1. Resumen ejecutivo (≤10 líneas): cuándo usarlo y qué promete.

CatBoost (Categorical Boosting) es un algoritmo de aprendizaje automático basado en *Gradient Boosting* sobre árboles de decisión. Es especialmente potente y recomendado cuando se trabaja con **datos tabulares heterogéneos** que incluyen **variables categóricas**. Promete resultados de estado del arte (SOTA) sin necesidad de un preprocesamiento exhaustivo de características (como One-Hot Encoding manual), ofreciendo robustez contra el sobreajuste gracias a su novedosa técnica de "Ordered Boosting" y una inferencia extremadamente rápida debido a su estructura de árboles simétricos.

2. Formulación matemática

- **Función objetivo**: Minimiza una función de pérdida $L(y, F(x))$ diferenciable (ej. MSE para regresión, LogLoss para clasificación) mediante descenso de gradiente en el espacio de funciones.
- **Mecanismo de aditividad**: El modelo final es una suma de modelos base (árboles): $F_M(x) = \sum_{m=1}^{M} \eta \cdot h_m(x)$, donde η es la tasa de aprendizaje.
- **Innovaciones específicas de CatBoost**:
  - **Ordered Boosting**: Para combatir el *prediction shift* (sesgo al calcular gradientes con los mismos datos de entrenamiento), CatBoost utiliza un esquema de permutaciones. Calcula el residuo para el ejemplo i utilizando un modelo entrenado solo con los ejemplos que le preceden en una permutación aleatoria virtual.
  - **Árboles Simétricos (Oblivious Trees)**: A diferencia de XGBoost/LGBM que crecen asimétricamente, CatBoost usa árboles donde la misma condición de división se aplica a todo el nivel. Esto actúa como regularización y permite una evaluación muy rápida.
  - **Manejo de Categóricas (Ordered Target Statistics)**: Transforma categorías a números usando estadísticas del target derivadas de permutaciones para evitar *data leakage*:

$$\hat{x}_i = \frac{\sum_{j \in S_i} \mathbb{I}_{\{x_j = x_i\}} \cdot y_j + a \cdot P}{\sum_{j \in S_i} \mathbb{I}_{\{x_j = x_i\}} + a}$$

    Donde S_i son ejemplos anteriores en la permutación, P es

el prior (promedio global) y a es un peso del prior.

3. Algoritmo (alto nivel)

1. **Inicialización**: Se comienza con un modelo constante (ej. media del target).
2. **Generación de Permutaciones**: Se generan múltiples permutaciones aleatorias de los datos de entrenamiento para el cálculo robusto de estadísticas y gradientes.
3. **Iteración ($t = 1$ a M)**:
   - **Cálculo de Gradientes/Residuos**: Se calculan los gradientes negativos de la función de pérdida usando el esquema *Ordered Boosting* (modelos parciales).
   - **Construcción del Árbol**: Se construye un árbol simétrico (oblivious tree).
     - Se evalúan divisiones para features numéricos y combinaciones de categóricas.
     - Se selecciona la división que maximiza la ganancia (ej. Cosine similarity con el gradiente).
   - **Cálculo de Hojas**: Se asignan valores a las hojas del árbol para minimizar la pérdida en los ejemplos que caen en ellas.
   - **Actualización**: Se añade el árbol al ensamble escalado por el *learning rate*: $F_t = F_{t-1} + \eta \cdot h_t$.

4. Hiperparámetros clave & efectos

- `iterations` / `n_estimators`: Número máximo de árboles. Determina la complejidad máxima.
- `learning_rate`: Paso de actualización. Valores bajos requieren más iteraciones pero generalizan mejor.
- `depth`: Profundidad del árbol simétrico. CatBoost suele funcionar bien con árboles poco profundos (4-10). Valores altos (>10) pueden ser muy lentos.
- `l2_leaf_reg`: Coeficiente de regularización L2 para los valores de las hojas. Ayuda a evitar overfitting.
- `rsm` **(Random Subspace Method)** / `colsample_bylevel`: Porcentaje de features consideradas en cada split.
- `subsample`: Tasa de muestreo de filas (solo si se usa `bootstrap_type` como Bernoulli o Poisson).
- `one_hot_max_size`: Umbral de cardinalidad bajo el cual las categóricas se codifican como One-Hot en lugar de Target Encoding.
- `border_count`: Número de divisiones para discretizar features numéricos (afecta velocidad y precisión).

5. Ventajas y limitaciones

- **Ventajas**:
  - **Manejo superior de categóricas**: No requiere codificación externa, usa toda la información de las categorías.
  - **Robustez**: Menos propenso a overfitting en datasets pequeños/medianos gracias a *Ordered Boosting*.
  - **Velocidad de inferencia**: Muy rápida debido a la estructura de árboles simétricos.
  - **Facilidad de uso**: Los hiperparámetros por defecto suelen dar resultados muy competitivos.
- **Limitaciones**:
  - **Velocidad de entrenamiento**: Puede ser más lento que LightGBM en CPU, aunque muy rápido en GPU.
  - **Opacidad**: Como todo boosting, es un modelo de "caja negra" (aunque interpretable con SHAP).
  - **Memoria**: Puede consumir más memoria durante el entrenamiento debido al manejo de permutaciones.

6. Buenas prácticas

- **No hacer One-Hot Encoding manual**: Pasar las columnas categóricas directamente al parámetro `cat_features`.
- **Usar GPU**: Si el dataset es grande, configurar `task_type="GPU"` acelera drásticamente el entrenamiento.
- **Early Stopping**: Usar un set de validación y `early_stopping_rounds` para detener el entrenamiento cuando la métrica deje de mejorar.
- **Manejo de Overfitting**: Si hay overfitting, aumentar `l2_leaf_reg`, reducir `depth`, o aumentar `rsm`.
- **Pools**: Usar la clase `catboost.Pool` para encapsular datos y features categóricas eficientemente.

7. Casos de uso y pitfalls frecuentes

- **Casos de uso**: Detección de fraude, riesgo de crédito, predicción de clics (CTR), recomendaciones, precios de seguros (cualquier problema tabular con variables cualitativas).
- **Pitfalls (Errores comunes)**:
  - Codificar categóricas manualmente antes de pasarlas a CatBoost (se pierde la ventaja del Target Encoding interno).
  - No especificar `cat_features` explícitamente.
  - Usar `Ordered` boosting (por defecto en datasets pequeños) en datasets masivos donde `Plain` podría ser mucho más

rápido y suficiente.
- Data leakage al validar series temporales (usar validación temporal estricta).

8. Checklist de tuning: orden recomendada

1. **Configuración inicial**: Usar defaults, fijar `cat_features` y `task_type="GPU"` si es posible.
2. **Número de árboles y Learning Rate**: Fijar un `learning_rate` (ej. 0.05 o 0.1) y encontrar el `iterations` óptimo con *early stopping*.
3. **Profundidad (`depth`)**: Probar valores como [4, 6, 8, 10]. Es el parámetro más sensible al rendimiento/tiempo.
4. **Regularización (`l2_leaf_reg`)**: Probar valores [1, 3, 5, 10] para controlar la varianza.
5. **Estocasticidad (`rsm`, `subsample`)**: Ajustar para añadir aleatoriedad y robustez.
6. **Refinamiento final**: Reducir el `learning_rate` y aumentar `iterations` proporcionalmente para el modelo final de producción.