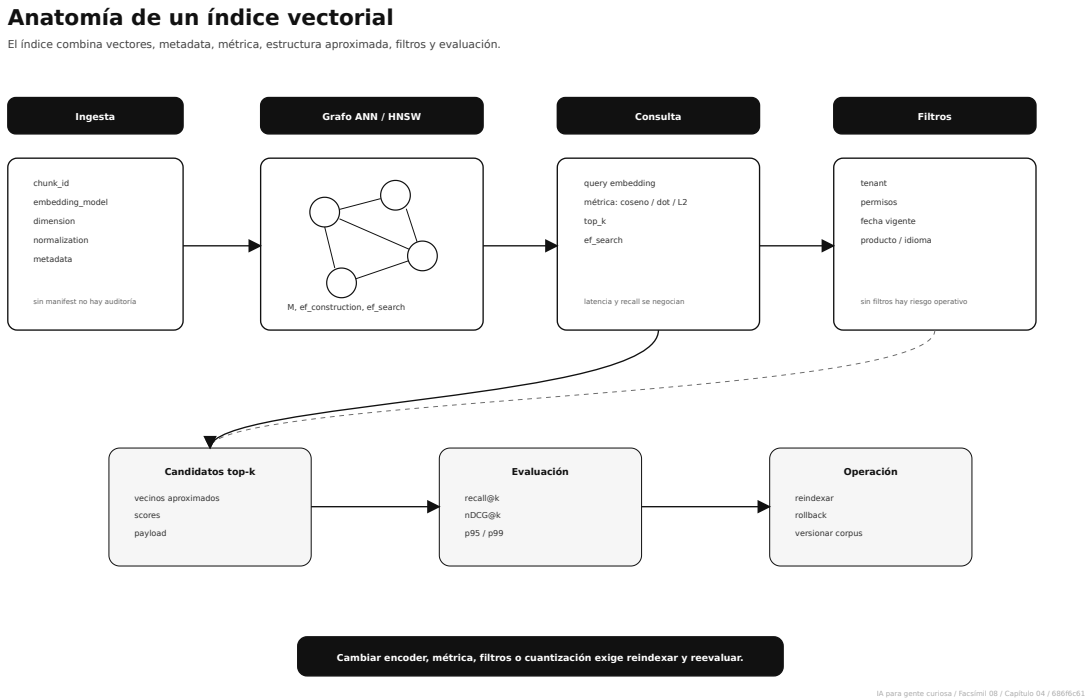
HNSW suele exponerse con parámetros que un ingeniero debe entender. `M` controla cuántas conexiones mantiene cada nodo del grafo; subirlo suele aumentar memoria y calidad de búsqueda. `ef_construction` afecta la calidad y coste de construir el índice. `ef_search` afecta el trabajo en consulta: subirlo puede mejorar recall, pero también latencia. Qdrant documenta estos parámetros dentro de su configuración de indexado, y Milvus también separa tipo de índice, métrica y parámetros de búsqueda.[40][41]

También aparece la compresión. Product Quantization, propuesta por Jégou, Douze y Schmid para búsqueda de vecinos cercanos, divide vectores en subespacios y los cuantiza para reducir memoria y acelerar búsqueda aproximada.[42] Es una herramienta potente, pero no gratuita: al comprimir puedes perder calidad de ranking. De nuevo, no se decide por intuición; se mide con qrels, slices, latencia y coste.

Los filtros de metadata son la otra mitad del sistema. Un índice que recupera un documento perfecto pero sin respetar permisos no es un éxito técnico; es un incidente. Pinecone, Qdrant, Milvus y Weaviate documentan filtrado por metadata o búsqueda híbrida porque en aplicaciones reales se busca dentro de subconjuntos: usuario, organización, idioma, fecha, producto, país, versión documental, estado legal o permisos.[43][44]

Cuando combinas ranking lexical y ranking vectorial, hay varias estrategias. Puedes concatenar candidatos, ponderar scores si están calibrados, usar un reranker o fusionar posiciones. Reciprocal Rank Fusion, de Cormack, Clarke y Buettcher, es una técnica clásica para fusionar rankings usando la posición de cada documento en distintas listas.[45] No hace falta que lo implementes en el primer día, pero sí conviene entender la idea: BM25 puede capturar códigos exactos; el embedding puede capturar paráfrasis; el reranker puede reordenar con más contexto. El sistema serio mide cada capa.

PIEZA DEL ÍNDICE	QUÉ CONTROLA	QUÉ PUEDE ROMPER	QUÉ MEDIRÍA
Dimensión	Memoria por vector y coste de distancia.	Latencia y tamaño de índice.	Memoria, p95/p99, recall@k.
Métrica	Coseno, producto escalar o L2.	Rankings distintos si no reindexas.	Comparación contra qrels.
HNSW `M`	Conectividad del grafo.	Memoria alta o búsqueda pobre.	Recall@k frente a memoria.
HNSW `ef_search`	Trabajo en consulta.	Latencia alta o vecinos perdidos.	Curva recall-latencia.
Cuantización	Compresión de vectores.	Pérdida de ranking fino.	nDCG@k antes/después.
Metadata filters	Alcance permitido de búsqueda.	Recuperar evidencia prohibida o caducada.	Tests por tenant, permisos y fecha.
Reindexado	Cuándo reconstruyes índice.	Mezclar modelos o versiones de corpus.	Manifest de versión y regresión de retrieval.



Un índice vectorial serio tiene parámetros, filtros, versiones, métricas y operación; no es solo una lista de embeddings.

En el día a día

En un equipo profesional, las features no deberían vivir escondidas dentro de un notebook. Deben tener nombre, definición, propietario, tipo, fuente, ventana temporal, reglas de null, método de cálculo, versión y criterio de retirada. Cuando la misma feature se usa para entrenamiento offline y predicción online, además aparece un problema clásico: que el cálculo de entrenamiento y el cálculo de producción no coincidan.

Ahí entra la idea de feature store: un sistema o una disciplina para compartir definiciones de features, materializarlas, servirlas y versionarlas. Feast, por ejemplo, documenta esta filosofía de definir entidades, feature views y fuentes para servir features de manera consistente.[46] Lo importante para este facsímil no es casarnos con una herramienta concreta, sino entender la responsabilidad: si una feature decide, esa feature debe poder explicarse.

En RAG ocurre algo parecido. El embedding de un chunk no es una columna tradicional, pero es un dato derivado. Debe conservar `document_id`, versión del documento, versión del chunking, encoder, dimensión, fecha de creación y permisos. Si cambias cualquiera de esas piezas, el índice ya no es el mismo aunque el texto visible parezca igual.

Por qué debería importarte

Una mala representación puede arruinar un sistema sin que el error sea evidente. El modelo compila, la API responde y la métrica puede subir, pero quizá aprendió una identidad, una fecha posterior, una categoría mal codificada o un embedding sin metadata. La representación es el lugar donde se decide qué puede aprender el sistema.

También es una cuestión de coste. Si tienes 10 millones de documentos y pasas de 384 a 3072 dimensiones, multiplicas almacenamiento y cálculo. Si además replicas el índice, guardas metadata y añades estructuras de vecinos aproximados, el coste ya no es marginal. FAISS y HNSW existen porque buscar vecinos en espacios vectoriales grandes requiere estructuras especializadas.[47][48]

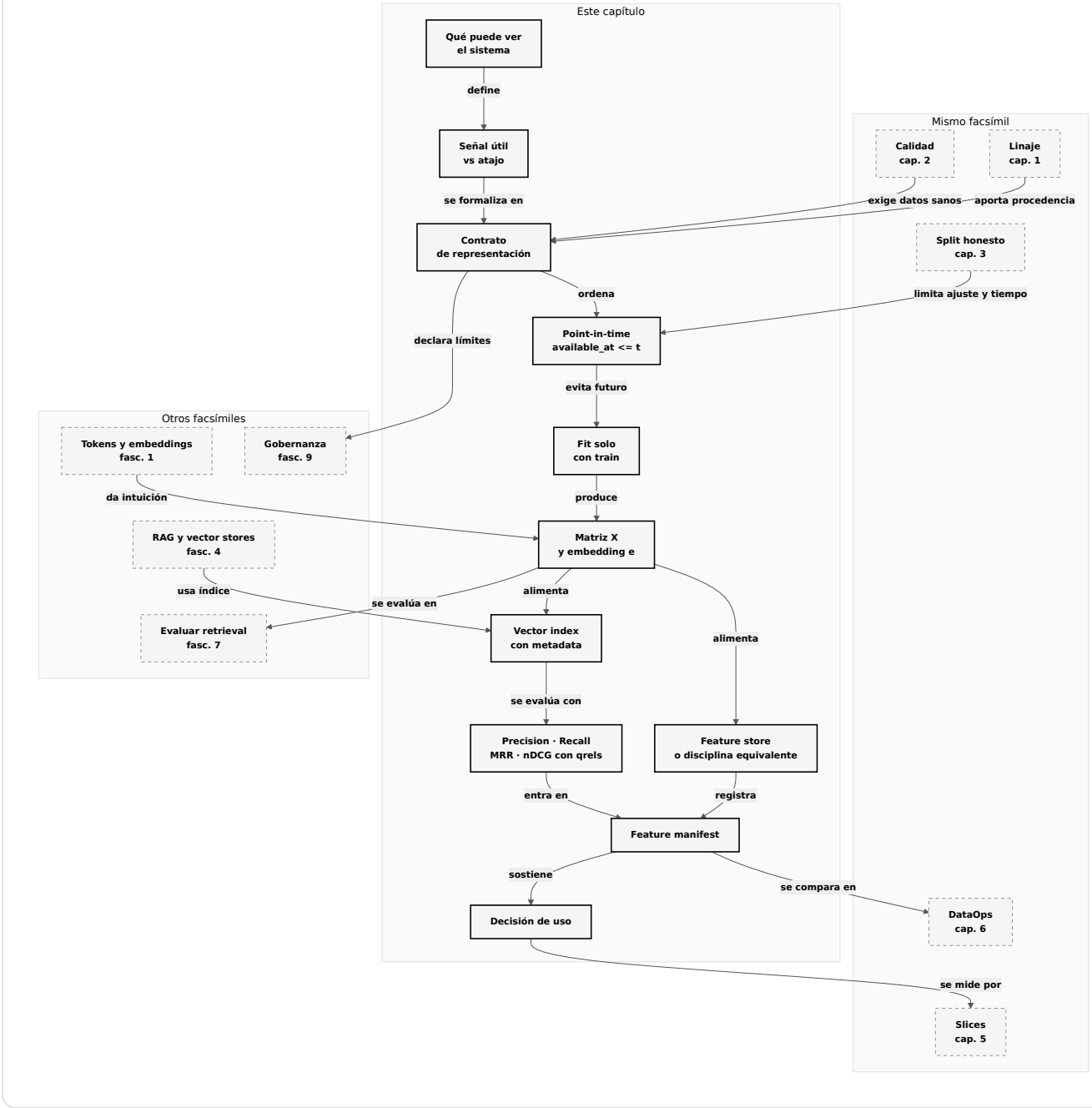
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Usar todas las columnas	Parece que más información siempre ayuda.	Separar entradas, target, IDs y metadata.
Ajustar vocabulario con todo el dataset	Es más cómodo hacerlo antes del split.	Crear split primero y hacer `fit` solo con train.
Llamar conocimiento a un embedding	El vector parece semántico.	Conservar evidencia, metadata, permisos y evaluación.
Ignorar dimensión y coste	El modelo de embeddings se ve como una caja externa.	Calcular almacenamiento, latencia e índice.
Mirar solo el top-1	El primer vecino puede ser casual.	Evaluar top-k, slices y términos fuera de vocabulario.
Confundir cobertura con recall@k	Una consulta “acierta producto” y parece suficiente.	Crear relevancia anotada antes de comparar retrieval en serio.
Entrenar y servir features distintas	Offline usa SQL histórico y online usa otra API o ventana.	Versionar definiciones y auditar feature skew.

Cómo encaja todo

Este mapa debe leerse como una brújula, no como un inventario de técnicas. Los capítulos anteriores del facsímil nos dieron linaje, calidad y splits; los facsímiles previos nos dieron tokens, embeddings, RAG y vector stores. Este capítulo hace de puente: decide **qué puede ver el sistema** y cómo queda esa decisión convertida en matriz, vector, manifiesto y búsqueda.

La continuidad posterior también es importante. Una representación no termina cuando se genera `feature_matrix.csv`: se evalúa por slices, se monitoriza cuando cambia la producción, se mide como recuperación si entra en RAG y se gobierna si toca permisos, identidad o decisiones sensibles.


Este capítulo
Qué puede ver el sistema
define
Señal útil vs atajo
se formaliza en
Contrato de representación
ordena
declara límites
Point-in-time available_at <= t
evita futuro
Fit solo con train
produce
Matriz X y embedding e
alimenta
alimenta
se evalúa en
Vector index con metadata
se evalúa con
Precision · Recall MRR · nDCG con qrels
Feature store o disciplina equivalente
entra en
registra
Feature manifest
sostiene
se compara en
Decisión de uso
Mismo facsímil
Calidad cap. 2
Linaje cap. 1
exige datos sanos
aporta procedencia
Split honesto cap. 3
limita ajuste y tiempo
DataOps cap. 6
se mide por
Slices cap. 5
Otros facsímiles
Tokens y embeddings fasc. 1
Gobernanza fasc. 9
da intuición
RAG y vector stores fasc. 4
usa índice
Evaluar retrieval fasc. 7

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Feature	Variable de entrada que un modelo puede usar.
Representación	Forma numérica de describir un objeto.
Alta cardinalidad	Situación en la que una variable categórica tiene muchos valores posibles.
One-hot	Codificación binaria de categorías.
Target encoding	Codificación de una categoría usando una estadística del objetivo, siempre con control estricto de leakage.
Normalización	Cambio de escala para comparar valores.
TF-IDF	Peso lexical basado en frecuencia local y rareza global.
Embedding	Vector denso que permite comparar objetos por similitud.
Dimensión	Longitud del vector o número de coordenadas.
Similitud coseno	Comparación de dirección entre dos vectores.
Out-of-vocabulary	Término que no estaba en el vocabulario ajustado.
Leakage de representación	Fuga introducida por transformaciones, selección o estadísticas ajustadas fuera de train.
Feature manifest	Artefacto que versiona columnas, vocabulario, hashes y dimensiones.
Feature skew	Diferencia entre el cálculo de features en entrenamiento y en producción.
Point-in-time correctness	Garantía de que la feature usada estaba disponible cuando se tomó la decisión.
Pipeline	Encadenamiento reproducible de transformaciones y modelo para evitar preprocesado suelto.
Contrato de datos	Reglas ejecutables sobre columnas, tipos, rangos, nulls, categorías y límites de uso.
Vector database	Sistema especializado, o extensión de base de datos, para almacenar y consultar embeddings.
Metadata filter	Filtro estructurado que limita la búsqueda vectorial por permisos, tenant, fecha, producto o fuente.

TÉRMINO	DEFINICIÓN BREVE
Búsqueda híbrida	Combinación de señal lexical, como BM25, y señal vectorial densa.
BM25	Ranking lexical que ajusta frecuencia, rareza global y longitud del documento.
ANN	Búsqueda aproximada de vecinos cercanos para acelerar recuperación vectorial.
HNSW	Índice de búsqueda aproximada basado en grafos jerárquicos.
Product Quantization	Técnica de compresión para búsqueda de vecinos cercanos con menor memoria.
Precision@k	Proporción de resultados relevantes dentro de los k primeros.
Recall@k	Proporción de documentos relevantes recuperados dentro de los k primeros.
MRR	Media del recíproco del ranking del primer resultado relevante.
nDCG@k	Métrica de ranking con relevancia graduada y descuento por posición.
Qrels	Tabla de relevancia anotada para evaluar recuperación de información.
Hard negative	Documento parecido al relevante, pero incorrecto para la consulta.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué una feature no es simplemente una columna?
2. ¿Qué diferencia hay entre feature, metadata y target?
3. ¿Por qué one-hot necesita vocabulario ajustado en train?
4. ¿Por qué una categoría de alta cardinalidad no debería ir a one-hot sin pensarlo?
5. ¿Qué riesgo tiene el target encoding si se calcula con todo el dataset?
6. ¿Qué aprende TF-IDF y por qué puede contaminar si se ajusta con todo el dataset?
7. ¿Qué significa que un embedding viva en $\mathbb{R}^m$?
8. ¿Qué mide la similitud coseno?
9. ¿Por qué la dimensión afecta coste y ranking?
10. ¿Qué guarda `feature_manifest.json`?
11. ¿Por qué `search_report.json` queda en `review`?
12. ¿Qué cambiarías para usar embeddings reales sin perder trazabilidad?

13. ¿Qué diferencia hay entre cobertura top-k por producto y recall@k formal?
14. ¿Cuándo usarías MRR y cuándo nDCG@k?
15. ¿Qué es un hard negative y por qué mejora una evaluación de retrieval?
16. ¿Por qué una feature puede ser correcta matemáticamente y aun así romper point-in-time correctness?
17. ¿Sabrías decir de qué tradición viene cada fórmula del capítulo y qué decisión práctica permite tomar?
18. ¿Qué herramienta usarías para validar contrato de datos y cuál usarías para servir features online?
19. ¿Qué parámetros mirarías en un índice HNSW antes de producción?
20. ¿Cuándo elegirías pgvector y cuándo una base vectorial dedicada?
21. ¿Qué metadata mínima exigirías antes de indexar chunks para RAG?

En resumen

IDEA	QUÉ TE LLEVAS
La representación es parte del sistema.	No se improvisa dentro de un notebook.
El `fit` pertenece a train.	Escalas, vocabularios e IDF se congelan antes de validation y test.
No todas las familias de features se tratan igual.	Numéricas, temporales, categóricas, texto y metadata tienen riesgos distintos.
El leakage puede entrar por la transformación.	Selección, escalado, IDF o target encoding pueden mirar test sin que se note.
Embedding no significa verdad.	Es una geometría útil que necesita metadata y evaluación.
La dimensión tiene coste.	Afecta memoria, latencia, índice y calidad de ranking.
Offline y online deben coincidir.	Si entrenamiento y producción calculan distinto, aparece feature skew.
Retrieval necesita relevancia anotada.	Sin `qrels`, la cobertura top-k es una señal inicial, no una métrica final.
El ranking se mide por más de una métrica.	Precision, recall, MRR y nDCG responden preguntas distintas.
Un índice vectorial es infraestructura.	Métrica, HNSW, filtros, cuantización, reindexado y permisos cambian el sistema.
La herramienta no sustituye el contrato.	Pipeline, feature store o vector DB solo ayudan si sabes qué decisión protegen.
El índice necesita metadata.	Sin permisos, versión, fecha y fuente, el embedding es difícil de auditar.
Una práctica real deja artefactos.	Contrato, matriz, manifiesto, reporte y decisión.

Para saber más

Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *SIGIR 2009*, 758-759. DOI

Databricks. (2026). Feature tables in Unity Catalog. Documentación

Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT 2019*, 4171-4186. DOI

Feast. (2026). Feast Documentation. Documentación

Feast. (2026). Point-in-time Joins. Documentación

Google Cloud. (2026). Introduction to feature management. Documentación

Great Expectations. (2026). Expectations Overview. Documentación

Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro

Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI

Järvelin, K. y Kekäläinen, J. (2002). Cumulated Gain-Based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4), 422-446. DOI

Jégou, H., Douze, M. y Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117-128. DOI

Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI

Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI

Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv

Milvus. (2026). Filtered Search. Documentación

OpenAI. (2026). Vector embeddings. Documentación

Pandera. (2026). Pandera Documentation. Documentación

pgvector. (2026). pgvector: Open-source vector similarity search for Postgres. GitHub

Pinecone. (2026). Hybrid search. Documentación

Qdrant. (2026). Indexing. Documentación

Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI

Robertson, S. y Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. DOI

scikit-learn. (2026). Common pitfalls and recommended practices. Documentación

scikit-learn. (2026). Column Transformer with Mixed Types. Documentación

scikit-learn. (2026). MinMaxScaler. Documentación

scikit-learn. (2026). Normalizer. Documentación

scikit-learn. (2026). OneHotEncoder. Documentación

scikit-learn. (2026). Pipeline. Documentación

scikit-learn. (2026). Permutation Feature Importance. Documentación

scikit-learn. (2026). TfidfTransformer. Documentación

Tecton. (2026). Construct Training Data. Documentación

Weaviate. (2026). Hybrid search. Documentación

Notas

1. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro. ↩

2. Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI. ↩

3. scikit-learn. (2026). *MinMaxScaler*. Documentación. Consultado el 22 de junio de 2026. ↩

4. scikit-learn. (2026). *Common pitfalls and recommended practices*. Documentación. Consultado el 22 de junio de 2026. ↩

5. scikit-learn. (2026). *Pipeline*. Documentación. Consultado el 22 de junio de 2026. ↩

6. scikit-learn. (2026). *OneHotEncoder*. Documentación. Consultado el 22 de junio de 2026. ↩

7. scikit-learn. (2026). *TfidfTransformer*. Documentación. Consultado el 22 de junio de 2026. ↩

8. Robertson, S. y Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333-389. DOI. ↩

9. Kaufman, S., Rosset, S., Perlich, C. y Stitelman, O. (2012). Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), 1-21. DOI. ↩

10. scikit-learn. (2026). *Permutation Feature Importance*. Documentación. Consultado el 21 de junio de 2026. ↩

11. Hastie, T., Tibshirani, R. y Friedman, J. (2009). *The Elements of Statistical Learning* (2.ª ed.). Springer. Libro. ↩

12. Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper. ↩

13. Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI. ↩

14. Bengio, Y., Ducharme, R., Vincent, P. y Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3, 1137-1155. Paper. ↩

15. Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv. ↩

16. Devlin, J., Chang, M.-W., Lee, K. y Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT 2019*, 4171-4186. DOI. ↩

17. Reimers, N. y Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP-IJCNLP 2019*, 3982-3992. DOI. ↩

18. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

19. scikit-learn. (2026). *Normalizer*. Documentación. Consultado el 22 de junio de 2026. ↩

20. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

21. Manning, C. D., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. ↩

22. Järvelin, K. y Kekäläinen, J. (2002). Cumulated Gain-Based Evaluation of IR Techniques. *ACM Transactions on Information Systems*, 20(4), 422-446. DOI. ↩

23. scikit-learn. (2026). *Column Transformer with Mixed Types*. Documentación. Consultado el 22 de junio de 2026. ↩

24. Pandera. (2026). *Pandera Documentation*. Documentación. Consultado el 6 de junio de 2026. ↩

25. Great Expectations. (2026). *Expectations Overview*. Documentación. Consultado el 6 de junio de 2026. ↩

26. Feast. (2026). *Feature retrieval*. Documentación. Consultado el 22 de junio de 2026. ↩

27. Tecton. (2026). *Construct Training Data*. Documentación. Consultado el 22 de junio de 2026. ↩

28. Databricks. (2026). *Feature tables in Unity Catalog*. Documentación. Consultado el 22 de junio de 2026. ↩

29. Google Cloud. (2026). *Introduction to feature management*. Documentación. Consultado el 22 de junio de 2026. ↩

30. OpenAI. (2026). *Vector embeddings*. Documentación. Consultado el 22 de junio de 2026. ↩

31. pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. GitHub. Consultado el 25 de mayo de 2026. ↩

32. Qdrant. (2026). *Indexing*. Documentación. Consultado el 25 de mayo de 2026. ↩

33. Weaviate. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

34. Milvus. (2026). *Filtered Search*. Documentación. Consultado el 25 de mayo de 2026. ↩

35. Pinecone. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

36. Feast. (2026). *Point-in-time Joins*. Documentación. Consultado el 22 de junio de 2026. ↩

37. Tecton. (2026). *Construct Training Data*. Documentación. Consultado el 22 de junio de 2026. ↩

38. Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI. ↩

39. Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI. ↩

40. Qdrant. (2026). *Indexing*. Documentación. Consultado el 25 de mayo de 2026. ↩

41. Milvus. (2026). *Filtered Search*. Documentación. Consultado el 25 de mayo de 2026. ↩

42. Jégou, H., Douze, M. y Schmid, C. (2011). Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1), 117-128. DOI. ↩

43. Pinecone. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

4. Weaviate. (2026). *Hybrid search*. Documentación. Consultado el 25 de mayo de 2026. ↩

5. Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods. *SIGIR 2009*, 758-759. DOI. ↩

6. Feast. (2026). *Feast Documentation*. Documentación. Consultado el 6 de junio de 2026. ↩

7. Johnson, J., Douze, M. y Jégou, H. (2019). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3), 535-547. DOI. ↩

8. Malkov, Y. A. y Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824-836. DOI. ↩