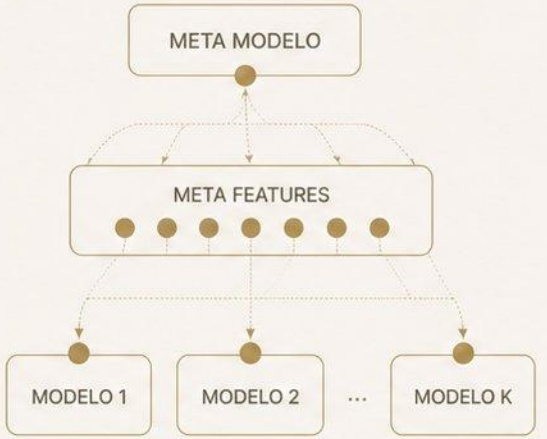
UNIVERSIDAD
Panamericana

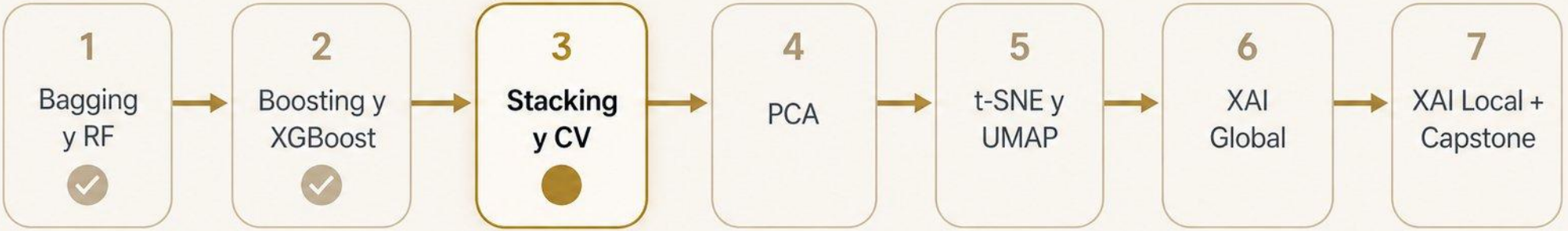
Clase 3: Stacking, Tuning y Validación Cruzada en Series de Tiempo

Cerrando el bloque de ensembles antes de pivotear a métodos no supervisados



UNIVERSIDAD Panamericana

¿Dónde estamos en el curso?



Tenemos un Random Forest. Tenemos un XGBoost. Hoy: cómo combinarlos en algo mejor que cualquiera de los dos por separado, cómo tunearlos en serio, y — lo más importante para datos financieros — cómo validar sin engañarnos.

Una validación incorrecta es peor que no tener validación. Te da una falsa confianza, y los modelos en producción fallan por razones que la métrica de validación no detectó.

Plan de la sesisión

1 1. Stacking y blending (30 min)

Cómo combinar modelos heterogéneos. Cuándo funciona, cuándo no, y dónde están las trampas de leakage.

2 2. Hyperparameter tuning (30 min)

Grid vs random vs Bayesian (Optuna). Por qué grid es casi siempre la peor opción cuando tienes más de 3 hiperparámetros.

3 3. Validación cruzada en series de tiempo (25 min)

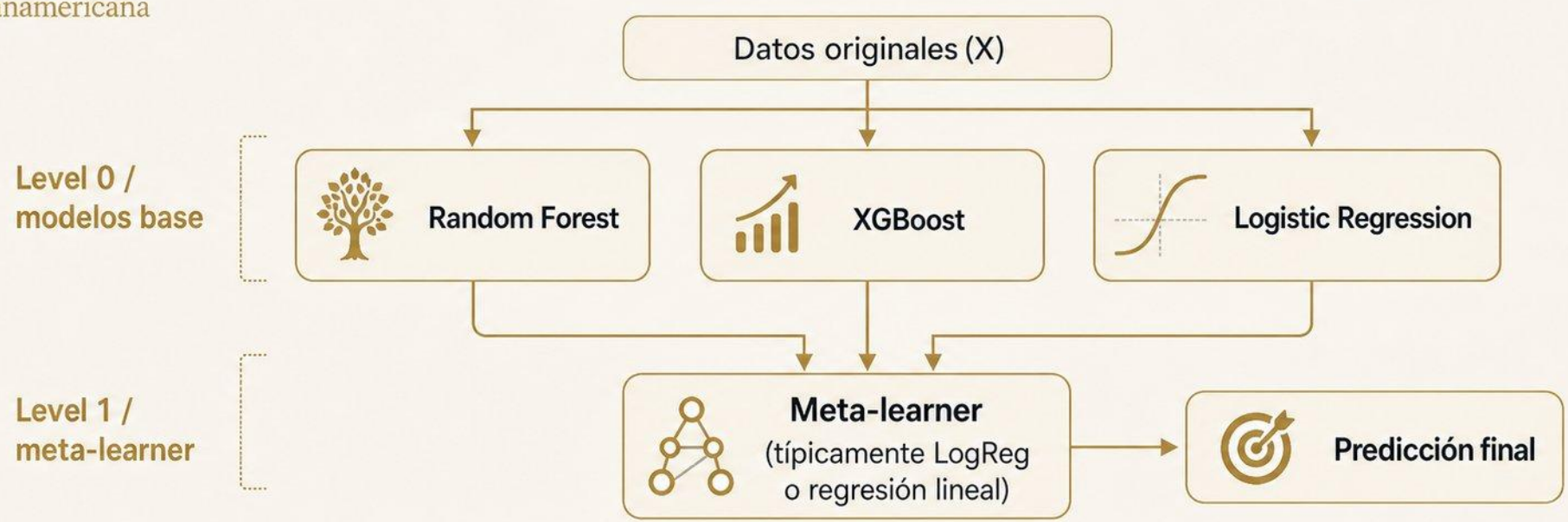
Finanzas MX

K-fold estándar miente cuando los datos tienen estructura temporal. Walk-forward, purged K-fold, embargo. Esto es crítico para datos financieros.

Colab: implementan stacking correcto sobre el dataset de crédito (RF de Clase 1 + XGBoost de Clase 2 → meta-learner LogReg), y comparan K-fold estándar vs walk-forward sobre datos con índice temporal.

UNIVERSIDAD Panamericana

Clase 3: Stacking: la idea en una imagen



Los modelos del level 0 ven los datos originales (X). El meta-learner del level 1 ve las predicciones de los modelos base como features. Aprende cómo ponderar/combinar las opiniones de los modelos base.

Funciona porque los modelos base se equivocan en lugares distintos. El meta-learner aprende cuándo confiar en quién.

UNIVERSIDAD Panamericana

Clase 3: Por qué stacking puede mejorar sobre cualquier modelo individual

Stacking funciona cuando los modelos base son diversos — cuando capturan distintos patrones en los datos. Si los tres modelos base se equivocan en exactamente las mismas filas, no hay nada que el meta-learner pueda hacer.

1. **Distintas familias de modelos.**
   RF (árboles, no-paramétrico, captura interacciones), LogReg (lineal, paramétrico, captura efectos directos), redes neuronales (representaciones aprendidas). Cada uno tiene sesgo inductivo distinto.

2. **Distintos hiperparámetros del mismo modelo.**
   XGBoost con max_depth=3 y XGBoost con max_depth=8 capturan estructuras distintas. Un meta-learner puede aprender cuándo el shallow es mejor y cuándo el deep.

3. **Distintos subsets de features.**
   Modelos entrenados sobre distintas columnas (ej: variables crediticias vs comportamiento transaccional) ven el problema desde ángulos distintos.

Si tus tres modelos base correlacionan a 0.99 en sus predicciones, stacking no va a ayudar. Mide la correlación de las predicciones antes de gastar tiempo en stacking.

UNIVERSIDAD Panamericana

MAL

Clase 3: La trampa: cómo NO hacer stacking

Versión naïve, ampliamente vista en notebooks de tutorial. Tiene leakage.

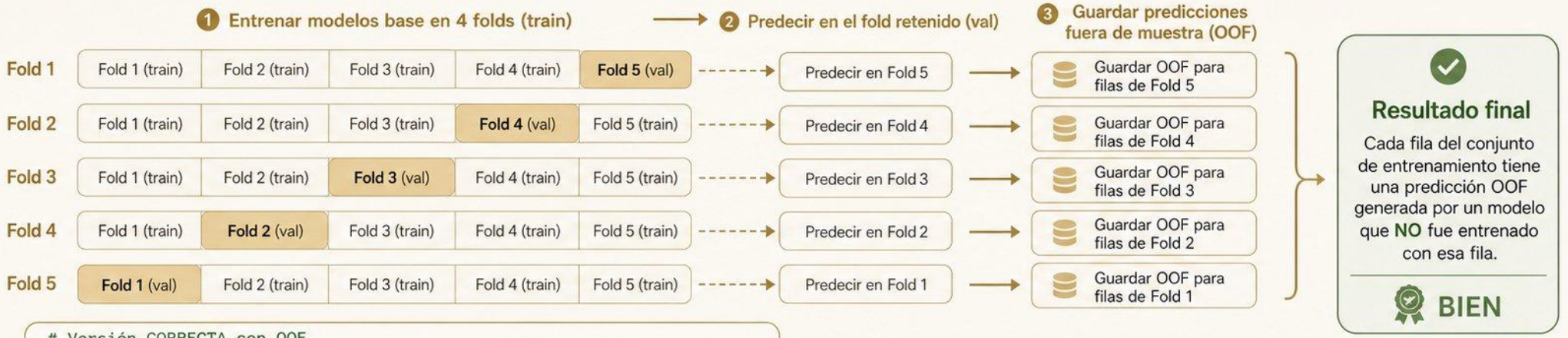
```
# Versión INCORRECTA con leakage
X_train, y_train = ...

# Entrenar modelos base sobre TODO X_train
rf.fit(X_train, y_train)
xgb.fit(X_train, y_train)

# Generar predicciones de los modelos base SOBRE LOS MISMOS DATOS
rf_preds = rf.predict_proba(X_train)
xgb_preds = xgb.predict_proba(X_train)

# Entrenar meta-learner usando esas predicciones como features
meta_X = stack([rf_preds, xgb_preds])
meta_model.fit(meta_X, y_train)
```

¿Qué está mal?

El problema: los modelos base ya 'vieron' las filas de **X**_train durante su entrenamiento. Sus predicciones sobre **X**_train son demasiado optimistas — están sobreajustadas. El meta-learner aprende a confiar en esas predicciones infladas, y cuando llegan datos reales (no vistos), todo colapsa.

Lo que se ve: validación impecable en el training set, performance pobre en producción. Es uno de los errores más comunes en pipelines de ML; pasa también en bancos serios.

Clase 3: La solución: out-of-fold (OOF) predictions




```python
# Versión CORRECTA con OOF
oof_preds_rf = zeros(n_train)
oof_preds_xgb = zeros(n_train)

for train_idx, val_idx in KFold(5).split(X_train):
    # Entrenar SOLO en train_idx
    rf.fit(X_train[train_idx], y_train[train_idx]
    xgb.fit(X_train[train_idx], y_train[train_idx])

    # Predecir en val_idx — datos que el modelo no vio
    oof_preds_rf[val_idx] = rf.predict_proba(X_train[val_idx])
    oof_preds_xgb(val_idx] = xgb.predict_proba(X_train[val_idx])

# Ahora SÍ es seguro entrenar el meta-learner con estas predicciones
meta_X = stack( [oof_preds_rf, oof_preds_xgb])
meta_model.fit(meta_X, y_train)
```

Ahora cada predicción que ve el meta-learner viene de un modelo que **no fue entrenado** en esa fila específica. Las predicciones reflejan el comportamiento real **fuera de muestra**.

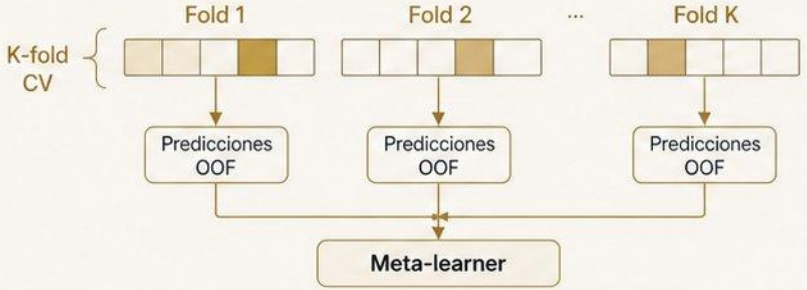
Después de generar las **OOF predictions** y entrenar el meta-learner, se reentrenan los modelos base sobre todo `X_train` para producción.
Las OOF predictions son sólo para entrenar el meta-learner.

Universidad Panamericana

Clase 3: Stacking vs Blending: dos variantes

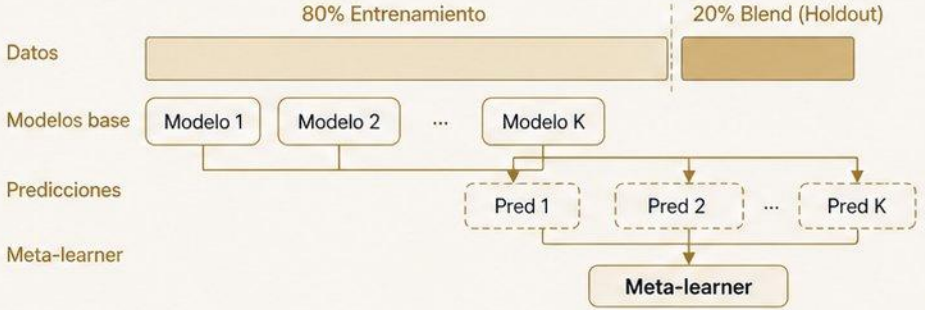
Stacking

- Usa K-fold CV para generar predicciones OOF para todo el training set
- Cada fila tiene una predicción OOF
- Entrena meta-learner sobre todas esas predicciones
- Más eficiente con datos (usa todo X_{train} para entrenar el meta-learner)
- Más caro computacionalmente (entrena cada modelo base K veces)



Blending

- Hace un único holdout split (ej: 80% train, 20% blend)
- Entrena modelos base en el 80%
- Genera predicciones sobre el 20%
- Entrena meta-learner sobre ese 20%
- Más barato (entrena cada modelo base sólo una vez)
- Menos eficiente con datos (sólo el 20% sirve para entrenar el meta-learner)



En general: stacking es más robusto y debería ser tu default. Blending es razonable cuando tus modelos base son muy caros de entrenar (ej: deep learning) y K entrenamientos son inviables.

UNIVERSIDAD Panamericana

¿Qué modelo usar como meta-learner?

El meta-learner ve, como input, las predicciones de los modelos base — un dataset pequeño (típicamente 2–10 columnas) y altamente correlacionado. La elección importa, pero menos de lo que parece.

1

Regresión logística / lineal (recomendado por default)

Simple, regularizable (Ridge, Lasso), interpretable. Los coeficientes aprendidos son las 'ponderaciones' de cada modelo base — útil para entender qué está pasando. Maneja bien la alta correlación entre predicciones de modelos base si usas regularización L2.

2

Otro modelo de árbol (XGBoost, RF)

Funciona pero es overkill — tienes 3-10 features. Más riesgo de sobreajuste sobre el meta-set.
Sólo úsalo si tienes muchas predicciones base (>20) o si quieres capturar interacciones entre las predicciones.

3

Promedio simple o ponderado a mano

No es un meta-learner real, pero es un baseline crítico. Si tu stacking sofisticado no le gana al promedio simple, algo está mal o stacking no es la solución.

Siempre comparar stacking contra promedio simple. Si el stacking te da 0.8453 AUC y el promedio te da 0.8447, no estás mejorando — estás añadiendo complejidad sin ganancia real.

Cuándo stacking NO mejora (y no debes forzarlo)

1. **Modelos base muy correlacionados.**
Si tus tres XGBoost con distintos hiperparámetros producen predicciones que correlacionan a 0.98, no hay diversidad para explotar. Solución: añadir un modelo de familia distinta (LogReg, redes), no sólo más XGBoosts.

2. **Dataset pequeño (n < 10,000).**
Stacking necesita generar OOF predictions para entrenar el meta-learner. Con datos escasos, las OOF tienen alta varianza, y el meta-learner aprende ruido. Mejor un solo modelo bien tuneado.

3. **Interpretabilidad es requerimiento (regulación, auditorías).**
Un stacked model es mucho más difícil de explicar que un único XGBoost. Si necesitas justificar decisiones a CNBV o a auditoría interna, agregar otra capa de modelo complica la conversación. Esto no es trivial en el contexto financiero mexicano.

En Kaggle, stacking gana competencias. En producción, **un único XGBoost bien tuneado y bien explicado** suele ser el mejor compromiso entre performance y mantenibilidad.

Y por supuesto, en la Clase 6 vamos a ver que el peor enemigo de la interpretabilidad es la complejidad innecesaria.

UNIVERSIDAD Panamericana

De combinar modelos a tunearlos correctamente

Stacking sólo ayuda si los modelos base son buenos individualmente.

Y para ser buenos individualmente, hay que tunearlos.
Pero tunear bien es menos obvio de lo que parece.

UNIVERSIDAD Panamericana

El problema fundamental del tuning

Un XGBoost típico tiene ~10 hiperparámetros importantes. Si probamos 5 valores de cada uno:

$$5^{10} = 9{,}765{,}625 \text{ combinaciones}$$

Si cada entrenamiento toma 5 minutos, eso son... **~93 años.** Necesitamos estrategias más inteligentes.

1. **Grid Search:**
   Probar todas las combinaciones de una grilla finita.
   Exhaustivo, simple, escala terrible.

2. **Random Search:**
   Muestrear aleatoriamente del espacio de hiperparámetros.
   Sorprendentemente competitivo (Bergstra & Bengio, 2012).

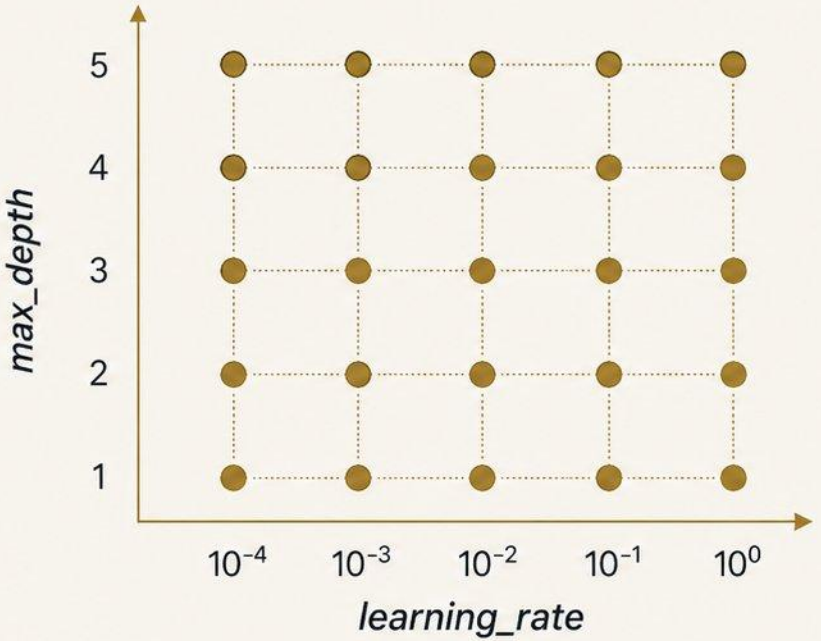
3. **Bayesian Optimization:**
   Aprender de las evaluaciones pasadas para elegir mejor las siguientes.
   Optuna, Hyperopt.

Cada uno tiene su lugar. La elección depende de cuánto cuesta cada evaluación, cuántos hiperparámetros tienes, y cuán importantes son entre sí.

UNIVERSIDAD Panamericana

Grid Search

Definir una grilla de valores para cada hiperparámetro. Probar todas las combinaciones.



Pros

- Reproducible
- Fácil de paralelizar
- Conceptualmente simple

Cons

- **Escala exponencialmente con el número de hiperparámetros**
- **Asume que sabes qué valores son razonables — eliges los puntos de la grilla a mano**
- **Desperdicia evaluaciones en regiones sub-óptimas**
- **Resolución limitada — si el óptimo está entre dos valores de la grilla, no lo vas a encontrar**

Grid search es razonable cuando: (1) tienes ≤2 hiperparámetros importantes y (2) sabes el rango aproximado del óptimo por experiencia previa. Casi nunca es ambas cosas.

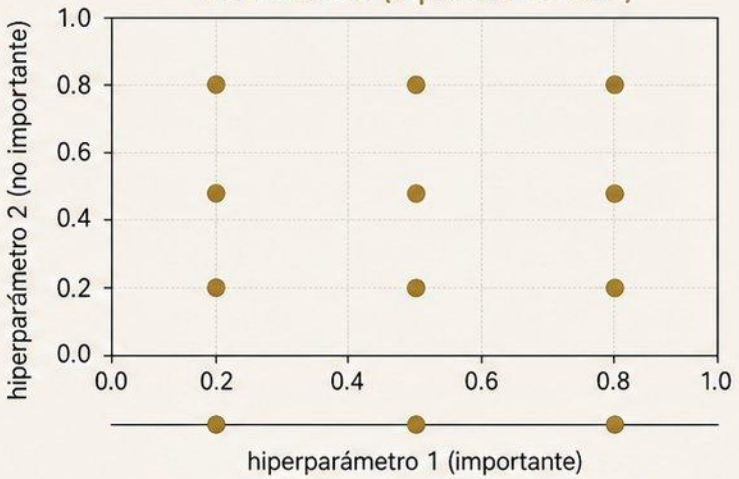
UNIVERSIDAD Panamericana

Random Search: mejor de lo que parece

Idea trivial: en lugar de muestrear de una grilla, muestrear aleatoriamente del espacio de hiperparámetros.

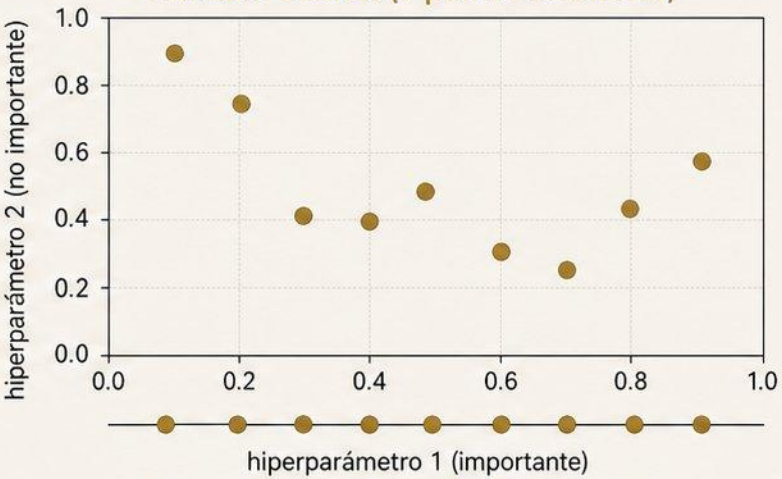
Bergstra & Bengio (2012) demostraron que esto gana a grid search en casi todos los casos prácticos. ¿Por qué?

Grid Search (9 puntos en 3×3)



Proyección marginal en eje x: sólo 3 valores únicos

Random Search (9 puntos aleatorios)



Proyección marginal en eje x: 9 valores únicos

Si sólo uno de los dos hiperparámetros realmente importa, grid search exploró sólo 3 valores de él.
Random search exploró 9. Con la misma cantidad de evaluaciones, random encuentra mejores valores **del hiperparámetro que sí importa.**

En la práctica, sólo unos pocos hiperparámetros realmente importan, pero no sabemos cuáles.
Random search distribuye exploración a lo largo de todos ellos sin pre-comprometerse.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. JMLR, 13.

UNIVERSIDAD Panamericana

Bayesian Optimization: aprender de las evaluaciones previas

Random search es ciego — cada evaluación es independiente de las anteriores.
¿Y si usamos los resultados previos para decidir dónde probar después?

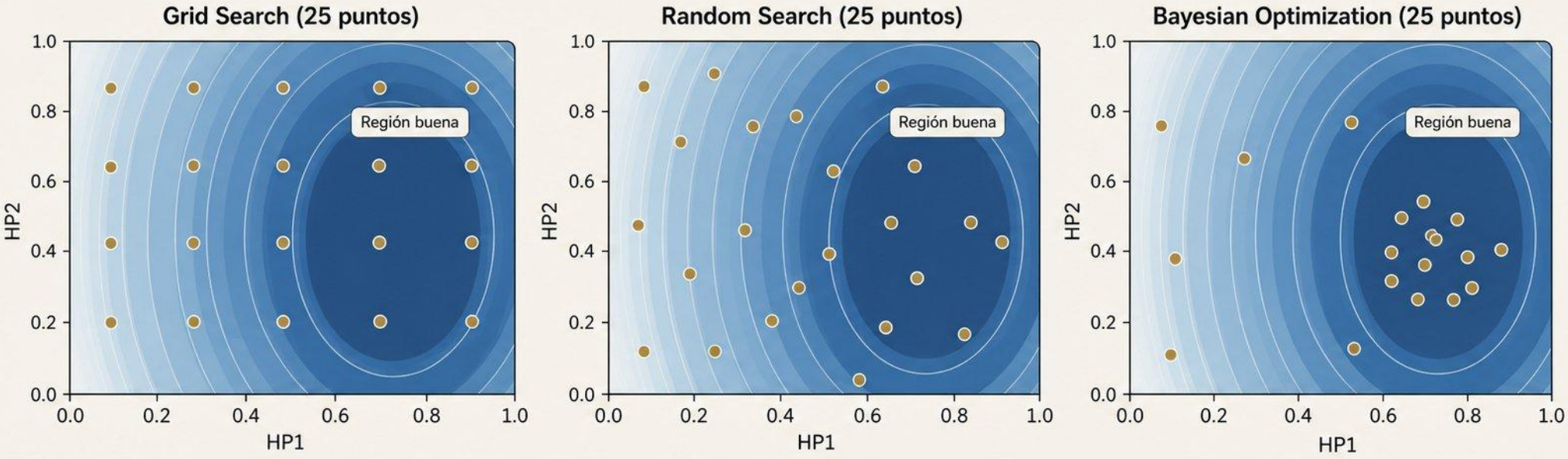
1. Después de cada evaluación, tenemos un punto (hiperparámetros, resultado).
2. Ajustamos un modelo subrogado sobre esos puntos — típicamente un *Gaussian Process* o un Tree-structured Parzen Estimator (TPE, lo que usa Optuna).
3. El modelo subrogado predice, para cada combinación de hiperparámetros, qué tan bueno se espera que sea el resultado y qué tan incierto es ese estimado.
4. Elegimos el siguiente punto a evaluar usando una *acquisition function* que balancea explotación (probar cerca de buenos resultados conocidos) y *exploración* (probar regiones inciertas).

Optuna es la librería estándar en Python. API limpia, paralelizable, integra bien con sklearn, XGBoost, LightGBM, PyTorch. Default razonable: TPE sampler + median pruner.
~50–100 trials suelen ser suficientes para tabular.

```python
import optuna

def objective(trial):
    params = {
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'learning_rate': trial.suggest_float('lr', 0.01, 0.3, log=True),
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
    }
    model = XGBClassifier(**params)
    score = cross_val_score(model, X, y, cv=5).mean()
    return score

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

UNIVERSIDAD Panamericana

Comparación visual: cómo exploran los tres métodos



Con el mismo número de evaluaciones (25 trials), Bayesian optimization concentra el presupuesto en regiones prometedoras. Para evaluaciones costosas, esto puede ser una diferencia de horas a minutos.

¿Cuándo usar cuál?

- **Pocos hiperparámetros (≤2) + evaluaciones baratas:** grid o random
- **Muchos hiperparámetros + evaluaciones caras:** Bayesian (Optuna)
- **Default razonable casi siempre:** Optuna con 50–100 trials

UNIVERSIDAD Panamericana

Errores comunes en hyperparameter tuning

1

Tunear sobre el test set.

Si tu loop de tuning usa el test set como objetivo, el test set deja de ser válido como estimador del error de generalización. Siempre tunear sobre validación, evaluar al final sobre test, una sola vez.

2

No fijar el random seed.

Si los modelos son entrenados con seeds distintos en distintas iteraciones del tuning, la varianza entre evaluaciones puede ser mayor que el efecto real de los hiperparámetros. Resultado: tuning 'encuentra' óptimos espurios.

3

Ignorar la varianza del estimador.

Si la diferencia entre dos configuraciones es de 0.001 AUC pero la desviación estándar entre folds es de 0.01, no hay diferencia significativa. Reportar siempre media ± std de los folds, no sólo la media.

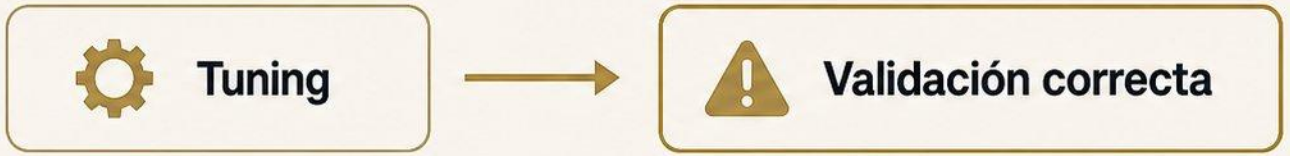
Buen tuning ≠ encontrar el AUC más alto.
Buen tuning = encontrar una configuración robusta que funcione bien en validación, sea reproducible, y generalice a producción.

UNIVERSIDAD Panamericana

Hasta aquí asumimos algo crïítico...

Toda esta discusión de tuning asume que K-fold CV nos da una estimación confiable del error de generalización.

Para datos tabulares iid, eso es razonable. Para datos financieros — series de tiempo, observaciones dependientes, regímenes que cambian — K-fold estándar miente.



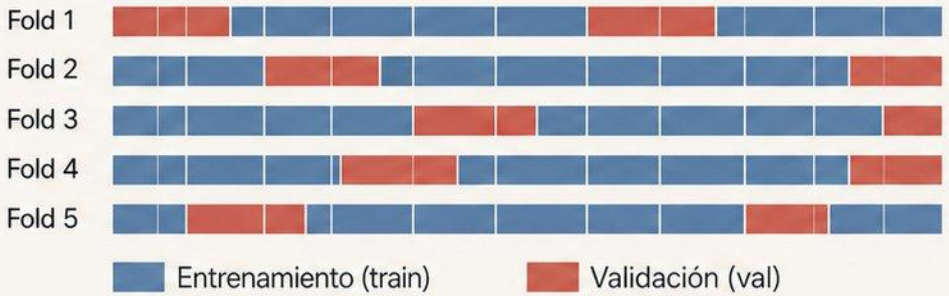
> **Esta es la sección más importante de la clase para su trabajo profesional. Presten atención.**

UNIVERSIDAD Panamericana

Por qué K-fold estándar falla en datos financieros

K-fold estándar asume que las observaciones son intercambiables — que el orden no importa.
Para datos iid, esto es cierto. Para series de tiempo, es falso.

K-fold estándar (mal)



Lo que pasa (fuga de información)

El modelo entrena con datos de 2023 (futuro)

Y luego se le pide predecir un punto en 2020 (pasado)

El modelo "ve" el futuro durante el entrenamiento y lo usa para predecir el pasado.

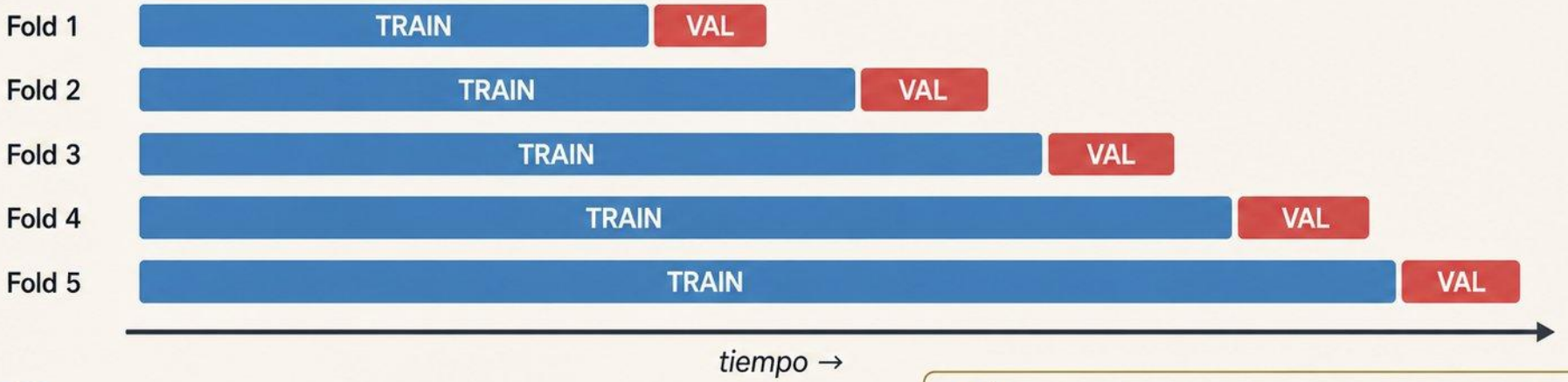
El modelo entrenó con datos posteriores al punto que está prediciendo. En producción, eso es imposible.
La métrica de validación está infladísima — y no te das cuenta hasta que el modelo entra en producción y empieza a fallar.

Caso real (pero anónimo): un modelo de scoring de crédito con K-fold AUC = 0.92 colapsó a AUC = 0.71 en producción. La causa: el dataset tenía features que cambiaban con el ciclo macroeconómico, y K-fold permitió al modelo 'ver' el futuro durante entrenamiento.

UNIVERSIDAD Panamericana

Walk-forward validation: respetar el flujo del tiempo

La idea correcta: en cualquier fold, los datos de validación deben estar **estrictamente después** que los datos de entrenamiento.



Expanding window (lo que muestra la figura): El training set crece con cada fold. Útil cuando más datos siempre ayudan.

Sliding window: El training set se mueve, manteniendo tamaño fijo. Útil cuando los regímenes cambian — datos viejos pueden ser activamente perjudiciales.

- **Para scoring de crédito** con datos de los últimos 5 años: expanding window.
- **Para trading con regímenes que cambian rápido:** sliding window.
- **Para microestructura de mercado intraday:** sliding window con períodos cortos.

En sklearn: `TimeSeriesSplit`. **En código propio:** ordenar por fecha y construir folds manualmente.

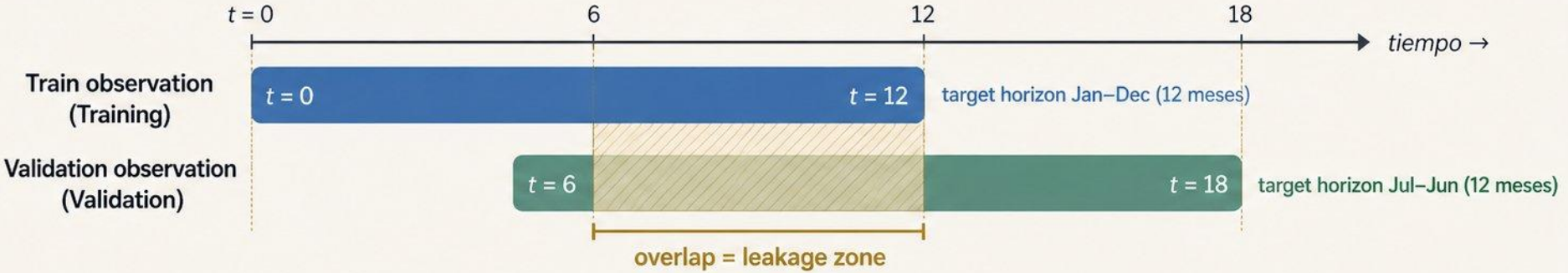
UNIVERSIDAD Panamericana

Clase 3: Refinamiento: purged K-fold con embargo

Walk-forward estándar resuelve el problema básico, pero hay sutilezas.

¿Y si tu target tiene horizonte temporal? Por ejemplo: 'default en los próximos 12 meses' o 'retorno acumulado a 30 días'.

Si la fila *i* del training set tiene fecha t_i, pero su target observa los próximos 12 meses, y la fila *j* del validation set está en $t_j = t_i + 6$ meses, entonces los targets se sobreponen. Eso es *leakage* incluso con walk-forward.



1. **Purging:** Eliminar del training set las observaciones cuyo target horizon se sobreponga con el validation set.
2. **Embargo:** Añadir un buffer de tiempo (típicamente 1–5% de los datos) entre el final del train y el inicio del validation, para evitar que features de corto plazo correlacionados sigan impactando.

Purging elimina información del pasado que "mira" hacia el futuro y contamina el validation.

Embargo bloquea efectos de dependencia de corto plazo que no se capturan solo con purging.

Marcos López de Prado, Advances in Financial Machine Learning (2018). Capítulo 7. *Lectura muy recomendada para todos los que trabajen con datos financieros.*

Universidad Panamericana

Caso especial: cuando las observaciones tienen grupos

Otro problema común que también rompe K-fold estándar: cuando tienes múltiples observaciones por entidad.
Ejemplos:

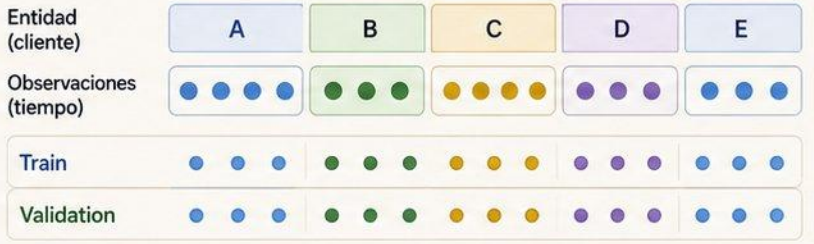
- Múltiples transacciones por cliente
- Múltiples préstamos por empresa
- Múltiples interacciones por usuario

La fuga de información:
Si las transacciones del cliente A están repartidas entre train y validation, el modelo "aprende" características específicas de A y las explota en validation. AUC se ve excelente, pero el modelo no generaliza a clientes nuevos.

La solución:
Group K-fold: asegurar que todas las observaciones de la misma entidad caigan en el mismo fold (todas en train o todas en validation, nunca repartidas).

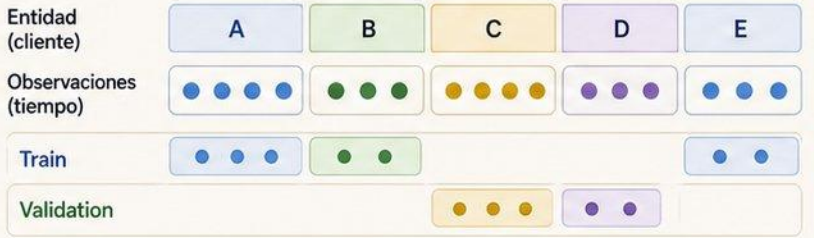
```
from sklearn.model_selection import GroupKFold

gkf = GroupKFold(n_splits=5)
for train_idx, val_idx in gkf.split(X, y, groups=cliente_id):
    ...
```

K-fold estándar (incorrecto: fuga de información)



La entidad A aparece en train y en validation → fuga (leakage).

Group K-fold (correcto: sin fuga)



Cada entidad completa cae en un solo fold (train o validation).

Si tus datos tienen estructura temporal Y agrupamiento (caso típico en finanzas), necesitas **group time-series CV** → sklearn no lo trae nativo, hay que implementarlo. Vale la pena.

UNIVERSIDAD Panamericana

Clase 3: Pipeline completo: tuning con validación correcta

Juntando todo: cómo se ve un pipeline serio para datos financieros.

```
# 1. Split temporal: últimos 6 meses son TEST (intocables hasta el final)
X_train, X_test, y_train, y_test = temporal_split(X, y, test_months=6)

# 2. Definir CV temporal con purging y embargo
cv = PurgedKFoldWithEmbargo(n_splits=5, embargo_pct=0.01,
                            target_horizon_days=30)

# 3. Optuna tunea usando este CV (no K-fold estándar)
def objective(trial):
    params = {...}
    model = XGBClassifier(**params)
    scores = cross_val_score(model, X_train, y_train, cv=cv)
    return scores.mean()

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)

# 4. Re-entrenar con los mejores hiperparámetros sobre TODO el train
best_model = XGBClassifier(**study.best_params).fit(X_train, y_train)

# 5. UNA sola evaluación sobre el test set
test_score = roc_auc_score(y_test, best_model.predict_proba(X_test)[:,1])
```

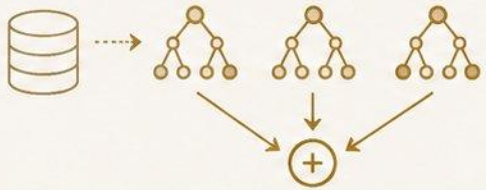
El test set se toca una sola vez, al final. Si después de ver el test score regresas a tunear más, contaminaste el test. La disciplina con el test set es lo que separa modelos que funcionan en producción de los que no.

UNIVERSIDAD Panamericana

Clase 3: Cierre: lo que cubre el bloque de ensembles (Clases 1–3)

1

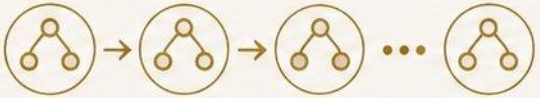
Clase 1: Bagging y RF



- Reducir varianza promediando modelos paralelos
- Bootstrap, OOB error, feature subsampling
- Tolerante a hiperparámetros

2

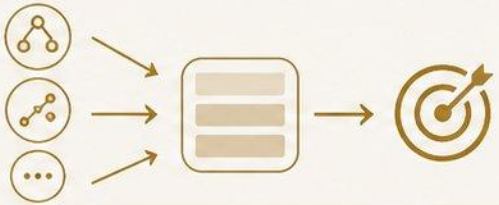
Clase 2: Boosting y XGBoost



- Reducir sesgo con modelos secuenciales
- Gradient descent en espacio de funciones
- Requiere tuning serio

3

Clase 3: Stacking, Tuning, CV



- Combinar modelos heterogéneos correctamente
- Tunear con Optuna, no con grid search
- Validar respetando estructura temporal

Cierran este bloque con un **toolkit completo**: pueden construir, combinar, tunear y validar ensembles para datos tabulares — incluyendo datos financieros con estructura temporal. La próxima clase pivoteamos a **métodos no supervisados** (PCA, t-SNE, UMAP) para visualización y reducción de dimensión.

UNIVERSIDAD Panamericana

Clase 3: Ahora al Colab

Notebook: 03_stacking_cv.ipynb

Qué vamos a hacer

- Implementar la versión incorrecta de stacking primero, ver el AUC inflado
- Implementar la versión correcta con OOF predictions, ver la diferencia
- Construir un stacked model con RF (Clase 1) + XGBoost (Clase 2) + LogReg meta-learner
- Sobre el dataset de crédito con índice temporal: comparar K-fold estándar vs walk-forward, ver cómo el AUC reportado cambia

Al final de la sesión: una celda donde reportan (1) el AUC del stacked model con OOF correcto, (2) el AUC del mismo stack con K-fold estándar mal usado, y (3) el AUC con walk-forward.
La diferencia entre los tres es el aprendizaje de la clase.

El dataset de crédito ahora viene con una columna fecha. Las funciones helper de Clases 1 y 2 ya están importadas.