Facsímil 08

La ciencia de los datos

Capítulo 03

Splits, muestreo y leakage: medir sin engañarse

686f6c61 · 2026

Capítulo 03: Splits, muestreo y leakage: medir sin engañarse

Entrando en el tema

Hay una forma muy cómoda de engañarse con datos sin mala intención: partir el dataset, entrenar, mirar una métrica y respirar tranquilo. El problema es que la métrica puede estar respondiendo a otra pregunta. No “¿generaliza el sistema?”, sino “¿cuánto se parece test a lo que ya vio el sistema, el equipo o el pipeline?”.

En ciencia de datos para IA, el split es una decisión experimental. Decide qué mundo puede conocer el sistema, qué mundo usamos para ajustar decisiones y qué mundo queda reservado para comprobar si la conclusión se sostiene. Si esa frontera está mal puesta, todo lo posterior queda torcido: features, embeddings, RAG, evaluación de LLMs, slices, drift y gobernanza.

Este capítulo no va de memorizar `train_test_split`. Va de aprender a escribir una política de partición defendible: qué unidad separas, qué leaks buscas, qué transformaciones pueden aprender, cuándo el test deja de ser test y qué evidencia guardas para que otra persona pueda repetir la evaluación.

Qué deberías poder hacer al terminar

El capítulo anterior nos enseñó a bloquear un dataset con fallos de calidad. Pero un dataset puede pasar schema, licencias, etiquetas y duplicados básicos, y aun así medir mal. La razón suele estar en cómo lo partimos.

Un split no es un porcentaje. Es una promesa de evaluación. Cuando decimos `train`, `validation` y `test`, no estamos repartiendo filas al azar como quien reparte cartas. Estamos diciendo qué puede ver el sistema, qué usamos para ajustar decisiones y qué reservamos para comprobar si generaliza.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Explicar para qué sirve cada split.	No usas test para elegir prompt, modelo o umbral.
Elegir estrategia de muestreo según la pregunta.	No haces random split si hay tiempo, grupos o fuentes repetidas.
Detectar leakage por entidad, fuente, texto y tiempo.	Sabes buscar estudiantes, documentos o textos que cruzan splits.
Medir distribución de etiquetas por split.	No celebras un split si test pierde clases críticas.
Comparar estrategias de partición.	Puedes justificar random, estratificado, por grupo, temporal o híbrido.
Construir un manifiesto de split.	Dejas asignaciones y hallazgos versionados.

La gracia de esta lista está en que no puedes cumplirla solo con una función de librería. Puedes llamar a `train_test_split` correctamente y aun así medir mal si la unidad, el tiempo o la fuente documental no encajan con la pregunta. Un buen split tiene intención: dice qué mundo representa train, qué decisiones permite validation y qué realidad debe simular test.

La frase central del capítulo:

> Un split mide una pregunta. Si partes mal, la respuesta parece científica y no lo es.

La escena: una métrica que sube demasiado

Imagina que evaluamos un asistente académico. Hacemos un `train/test split` aleatorio y obtenemos una métrica preciosa. El sistema responde muy bien sobre matrículas, becas y pagos. Todo parece listo para enseñar.

Luego alguien mira los casos. Un estudiante aparece en train y test. Un documento fuente está repetido en ambos. Una consulta de test es casi igual a una de train. Además, algunos ejemplos de train tienen fecha posterior a ejemplos de test. La métrica no está midiendo generalización: está midiendo familiaridad.

Este es uno de los errores más caros porque no rompe el código. Al contrario: el código funciona, el reporte se ve profesional y la gráfica sube. Justo por eso hay que tratar los splits como ingeniería, no como una llamada rápida a una función.

Qué no es un split

Un split no es “60/20/20 y ya”. Esos números dicen cuántas filas hay, no si la partición responde bien a la pregunta. Si hay varios tickets de la misma persona, un random split puede poner unos en train y otros en test. Si hay documentos con versiones parecidas, puede partir la misma evidencia en dos. Si hay tiempo, puede entrenar con eventos posteriores y evaluar con eventos anteriores.

Tampoco es una garantía automática de independencia. Dos filas distintas pueden compartir entidad, fuente, plantilla, texto, anotador o evento. Para un modelo o un retriever, esa cercanía puede ser suficiente para inflar resultados.

Y test no es un lugar donde mirar muchas veces. Si usamos test para decidir prompts, elegir modelo, ajustar umbrales, cambiar chunking o seleccionar features, test deja de ser una medida final. Se convierte en otra validación, aunque sigamos llamándolo test.

Qué sí es un split de evaluación

Un split es una partición diseñada para responder a una pregunta de generalización. La notación siguiente no es una métrica propia del facsímil: es la forma básica de escribir una partición de conjuntos. El dataset completo se expresa como unión de subconjuntos, y esos subconjuntos no deben compartir filas:

$$D = D_{train} \cup D_{val} \cup D_{test}$$

con una condición de disjunción por pares:

$$D_i \cap D_j = \varnothing$$

para $i \neq j$, con $i, j \in \{train, val, test\}$. Dicho sin símbolos: una misma fila no debería estar a la vez en entrenamiento, validación y test.

SÍMBOLO	SIGNIFICADO	EJEMPLO
D	Dataset completo.	24 casos de soporte.
D_{train}	Datos que el sistema puede usar para aprender.	Casos anteriores usados para ajustar.
D_{val}	Datos para elegir decisiones durante desarrollo.	Elegir prompt, umbral o estrategia.
D_{test}	Datos reservados para medir al final.	Casos no usados para ajustar nada.
$D_i \cap D_j = \varnothing$	Los splits son disjuntos por pares.	Ninguna fila debería estar en dos splits.

La intersección vacía de filas es necesaria, pero no suficiente. También necesitamos evitar solapamiento de entidades, fuentes o información temporal cuando esas dimensiones afecten a la pregunta. En un dataset de IA, dos filas distintas pueden seguir estando contaminadas si pertenecen al mismo estudiante, salen del mismo documento o se generaron desde la misma traza.

Fecha de corte del estado del arte

Fecha de corte: 22 de junio de 2026. **Fuentes consultadas:** documentación oficial de scikit-learn sobre `train_test_split`, `GroupKFold`, `StratifiedGroupKFold`, `TimeSeriesSplit`, `Pipeline` y sus errores comunes; literatura sobre leakage en data mining y reproducibilidad de evaluación con machine learning; Wilson para intervalos de proporciones; Efron para bootstrap; Codd para pensar tablas y restricciones; DuckDB, dbt, Dagster y Airflow para aterrizar checks y orquestación; Databricks, Tecton y Feast para point-in-time joins; DVC y MLflow para trazabilidad de datasets; y el paper original de Retrieval-Augmented Generation para situar por qué, en RAG, documentos y consultas también forman parte de la evaluación.

`train_test_split` permite partir arrays o matrices en subconjuntos aleatorios de entrenamiento y test, con opciones como `test_size`, `train_size`, `random_state`, `shuffle` y `stratify`.[1] Esa función es útil, pero no decide por nosotros si un random split es correcto.

`GroupKFold` mantiene grupos no solapados entre folds, y `StratifiedGroupKFold` intenta preservar proporciones de clases sin partir grupos.[2][3] `TimeSeriesSplit` trabaja con particiones ordenadas en el tiempo, donde cada train es anterior al test correspondiente.[4]

Kaufman, Rosset, Perlich y Stitelman definieron leakage como la introducción de información sobre el objetivo que no debería estar disponible legítimamente para el modelo, y lo trataron como un error recurrente en proyectos reales y competiciones.[5] Kapoor y Narayanan revisaron leakage como una fuente importante de problemas de reproducibilidad en ciencia basada en machine learning.[6]

Lewis y colaboradores formalizaron RAG como una combinación de modelo paramétrico y memoria no paramétrica recuperada desde documentos.[7] Esa idea cambia la conversación sobre splits: no basta con partir preguntas, también hay que controlar documentos, versiones, chunks e índices de recuperación.

scikit-learn documenta `Pipeline` como una secuencia de transformadores y, en sus errores comunes, lo recomienda para evitar que estadísticas del test se filtren hacia el entrenamiento durante preprocesamiento, validación cruzada o búsqueda de hiperparámetros.[8][9] DVC se presenta como control de versiones para datos, modelos y experimentos, y MLflow documenta tracking de datasets y evaluation datasets para conservar linaje entre datos, evaluación y resultados.[10][11][12]

La lección estable es esta: las funciones de partición ayudan, pero la garantía viene de entender la estructura del dato.

Estrategias de muestreo

Antes de elegir una estrategia, debemos escribir la pregunta de evaluación. ¿Queremos saber si el sistema generaliza a nuevos casos de las mismas personas? ¿A nuevas personas? ¿A documentos futuros? ¿A otro periodo? ¿A clases raras? Cada pregunta pide un split distinto.

ESTRATEGIA	QUÉ PRESERVA	QUÉ PUEDE ROMPER
Random por fila	Proporciones aproximadas si hay suficientes datos.	Grupos, fuentes, tiempo y textos cercanos.
Estratificado por etiqueta	Distribución de clases.	Entidades repetidas o tiempo.
Por grupo	Entidades no solapadas.	Distribución de etiquetas o cronología.
Temporal	Simulación de futuro.	Grupos repetidos o clases raras.
Temporal por grupo	Tiempo y entidad a la vez.	Puede dejar alguna distribución en revisión.

No hay estrategia universal. Si todos los casos de una misma persona se parecen, necesitas grupo. Si el producto cambia con el tiempo, necesitas tiempo. Si hay clases muy raras, necesitas mirar cobertura por etiqueta. Si hay documentos comunes, quizá necesitas agrupar por fuente.

La estrategia correcta suele ser la que te obliga a renunciar a comodidad. Un random split da más filas repartidas y métricas más estables, pero puede estar midiendo familiaridad. Un split por grupo baja muestra efectiva, pero responde mejor si quieres generalizar a nuevas personas o empresas. Un split temporal puede ser más duro, pero se parece más a producción. En ingeniería de IA, una métrica más baja con un split honesto suele valer más que una métrica brillante con una partición ingenua.

Las fórmulas que sí conviene saber

Para medir si los splits conservan etiquetas, podemos comparar proporciones empíricas. Esto es la frecuencia relativa de una clase dentro de un split: no es una notación propia del facsímil, sino la forma estándar de estimar una distribución categórica a partir de conteos.

$$p_s(y = k) = \frac{n_{s,k}}{n_s}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
s	Split.	`train`, `validation`, `test`.
k	Etiqueta concreta.	`escalate`.
$n_{s,k}$	Ejemplos de la etiqueta k en el split s.	Casos `escalate` en test.
n_s	Total de ejemplos del split s.	5 casos de test.

Para comparar la distribución de un split con la global usamos distancia de variación total, una distancia habitual entre distribuciones de probabilidad.[13]

$$D_{TV}(P, Q) = \frac{1}{2} \sum_i |P_i - Q_i|$$

SÍMBOLO	SIGNIFICADO
P_i	Proporción global de la clase i.
Q_i	Proporción de la clase i en el split.
D_{TV}	Distancia entre ambas distribuciones.

Para textos cercanos usamos Jaccard, igual que en el capítulo anterior. La similitud de Jaccard viene de comparar intersección y unión de conjuntos.[14]

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

SÍMBOLO	SIGNIFICADO
A	Tokens del primer texto.
B	Tokens del segundo texto.
$\text{card}(A \cap B)$	Tokens compartidos.
$\text{card}(A \cup B)$	Tokens únicos totales.

Para leakage temporal no necesitamos inventar una métrica. Necesitamos una comprobación directa: si train contiene información posterior a la ventana que se quiere simular como futuro, la partición no sirve para esa pregunta.

COMPROBACIÓN	QUÉ MIRA	QUÉ IMPLICA SI FALLA
Fecha de train	Que las filas de aprendizaje son anteriores a test.	El modelo puede haber visto futuro.
Fecha de features	Que cada feature existía en el momento de decisión.	Hay leakage por variable calculada tarde.
Fecha de fuente	Que documentos y versiones pertenecen a la ventana correcta.	El RAG puede recuperar evidencia que no existía.
Fecha de etiqueta	Que la etiqueta no depende de un evento posterior no disponible.	La evaluación mide una ventaja imposible.

La lectura práctica de estas fórmulas y comprobaciones no es decorar el capítulo con símbolos. La distribución por split te dice si cada partición tiene casos suficientes para medir. La distancia de variación total te avisa cuando una partición deja de parecerse al conjunto que dice representar. Jaccard te ayuda a encontrar textos demasiado cercanos. Las fechas evitan que el sistema mire futuro. Ninguna decide sola, pero juntas obligan a mirar el split como un objeto medible y no como una partición hecha al vuelo.

Tipos de leakage que debe mirar un ingeniero

Leakage no siempre se ve como una fila duplicada. Muchas veces aparece como una relación escondida.

TIPO	CÓMO APARECE	QUÉ CHECK AYUDA
Por fila	Mismo `case_id` o mismo texto.	Duplicado exacto.
Por entidad	Misma persona, cliente, documento o dispositivo en train y test.	Group split.
Por fuente	Mismo `source_id` o versión documental.	Agrupar por fuente.
Por texto cercano	Preguntas casi iguales en splits distintos.	Jaccard, embeddings o revisión.
Temporal	Train ve eventos posteriores a test.	Split temporal.
De preprocesamiento	Escalado, imputación o selección de features hecha con todo el dataset.	Fit solo con train.
De decisión	Test se usa para elegir prompt, modelo o umbral.	Separar validation y test real.

El leakage de preprocesamiento merece una frase aparte. Si calculas la media de una columna con todo el dataset y luego escalas train y test, test ya influyó en train. La solución es sencilla conceptualmente: `fit` de transformaciones solo en train; `transform` en validation y test.

El split como contrato de evaluación

En ingeniería conviene dejar de hablar del split como una operación suelta y empezar a tratarlo como un contrato. Un contrato de split responde a preguntas que deberían quedar por escrito:

PREGUNTA	RESPUESTA QUE DEBE QUEDAR VERSIONADA
¿Qué decisión medimos?	Clasificador, RAG, prompt, modelo, retriever, umbral o pipeline completo.
¿Qué estrategia se eligió?	Random, estratificada, por grupo, temporal, temporal por grupo o una combinación propia.
¿Qué claves protegen la independencia?	`case_id`, `student_id`, `source_id`, `created_at`, `label`, `text`.
¿Qué puede hacerse con cada split?	Train entrena, validation decide, test mide, holdout confirma.
¿Qué está prohibido después de mirar test?	Elegir modelo, reescribir prompt, ajustar umbral, cambiar chunking o modificar el retriever.
¿Qué hashes demuestran reproducibilidad?	Hash del dataset, hash de la política y asignaciones por split.

El artefacto que recoge todo eso es el manifiesto de split. En el cuaderno del facsímil se genera como `output/split_manifest.json`. Un manifiesto mínimo debería contener:

```
{
  "policy_id": "support-split-policy-v1",
  "dataset": {
    "sha256": "..."
  },
  "split_contract": {
    "selected_strategy": "time_group_holdout",
    "gate": "review",
    "keys": {
      "group": "student_id",
      "source": "source_id",
      "time": "created_at"
    }
  }
}
```

El hash no es decoración. Si cambias una fila, una etiqueta o una fecha, el hash del dataset cambia. Si cambias una regla de la política, el hash de la política cambia. Eso permite reconstruir qué se midió, con qué reglas y por qué la métrica era defendible en ese momento.

En un equipo real, el manifiesto evita discusiones imposibles semanas después. Si alguien pregunta “¿por qué aquella métrica bajó al repetir el experimento?”, no deberíamos responder mirando una gráfica suelta. Deberíamos poder abrir el manifiesto, comprobar el hash del dataset, ver qué política estaba activa, saber qué estrategia asignó cada `case_id` y

revisar si test se usó solo para medir o también para decidir. Esa diferencia parece administrativa, pero es ingeniería: sin trazabilidad, una métrica deja de ser una medida y se convierte en una anécdota difícil de reproducir.

El manifiesto también ayuda a integrar evaluación en CI/CD. Puedes bloquear una ejecución si cambia el hash del dataset sin regenerar el split, si falta una predicción para un caso de test, si validation o test se usan para hacer `fit` de un transformador, o si una política nueva modifica los umbrales sin actualizar la decisión. En proyectos de IA, muchas regresiones no vienen de una línea de modelo, sino de una línea de datos que cambió sin que nadie la registrara.

Es un artefacto de ingeniería. Un manifiesto de split debe guardar, como mínimo, estas piezas:

PIEZA	QUÉ DEBE QUEDAR VERSIONADO	POR QUÉ IMPORTA
Dataset	Hash del snapshot evaluado.	Si cambia una fila, la métrica ya no describe lo mismo.
Política	Hash o versión de la regla de split.	Permite saber por qué se eligió una estrategia.
Estrategia	Random, estratificada, por grupo, temporal o combinada.	Explica qué generalización se pretendía medir.
Asignación	Qué IDs quedaron en train, validation y test.	Permite reproducir la evaluación.
Gate	`pass`, `review` o `block`, con motivos.	Convierte la partición en decisión auditable.

La métrica sin manifiesto es débil. Puede estar bien, pero no sabemos repetirla ni auditarla.

Point-in-time correctness: el split también vive en las features

Hasta aquí hemos hablado de filas, etiquetas, grupos y fuentes. Para un ingeniero de datos falta una pieza crítica: cada feature también tiene tiempo. No basta con que el `case_id` esté en el split correcto si las columnas usadas para entrenar o evaluar fueron calculadas con información posterior al momento en que ese caso existía.

En feature stores y pipelines de entrenamiento esto suele llamarse *point-in-time correctness*: construir cada fila de entrenamiento con los valores que estaban disponibles en el momento de decisión, no con el estado final de la tabla cuando alguien reejecuta el job.[151617]

La restricción formal es sencilla y poderosa. Si el caso e_i se decide en el instante t_i, una feature x_{ij} solo es válida si su fecha de disponibilidad no es posterior a t_i:

$$\text{available\_at}(x_{ij}) \leq t_i$$

SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
e_i	Evento o caso que queremos entrenar/evaluar.	Caso `s023`.
t_i	Momento en que ese caso existía para el sistema.	`created_at = 2026-04-23`.
x_{ij}	Feature j usada para ese caso.	`open_cases_30d`.
$\text{available\_at}(x_{ij})$	Momento en que esa feature estaba disponible para usar.	`2026-04-20` o `2026-04-24`.

Esta desigualdad no es una métrica nueva del facsímil. Es una forma explícita de escribir la restricción de no mirar información que no estaba legítimamente disponible, justo el tipo de fuga que Kaufman y colaboradores describen como leakage. Si `s023` ocurrió el 23 de abril de 2026, una feature disponible el 20 de abril es válida; una feature recalculada o cargada el 24 de abril no puede usarse para simular la decisión del día 23.

Cuando hay varias versiones históricas de una feature, la operación correcta es un *as-of join*: para cada caso, elegir la versión más reciente que no mire al futuro. La condición temporal es:

$$\text{available\_at}(r) \leq t_i$$

Entre los registros que cumplen esa condición, elegimos el más reciente:

$$r_i^\star = \underset{r \in H_j(e_i),\ \text{available\_at}(r) \leq t_i}{\arg\max} \text{available\_at}(r)$$

Después usamos el valor de ese registro histórico como feature del caso:

$$x_j(e_i) = \text{value}(r_i^\star)$$

SÍMBOLO	SIGNIFICADO
$H_j(e_i)$	Historial de valores candidatos de la feature j para la entidad del caso e_i.
r	Una versión histórica concreta de esa feature.
$r_i^\star$	Registro histórico elegido para el caso e_i.
$\arg\max$	Selecciona la versión candidata con mayor fecha de disponibilidad permitida.

Ejemplo: si el caso `s020` es del 20 de abril y tenemos features del 19 y del 22, el as-of join debe quedarse con el valor del 19. El del 22 puede ser correcto como dato histórico, pero es incorrecto para ese evento. Si el job no distingue `event_time`, `feature_timestamp`, `available_at` e `ingested_at`, tarde o temprano terminará entrenando con futuro.

Esta es una diferencia importante entre ciencia de datos de notebook y ciencia de datos operable. En un notebook puedes leer la tabla final y calcular una métrica. En un sistema real necesitas saber qué snapshot, qué ventana y qué versión de feature existían cuando se tomó cada decisión.

Split materializado: que la partición sea una tabla

El split no debería vivir solo dentro de un script. Si una asignación decide qué filas entrenan y qué filas miden, debe poder auditarse como cualquier otra tabla de datos. En teoría relacional, Codd propuso tratar los datos como relaciones con atributos, claves y restricciones.[18] Para un split, esa idea baja a una regla muy concreta: cada caso debe tener una y solo una asignación activa en una versión de split.

Si A_v es la tabla de asignaciones de la versión v, la restricción de unicidad puede escribirse así:

$$\forall c \in D, \quad \operatorname{card}\{s : (c, s) \in A_v\} = 1$$

Aquí s pertenece al catálogo cerrado de splits: train, validation o test. La fórmula no pretende inventar una métrica nueva; expresa una restricción relacional muy básica: para cada caso hay exactamente una asignación en la versión del split.

SÍMBOLO	SIGNIFICADO	QUÉ COMPRUEBA
D	Dataset que queremos partir.	Todos los `case_id` del snapshot.
A_v	Tabla de asignaciones del split versión v.	`dataset_split_assignments.csv`.
c	Caso individual.	`s023`.
s	Split asignado.	`train`, `validation` o `test`.
$\text{card}(\cdot) = 1$	Hay exactamente una asignación.	No falta ni se duplica el caso.

El cuaderno materializa esa tabla como `output/dataset_split_assignments.csv`. No sustituye al manifiesto; lo complementa. El manifiesto cuenta la política, hashes y decisión. La tabla de asignaciones permite preguntar con SQL:

COLUMNA	POR QUÉ EXISTE
`case_id`	Clave de la fila evaluada.
`split`	Uso permitido de la fila.
`split_version`	Versión reproducible de la partición.
`policy_id`	Política que generó la asignación.
`dataset_sha256`	Snapshot de datos al que pertenece.
`policy_sha256`	Reglas exactas usadas para partir.
`assigned_at_utc`	Momento en que se generó la tabla.

En un equipo de datos, esta tabla puede vivir en DuckDB, BigQuery, Snowflake, Postgres o el almacén que uses. Lo importante no es la marca, sino que el split deje de ser memoria de un script y pase a ser un artefacto consultable, versionado y revisable por otra persona.

SQL de auditoría: pensar como data engineer

Una práctica útil es convertir los checks de split en consultas. dbt llama *data tests* a consultas que devuelven filas que incumplen una aserción; si no devuelven filas, la aserción pasa.[19] Dagster permite asociar checks a assets de datos.[20] Esa filosofía encaja muy bien aquí: un split sano no se “cree”, se comprueba.

En el cuaderno del facsímil hay dos niveles. Primero, `ops/run_sql_audit.py` ejecuta una auditoría con biblioteca estándar de Python para funcionar sin instalar nada. Segundo, `sql/split_audit_duckdb.sql` deja las consultas en estilo DuckDB para que un alumno pueda llevarlas a una base analítica real. DuckDB permite leer CSV directamente desde SQL, lo que lo convierte en una herramienta cómoda para prototipar auditorías locales.[21]

Las preguntas que hacemos con SQL son muy de ingeniería:

PREGUNTA	QUÉ REVELA
¿Hay `case_id` duplicados en la tabla de split?	La misma fila podría entrenar y evaluar.
¿Falta algún `case_id` del dataset?	La evaluación no cubre todo el snapshot declarado.
¿Un `student_id` aparece en train y test?	Leakage por entidad.
¿Un `source_id` aparece en train y test?	Leakage por fuente documental.
¿Hay features posteriores al `created_at` del caso?	Un join ingenuo puede mirar futuro.
¿El test cubre las etiquetas obligatorias?	La métrica global puede esconder clases ausentes.

La parte importante es el matiz de las features. Que existan features posteriores a un caso no es malo por sí mismo: los datos históricos pueden seguir creciendo. Lo malo es que el join elija esas filas. Por eso el reporte distingue entre "hay candidatas futuras que un join ingenuo podría coger" y "el as-of join seguro selecciona la última feature disponible antes de la decisión".

Este es un buen ejemplo de práctica que sí se lleva al trabajo. Un SQL de auditoría puede vivir en dbt, en DuckDB local, en BigQuery o en Snowflake, pero la pregunta es la misma: "¿qué filas violan mi contrato?". Si la consulta devuelve casos, no tienes una opinión; tienes evidencia revisable. Esa forma de trabajar acerca la evaluación de IA a ingeniería de datos profesional.

Backfills y re-splits

Un *backfill* es un reproceso histórico. Puede aparecer porque llegó tarde una fuente, se corrigieron etiquetas, cambió un parser, se rehidrató una tabla o se recalculó una feature. En ciencia de datos para IA, un backfill no es una tarea inocente: puede cambiar el dataset sobre el que se entrenó, el split que se había congelado o la métrica que se comunicó.

El error común es "vuelvo a ejecutar y ya está". Eso borra la pregunta original. Si el snapshot cambia, la versión del split también debe revisarse. Si una etiqueta cambia, una métrica anterior puede quedar invalidada. Si una feature se recalcula con una fecha de disponibilidad

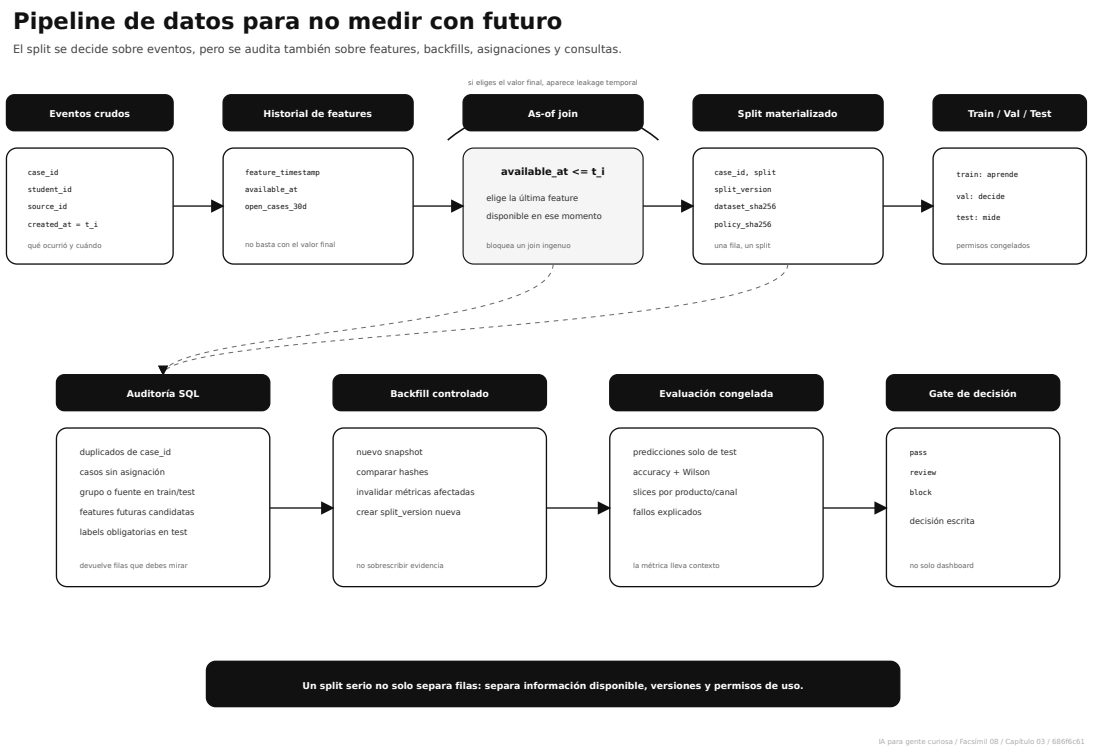
distinta, puede introducir leakage temporal aunque el código siga pasando.

Una política práctica:

CAMBIO	QUÉ HACER ANTES DE MEDIR
Llegan filas históricas tarde.	Crear nuevo snapshot y comparar asignaciones.
Cambia una etiqueta.	Marcar métricas afectadas y regenerar reporte.
Cambia una fuente documental.	Revisar grupos por `source_id` o versión.
Cambia una feature.	Auditar `available_at` y repetir as-of join.
Cambia la política de split.	Crear `split_version` nueva, no sobrescribir la anterior.
Cambia el parser o la limpieza.	Recalcular hashes y revisar duplicados/leakage textual.

Airflow documenta scheduling sensible a datasets: tareas que se disparan cuando cambian datasets upstream.[22] Eso es útil, pero no suficiente. Que un dataset cambie y dispare un pipeline no significa que la evaluación siga siendo comparable. Necesitas una decisión explícita: mantener split versionado, crear re-split o bloquear hasta revisión.

Pipeline técnico: de eventos a evaluación reproducible



Pipeline de ingeniería de datos para auditar splits: eventos, features temporales, as-of join, asignaciones materializadas, SQL, backfills y gate final.

Leakage de preprocesamiento

Una fuga clásica no aparece en el split, sino en el pipeline. Suele pasar cuando hacemos esto:

1. Cargar todo el dataset.
2. Normalizar, imputar, seleccionar variables o crear vocabulario.
3. Partir en train, validation y test.
4. Entrenar y medir.

El problema está en el paso 2. Si una transformación aprende algo del dataset completo, validation y test ya han participado en el entrenamiento, aunque el modelo principal no los haya visto.

La versión correcta es un patrón de ejecución, no una ecuación:

PASO	QUÉ SE PERMITE APRENDER	DÓNDE SE APLICA DESPUÉS
`fit` de la transformación	Parámetros de escalado, imputación, vocabulario, PCA o calibración.	Solo con `train`.
Congelar parámetros	La media, vocabulario, componentes o reglas aprendidas.	Se versionan con el pipeline.
`transform`	No aprende parámetros nuevos.	Se aplica a `train`, `validation` y `test` con lo aprendido en `train`.

La regla práctica:

TRANSFORMACIÓN	QUÉ APRENDE	DÓNDE SE HACE `FIT`	QUÉ SE APLICA A VALIDATION/TEST
Normalización	Media, desviación, mínimos o máximos.	Solo train.	`transform` con parámetros de train.
Imputación	Media, moda, percentil o modelo auxiliar.	Solo train.	Relleno con reglas aprendidas en train.
Selección de features	Variables que parecen predictivas.	Solo train, o dentro de cada fold.	Mismas features elegidas.
PCA	Componentes y proyección.	Solo train.	Misma proyección.
TF-IDF	Vocabulario e IDF.	Solo train.	Misma matriz de vocabulario.
Oversampling	Ejemplos sintéticos o pesos.	Solo train.	Nunca se sintetiza test.

La frase que debería quedar pegada a la pantalla: primero partes, luego aprendes transformaciones.

En el cuaderno del facsímil, esta idea se convierte en un script ejecutable: `ops/preprocessing_fit_audit.py`. El script no entrena un modelo grande; hace algo más pequeño y muy útil para aprender: compara el vocabulario que obtendrías si ajustas un vectorizador con solo train frente al vocabulario que obtendrías usando todo el dataset. Si el vocabulario global aprende palabras que solo aparecen en validation o test, acabas de demostrar una fuga de preprocesamiento con un ejemplo que cabe en una terminal.

Por ejemplo, con el split recomendado del cuaderno, el vectorizador ajustado solo con train no conoce algunos términos que aparecen en test, como `convocatoria`, `abonado`, `confirmacion` o `bloqueada`. Eso es normal: test representa futuro o datos reservados. Si ajustas el vectorizador con todo el dataset antes de partir, esas palabras entran en el vocabulario desde

el principio. El modelo todavía no ha visto la etiqueta de test, pero su representación ya fue preparada con información del test. En texto, ese detalle es importante porque vocabulario, IDF, normalizadores y reglas de tokenización forman parte de lo que el sistema aprende.

Puedes ejecutarlo así:

```
# Desde la carpeta extraída del ZIP:
python3 ops/split_audit.py --write
python3 ops/preprocessing_fit_audit.py --write
cat output/preprocessing_fit_decision.md
```

La salida no pretende decir “este vocabulario es malo”. Dice algo más concreto: si quieres medir honestamente, el `fit` del vectorizador debe ocurrir después del split y solo con train. Luego aplicas ese mismo vectorizador a validation y test. Ese patrón se traslada igual a `StandardScaler`, imputadores, PCA, selección de features, calibración, oversampling o cualquier transformación que aprenda parámetros.

RAG y LLM eval también tienen splits

En un clasificador tabular, solemos pensar en filas. En RAG y evaluación de LLMs hay más piezas:

OBJETO	POR QUÉ PUEDE CONTAMINAR LA EVALUACIÓN
Documento	El mismo documento puede generar chunks, preguntas y respuestas en varios splits.
Chunk	Dos chunks hermanos pueden contener casi la misma evidencia.
Pregunta sintética	Si se genera desde un documento, hereda su grupo.
Respuesta esperada	Si acaba en trazas de desarrollo, el test deja de ser final.
Prompt few-shot	Ejemplos demasiado parecidos al test facilitan la tarea artificialmente.
Índice de recuperación	Si mezcla corpus de desarrollo y test, la medición no separa conocimiento.
Herramientas y trazas	Logs de ejecución pueden contener soluciones, IDs o rutas documentales.

Para RAG, la regla no debería ser “parto preguntas”. Debería ser un contrato documental:

SI COMPARTEN...	DEBEN QUEDARSE JUNTOS CUANDO MIDES...	EJEMPLO
`document_id`	Generalización a documentos nuevos.	Chunks hermanos del mismo PDF.
`source_id` o URL canónica	Recuperación sobre fuentes no vistas.	Varias preguntas generadas desde la misma normativa.
Versión temporal	Robustez ante cambios de documento.	Reglamento 2025 frente a reglamento 2026.

Si dos chunks vienen del mismo documento, no deberían terminar uno en train y otro en test salvo que la evaluación esté diseñada explícitamente para medir recuperación sobre documentos compartidos. Y si ese caso existe, debe declararse, porque ya no estamos midiendo generalización documental.

En evaluación de LLMs hay otra trampa: mirar fallos de test para mejorar el prompt. La primera vez puede ser análisis. La segunda empieza a parecer ajuste. Si cambiamos el prompt, el modelo, el formato de salida, el sistema de herramientas o la rúbrica porque vimos test, test ya no es final. Necesitamos validation o un nuevo holdout.

Un ejemplo operativo: supón que tienes 200 preguntas para evaluar un RAG de normativa universitaria. Si generas 100 preguntas desde un documento de becas y luego repartes esas preguntas al azar, algunas acabarán en train, otras en validation y otras en test. El modelo o el equipo no ven exactamente la misma pregunta, pero sí ven el mismo documento, los mismos términos y a veces la misma respuesta escrita de otra manera. Para medir generalización documental, eso no vale. El grupo no debería ser la pregunta: debería ser el documento, la versión del documento o la fuente canónica.

Otro caso frecuente: usas ejemplos few-shot en el prompt. Esos ejemplos también tienen que salir de train o validation, no del test. Si al mirar el test descubres que el modelo falla en “matrícula bloqueada” y copias un ejemplo parecido al prompt, has convertido el test en una herramienta de ajuste. La práctica correcta no es dejar de aprender del error; es mover esa decisión a validation y reservar un nuevo holdout si quieres comunicar una cifra final.

Validación cruzada avanzada

La validación cruzada no arregla por sí sola un split mal pensado. Solo repite una regla de partición varias veces. Si la regla parte grupos o mezcla futuro, el problema se repite en cada fold.

MÉTODO	CUÁNDO SIRVE	RIESGO SI SE USA SIN PENSAR
`KFold`	Datos independientes y suficientes.	Parte entidades repetidas.
`StratifiedKFold`	Clasificación con clases desbalanceadas.	Conserva etiquetas, pero puede mezclar grupos.
`GroupKFold`	Hay usuarios, estudiantes, clientes, pacientes, documentos o dispositivos.	Puede dejar folds con etiquetas descompensadas.
`StratifiedGroupKFold`	Necesitas grupos no compartidos y proporciones de clases razonables.	No siempre puede satisfacer ambas cosas si el dataset es pequeño.
`TimeSeriesSplit`	Evaluación hacia futuro.	No protege entidades repetidas por sí solo.
Validación cruzada anidada	Separar selección de hiperparámetros y estimación final.	Es más cara y exige disciplina con pipelines.

La validación cruzada anidada se usa cuando queremos seleccionar hiperparámetros sin engañar la estimación. La idea operativa se entiende mejor como dos bucles con papeles distintos:

BUCLE	PAPEL	QUÉ NO DEBE HACER
Externo	Estimar rendimiento final en folds que no han decidido la configuración.	Elegir hiperparámetros mirando su propio test.
Interno	Elegir hiperparámetros dentro del train de cada fold externo.	Usar datos del fold externo reservado.

Si usas el mismo fold para elegir y medir, el resultado tiende a ser optimista. No porque nadie haya hecho nada extraño, sino porque el proceso de selección ya miró esa señal.

Negativos difíciles, ranking y recuperación

En RAG, ranking y clasificación semántica, los ejemplos negativos importan mucho. Un negativo fácil es un caso claramente distinto. Un negativo difícil es uno que se parece mucho al positivo pero requiere distinguir un detalle.

CASO	POSITIVO	NEGATIVO DIFÍCIL
Beca	“requisitos de beca general”	“requisitos de beca de movilidad”
Pago	“justificante de pago pendiente”	“pago duplicado reclamado”
Matrícula	“plazo ordinario de matrícula”	“ampliación de matrícula fuera de plazo”

Los negativos difíciles enseñan y evalúan mejor, pero también pueden contaminar si se generan mezclando splits. La regla operativa:

1. Si generas negativos antes de partir, agrupa por entidad y fuente para que no crucen splits.
2. Si generas negativos después de partir, hazlo dentro de cada split.
3. Si usas embeddings para buscar negativos, no uses el índice de test para seleccionar ejemplos de train.
4. Si ajustas un reranker, mide con consultas y documentos que no hayan participado en su selección.

Para un retriever, una evaluación mínima debería reportar:

MÉTRICA	QUÉ PREGUNTA RESPONDE
`recall@k`	¿El documento correcto aparece entre los k primeros?
`MRR`	¿A qué altura aparece el primer resultado útil?
Precisión de contexto	¿Cuánto contexto recuperado es realmente relevante?
Cobertura por fuente	¿El sistema recupera bien todos los tipos de documento?
Latencia	¿La recuperación cabe en el presupuesto operativo?

Este punto prepara el capítulo de embeddings y los capítulos de RAG: la evaluación de recuperación también empieza por un split honesto.

Tamaño mínimo, intervalos y slices

Un test pequeño puede ser honesto y aun así poco informativo. Si medimos accuracy con 20 ejemplos, pasar de 15 a 16 aciertos cambia la métrica de 75% a 80%. Eso no es una mejora estable; puede ser ruido muestral.

Para una proporción binomial, como accuracy cuando cada caso cuenta como acierto o fallo, el intervalo de Wilson suele ser más prudente que el intervalo normal simple cuando el test es pequeño.[23]

$$IC_{Wilson} = \frac{\hat{p} + \frac{z^2}{2n} \pm z \sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + \frac{z^2}{n}}$$

SÍMBOLO	SIGNIFICADO
$\hat{p}$	Métrica como proporción: accuracy, tasa de acierto, recall binario.
n	Número de ejemplos del test.
z	Cuantil normal asociado al nivel de confianza; para 95% se usa aproximadamente 1.96.
IC_{Wilson}	Intervalo de Wilson para la proporción.

Ejemplo del cuaderno: si $\hat{p} = 0.75$ y $n = 4$, el intervalo de Wilson al 95% queda aproximadamente entre 0.30 y 0.95. Ese rango es enorme. La lectura no es “el sistema está entre 30% y 95% y ya está”, sino “con cuatro casos no hay base suficiente para cerrar una afirmación fuerte”.

Para métricas más complejas, bootstrap suele ser una opción práctica: remuestrear el test muchas veces y observar la distribución de la métrica.[24] Pero incluso bootstrap necesita una pregunta honesta: si el test está contaminado, remuestrear no arregla la contaminación.

Además de mirar la media global, hay que mirar slices. Un sistema puede tener buen resultado global y fallar justo en el segmento que importa.

SLICE	POR QUÉ MIRARLO
`product`	Matrícula, becas, pagos, horarios o prácticas pueden tener dificultad distinta.
`channel`	Portal, email y chat no generan el mismo lenguaje.
`label`	Las clases raras suelen desaparecer en tests pequeños.
`source_id`	Algunas fuentes documentales son más ambiguas.
Fecha	Cambios de periodo pueden alterar vocabulario y reglas.
Criticidad	No todos los errores cuestan lo mismo.
Idioma o región	Si existen, pueden cambiar distribución y estilo.

La pregunta técnica no es solo “¿cuánto acierta?”. También es: “¿dónde falla, con cuánta evidencia y qué decisión puedo tomar sin engañarme?”.

El segundo script práctico del cuaderno trabaja justo esta parte: `ops/evaluate_test_slices.py`. Lee `data/model_predictions.csv`, toma del manifiesto los IDs asignados a test y evalúa solo esos casos. Esto es importante: no deja que se cuele una predicción de train o validation en la métrica final. Después calcula accuracy, intervalo de Wilson, latencia y slices por `product`, `channel` y `label`.

Con el dataset pequeño del capítulo, el resultado esperado es deliberadamente incómodo: accuracy de 0.75 con solo 4 casos de test, un intervalo de Wilson muy ancho, un fallo en el caso `s023` y una decisión `review_test_too_small`. Eso enseña una lección que muchos proyectos aprenden tarde: una métrica global puede sonar bien y seguir sin ser suficiente para tomar una decisión fuerte. Si el único caso `escalate` del test falla, la media global no cuenta toda la historia. Si cada producto tiene un único ejemplo, ningún slice tiene evidencia suficiente. El reporte no bloquea por capricho; te obliga a mirar la métrica con tamaño, contexto y coste del error.

Puedes ejecutarlo así:

```
# Desde la carpeta extraída del ZIP:
python3 ops/evaluate_test_slices.py --write
cat output/evaluation_slice_decision.md
python3 -m json.tool output/evaluation_slice_report.json
```

Este ejemplo sí se puede reutilizar. Cambias `data/model_predictions.csv` por las predicciones de tu modelo, mantienes `case_id`, `predicted_label`, `confidence` y `latency_ms`, y el script usa el manifiesto para decidir qué pertenece a test. Si tu proyecto mide otra cosa, cambias la función de métrica, pero mantienes el contrato: IDs versionados, split congelado, slices explícitos y decisión documentada.

Herramientas y patrones que sí ayudan

En este capítulo no basta con decir “usa una librería”. Las herramientas ayudan cuando codifican la política correcta. Si la política está mal, la herramienta solo ejecuta más rápido el error.

NECESIDAD	HERRAMIENTA O PATRÓN	QUÉ APORTA	QUÉ DEBES VIGILAR
Partir datos tabulares o arrays	`train_test_split`, `GroupKFold`, `StratifiedGroupKFold`, `TimeSeriesSplit`	Implementa estrategias conocidas y reproducibles.	La función no sabe cuál es tu unidad experimental real.
Evitar leakage de preprocesamiento	`Pipeline` y transformadores con `fit` / `transform`	Encadena pasos para que cada fold entrene transformadores solo con train.	Un `fit` manual fuera del pipeline puede volver a contaminarlo todo.
Versionar datos y manifiestos	DVC, lakeFS, hashes propios o registro interno	Permite reconstruir qué snapshot produjo qué métrica.	Versionar archivos no sustituye escribir una política de split.
Auditar como tabla	DuckDB, SQL en almacén analítico, dbt data tests	Convierte reglas de split en consultas que devuelven filas problemáticas.	Un test SQL mal planteado puede pasar aunque la pregunta de evaluación sea equivocada.
Orquestar cambios	Dagster asset checks, Airflow data-aware scheduling	Conecta cambios de datasets con checks y pipelines.	Reejecutar no basta: un backfill puede exigir re-split o invalidar métricas.
Construir features temporales	Feast, Tecton, Databricks Feature Store	Ayuda a hacer as-of joins y point-in-time correctness.	Si `available_at` está mal definido, la herramienta no puede adivinar la realidad.
Registrar evaluaciones	MLflow Dataset Tracking o evaluation datasets	Conecta dataset, run, métrica y artefactos de evaluación.	Si el test se usó para decidir, el tracking debe decirlo.
Integrar en CI/CD	`make test`, GitHub Actions, GitLab CI, Buildkite	Bloquea cambios cuando faltan outputs, predicciones o manifiesto.	Un CI que solo mira que el script termina no evalúa calidad semántica.

El patrón profesional es este: primero escribes la pregunta de evaluación y la política de split; después eliges herramienta. Si empiezas por la herramienta, acabarás aceptando sus valores por defecto como si fueran criterio científico.

Política de test y holdout final

La política sencilla:

SPLIT	PERMISO
Train	Aprender.
Validation	Elegir.
Test	Medir una decisión cerrada.
Holdout final	Confirmar una conclusión importante.

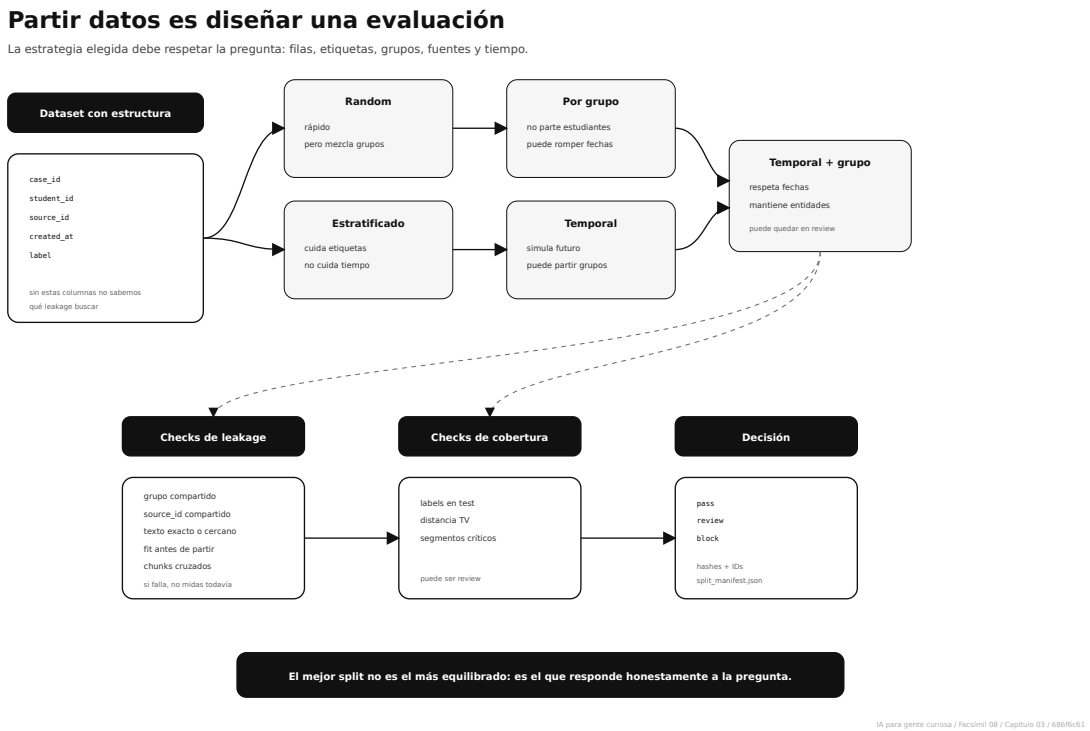
Cuando test se usa para decidir, deja de ser test final. En un proyecto real eso no debería vivirse como culpa, sino como trazabilidad: se anota y se crea una nueva partición final si hace falta.

Un ejemplo concreto:

1. Entrenas tres modelos con train.
2. Eliges el mejor con validation.
3. Mides una vez con test.
4. Ves que falla en “becas”.
5. Cambias el prompt, el chunking o el retriever.
6. La siguiente medición ya no debería venderse como test puro.

La solución profesional es reservar un holdout final para decisiones importantes o crear una nueva versión de evaluación. No siempre hace falta en un ejercicio pequeño; sí hace falta cuando vas a comunicar rendimiento, comparar sistemas o tomar una decisión con impacto.

Anatomía de una partición que no se engaña



Una partición seria combina estrategia, checks de leakage, cobertura y una decisión documentada.

En el día a día

Un equipo profesional debería discutir el split antes de mirar resultados. Si elegimos estrategia después de ver métricas, ya estamos contaminando la decisión. La pregunta correcta es: “¿qué uso real queremos simular?”.

SITUACIÓN	SPLIT RAZONABLE	POR QUÉ
Tickets de los mismos clientes	Por cliente o cuenta.	Evita que el sistema vea estilos repetidos.
Documentación viva	Temporal por versión.	Evalúa contra futuro documental.
Datasets médicos o académicos por persona	Por sujeto o estudiante.	Evita que la misma entidad cruce particiones.
Clases raras	Estratificado con revisión manual.	Evita perder casos críticos en test.
Logs de producto	Temporal.	Simula despliegue hacia adelante.

La regla profesional: si una entidad puede generar varias filas parecidas, no partas por fila hasta demostrar que no contamina.

Por qué debería importarte

Una mala partición puede hacer que un modelo mediocre parezca excelente. También puede hacer que un modelo bueno parezca inestable si el test no cubre clases importantes. En ambos casos, la decisión técnica se apoya en una medición defectuosa.

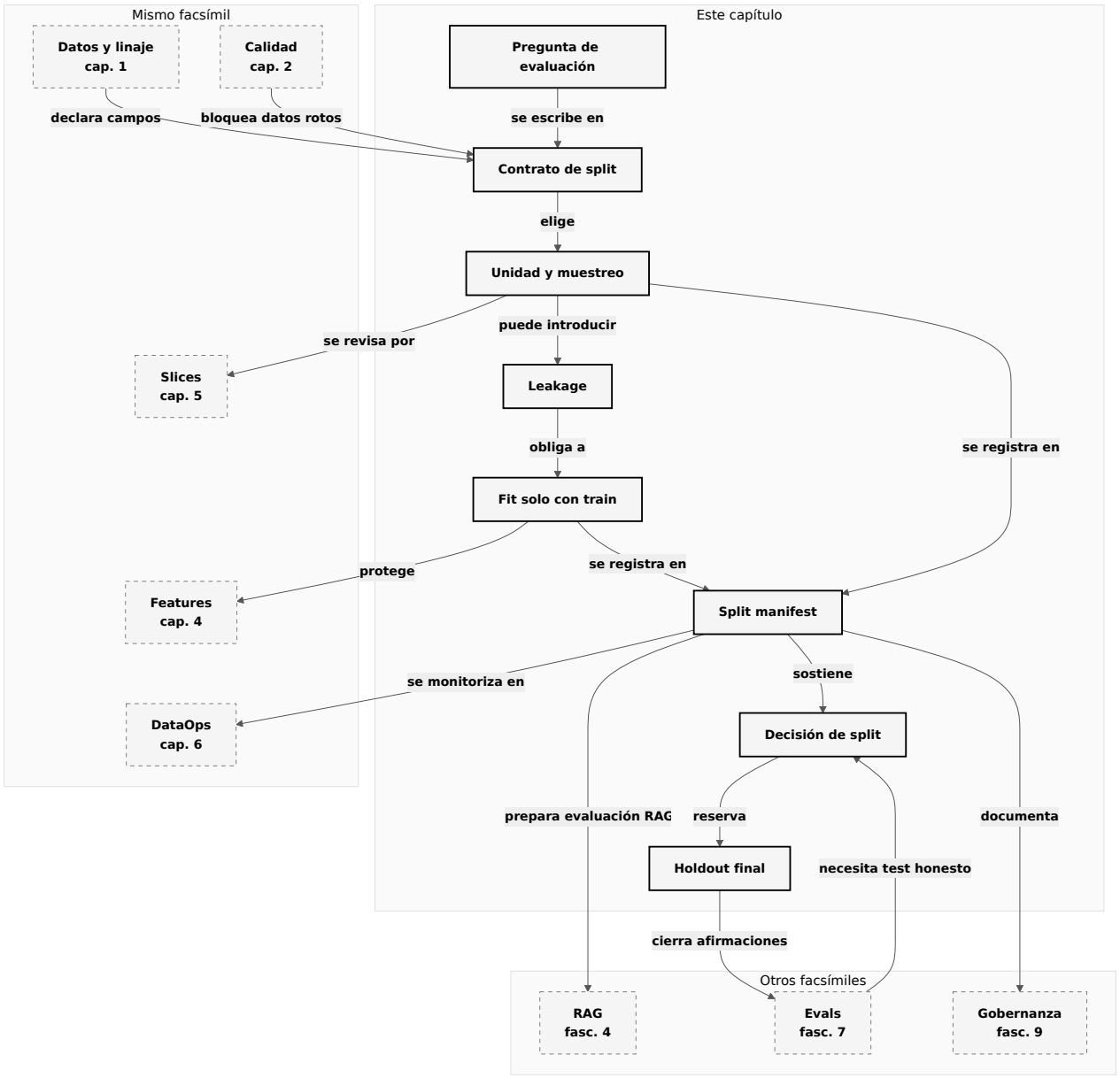
Este capítulo importa porque casi todo lo que sigue depende de aquí: features, embeddings, slices, drift, análisis causal y gobernanza. Si test no es honesto, los capítulos posteriores están midiendo sobre una base torcida.

Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Confundir porcentaje con validez	60/20/20 parece profesional.	Preguntar qué decisión simula cada split.
Estratificar y olvidarme de grupos	Las clases quedan bonitas.	Revisar `student_id`, cliente, fuente o documento.
Hacer split después de transformar	El pipeline parece más cómodo.	Separar primero; ajustar transformaciones solo en train.
Mirar test demasiadas veces	Es tentador ajustar contra la métrica final.	Usar validation para decisiones y test para cierre.
Ignorar el tiempo	Random parece estable.	Usar split temporal si producción será futura.
Partir preguntas RAG sin agrupar documentos	Parece que cada pregunta es independiente.	Agrupar por documento, versión, fuente o URL canónica.
Hacer negativos difíciles con todo el índice	Sale una evaluación más dura, pero mezclada.	Generarlos dentro de cada split o con grupos congelados.
No guardar manifiesto	El reporte parece suficiente.	Versionar hashes, política, estrategia e IDs por split.
Celebrar una métrica con test pequeño	El número tiene demasiada varianza.	Acompañar con tamaño, intervalo y slices.

Cómo encaja todo

Este capítulo conecta calidad de datos con evaluación. El capítulo 02 detectaba fallos dentro del dataset; este capítulo pregunta si la partición permite medir sin engañarse. El capítulo 04 usará estos splits para hablar de features y embeddings: ahí veremos que una transformación también puede introducir leakage si se ajusta con todo el dataset.


Mismo facsímil
Datos y linaje cap. 1
Calidad cap. 2
declara campos
bloquea datos rotos
Slices cap. 5
Features cap. 4
DataOps cap. 6
Este capítulo
Pregunta de evaluación
se escribe en
Contrato de split
elige
Unidad y muestreo
se revisa por
puede introducir
Leakage
obliga a
Fit solo con train
protege
se registra en
se registra en
Split manifest
se monitoriza en
sostiene
Decisión de split
prepara evaluación RAG
reserva
documenta
Holdout final
necesita test honesto
cierra afirmaciones
Otros facsímiles
RAG fasc. 4
Evals fasc. 7
Gobernanza fasc. 9

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Split	Partición de un dataset con finalidad concreta.
Train	Split que puede usarse para aprender o ajustar.
Validation	Split para elegir decisiones durante desarrollo.
Test	Split reservado para medir al final.
Holdout	Conjunto separado que no se usa durante ajuste.
Estratificación	Mantener proporciones de etiquetas o segmentos.
Split por grupo	Mantener todas las filas de una entidad juntas.
Split temporal	Respetar orden cronológico.
Leakage	Información indebida que contamina la evaluación.
Purga	Separar o retirar ejemplos demasiado cercanos.
Distancia TV	Medida de diferencia entre distribuciones.
Manifiesto de split	Documento técnico con hashes, estrategia, claves, permisos y asignaciones.
Leakage de preprocesamiento	Fuga causada por transformaciones ajustadas antes de partir o usando splits reservados.
RAG leakage	Fuga en documentos, chunks, preguntas, respuestas esperadas, prompts o índices.
Negativo difícil	Ejemplo muy parecido al positivo, pero con etiqueta o respuesta distinta.
Holdout final	Partición reservada para confirmar una conclusión después de cerrar decisiones.
Slice	Subconjunto relevante del test: producto, canal, etiqueta, fuente, fecha o criticidad.
Intervalo de Wilson	Intervalo de confianza para una proporción binomial, útil cuando el test es pequeño.
Pipeline de preprocesamiento	Cadena de transformaciones donde cada paso que aprende parámetros debe hacer `fit` solo con train.
Dataset tracking	Registro de datasets, hashes, splits, runs y resultados para poder reconstruir una evaluación.

TÉRMINO	DEFINICIÓN BREVE
Point-in-time correctness	Garantía de que cada feature usada estaba disponible en el momento de decisión del caso.
As-of join	Unión temporal que elige el último valor válido antes o en la fecha del evento.
Split materializado	Tabla versionada que asigna cada caso a un único split.
Backfill	Reproceso histórico que puede cambiar snapshots, features, etiquetas o métricas.
Re-split	Nueva versión de partición creada cuando cambia el dato, la política o la pregunta de evaluación.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Por qué un split no es solo un porcentaje?
2. ¿Qué diferencia hay entre train, validation y test?
3. ¿Cuándo usarías split por grupo?
4. ¿Cuándo usarías split temporal?
5. ¿Por qué una estrategia estratificada puede seguir contaminada?
6. ¿Qué mide Jaccard al detectar duplicados entre splits?
7. ¿Qué significa `future_train_vs_test`?
8. ¿Por qué `time_group_holdout` queda en `review` y no en `pass`?
9. ¿Qué artefacto explica los hallazgos de leakage?
10. ¿Cómo evitarías leakage de preprocesamiento?
11. ¿Por qué un documento RAG puede contaminar varias preguntas?
12. ¿Qué pasa si miras test para cambiar el prompt?
13. ¿Qué diferencia hay entre validation, test y holdout final?
14. ¿Qué debería guardar un `split_manifest.json`?
15. ¿Cuándo usarías `StratifiedGroupKFold` en vez de `KFold`?
16. ¿Por qué una métrica global puede esconder fallos por slice?
17. ¿Por qué usamos Wilson cuando el test es pequeño?
18. ¿Qué demuestra `preprocessing_fit_audit.py`?
19. ¿Por qué `evaluate_test_slices.py` filtra por IDs del manifiesto?

20. ¿Qué significa que una feature cumpla `available_at <= created_at`?
21. ¿Por qué un as-of join protege contra leakage temporal?
22. ¿Qué comprueba `dataset_split_assignments.csv` que el manifiesto no muestra tan cómodamente?
23. ¿Qué harías si un backfill cambia etiquetas o features históricas?
24. ¿Qué mirarías en `sql_split_audit_decision.md` antes de confiar en el split?
25. ¿Qué cambiarías en `model_predictions.csv` para usarlo con tu propio modelo?

En resumen

IDEA	QUÉ TE LLEVAS
Un split mide una pregunta.	La estrategia depende del uso real.
Random no siempre es inocente.	Puede mezclar grupos, fuentes y tiempo.
Estratificar no basta.	Conserva labels, pero no protege entidades.
El tiempo cambia la evaluación.	Entrenar con futuro infla resultados.
La mejor estrategia puede quedar en review.	Evitar leakage no garantiza cobertura perfecta.
El split debe versionarse.	Asignaciones, hallazgos y decisión son artefactos.
Preprocesar también aprende.	Todo `fit` que aprende parámetros debe usar train.
RAG añade nuevas fronteras.	Documentos, chunks, prompts y respuestas esperadas también se separan.
Test no decide.	Si test decide, necesitas validation nueva o holdout final.
Los negativos difíciles se controlan.	Son útiles, pero deben respetar grupos y particiones.
El tamaño importa.	Un test pequeño necesita intervalo, cautela y lectura por slices.
Las herramientas no deciden por ti.	`Pipeline`, DVC o MLflow ayudan si la política de split ya está bien escrita.
Los ejemplos deben ejecutarse.	El cuaderno genera reportes de split, preprocesado y evaluación por slices.
Las features también tienen tiempo.	Point-in-time correctness evita entrenar con información que aún no existía.
El split debe poder consultarse.	Una tabla materializada permite checks SQL, CI y revisión por pares.
Los backfills cambian la evidencia.	Si cambia el snapshot, quizá necesitas re-split o invalidar métricas anteriores.

Para saber más

Kapoor, S. y Narayanan, A. (2023). Leakage and the Reproducibility Crisis in Machine-Learning-Based Science. *Patterns*, 4(9), 100804. 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

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S. y Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020*. Paper

Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158), 209-212. DOI

DVC. (2026). What is DVC?

MLflow. (2026). Dataset Tracking

MLflow. (2026). Evaluation Dataset Concepts

Apache Airflow. (2026). Data-aware scheduling

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387. DOI

Databricks. (2026). Point-in-time feature joins

Dagster. (2026). Testing assets with asset checks

dbt Labs. (2026). Add data tests to your DAG

DuckDB. (2026). CSV Loading

Feast. (2026). Point-in-time joins

Tecton. (2026). Construct Training Data

scikit-learn. (2026). GroupKFold

scikit-learn. (2026). Pipeline

scikit-learn. (2026). StratifiedGroupKFold

scikit-learn. (2026). TimeSeriesSplit

scikit-learn. (2026). train_test_split

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

---

Notas

1. scikit-learn. (2026). *train_test_split*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
2. scikit-learn. (2026). *GroupKFold*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
3. scikit-learn. (2026). *StratifiedGroupKFold*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
4. scikit-learn. (2026). *TimeSeriesSplit*. Documentación oficial. Consultado el 6 de junio de 2026. ↩
5. 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. ↩
6. Kapoor, S. y Narayanan, A. (2023). Leakage and the Reproducibility Crisis in Machine-Learning-Based Science. *Patterns*, 4(9), 100804. DOI. ↩
7. Lewis, P. y otros (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020*. Paper. ↩
8. scikit-learn. (2026). *Pipeline*. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html. Consultado el 22 de junio de 2026. ↩
9. scikit-learn. (2026). *Common pitfalls and recommended practices*. https://scikit-learn.org/stable/common_pitfalls.html. Consultado el 22 de junio de 2026. ↩
10. DVC. (2026). *What is DVC?* https://dvc.org/doc/user-guide/what-is-dvc. Consultado el 6 de junio de 2026. ↩
11. MLflow. (2026). *Dataset Tracking*. https://mlflow.org/docs/latest/ml/dataset/. Consultado el 22 de junio de 2026. ↩
12. MLflow. (2026). *Evaluation Dataset Concepts*. https://mlflow.org/docs/latest/genai/concepts/evaluation-datasets/. Consultado el 22 de junio de 2026. ↩
13. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. ↩
14. Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37, 547-579. ↩

.5. Databricks. (2026). *Point-in-time feature joins*. https://docs.databricks.com/aws/en/machine-learning/feature-store/time-series. Consultado el 22 de junio de 2026. ↩

.6. Tecton. (2026). *Construct Training Data*. https://docs.tecton.ai/docs/reading-feature-data/reading-feature-data-for-training/constructing-training-data. Consultado el 22 de junio de 2026. ↩

.7. Feast. (2026). *Point-in-time joins*. https://docs.feast.dev/getting-started/concepts/point-in-time-joins. Consultado el 22 de junio de 2026. ↩

.8. Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 377-387. https://doi.org/10.1145/362384.362685 ↩

.9. dbt Labs. (2026). *Add data tests to your DAG*. https://docs.getdbt.com/docs/build/data-tests. Consultado el 22 de junio de 2026. ↩

!0. Dagster. (2026). *Testing assets with asset checks*. https://docs.dagster.io/guides/test/asset-checks. Consultado el 22 de junio de 2026. ↩

!1. DuckDB. (2026). *CSV Loading*. https://duckdb.org/docs/stable/data/csv/overview. Consultado el 22 de junio de 2026. ↩

!2. Apache Airflow. (2026). *Data-aware scheduling*. https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/datasets.html. Consultado el 22 de junio de 2026. ↩

!3. Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158), 209-212. https://doi.org/10.1080/01621459.1927.10502953 ↩

!4. Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1), 1-26. https://doi.org/10.1214/aos/1176344552 ↩