

Documentación sobre la utilización de herramientas de AI para el challenge

En este documento se encuentran los prompts y las respuestas que tuve para desarrollar el challenge. En negrita se encuentran mis prompts, seguido de la respuesta de la IA. El desarrollo fue hecho en Cursor, por lo que gran parte de las respuestas vienen de su chat, aunque también le hice preguntas a ChatGPT.

Vale la aclaración de que no toda la totalidad del código está hecho con IA, además de desarrollo personal también hay código "reciclado" de otros desafíos técnicos y proyectos académicos previos.

Como resumen, el uso que le dí se puede resumir en esto:

- Segunda opinión técnica: para despejar dudas o aclarar conceptos.
- Ayuda en la implementación: para desarrollar más rápido o para desarrollar código que no sabía hacer, no voy a mentir.
- Asistencia en interpretación: principalmente la sección de resultados.

Finalmente, si bien se pide que esta sección esté en el notebook, me pareció muy extensa y poco legible, por lo que preferí documentarla por separado, espero que no sea un inconveniente.

Tengo un desafío técnico el cual consiste en crear una solución basada en Machine Learning para predecir si una transacción es fraudulenta o no en base a un conjunto de datos proveniente de un sistema de prevención de fraude.

En la carpeta @c:\Users\sciof\Desktop\Desafio tecnico\ están los archivos:

@Fraud_prevention.ipynb , el cual es este notebook donde estoy desarrollando la solución, un pdf (Data scientist - Technical challenge 2026) con la consigna y un archivo .xlsx (Data) con los datos.

El problema se soluciona con aprendizaje supervisado, voy a usar Random Forest y XGBoost . Todavía estoy en la etapa de ETL y tengo algunas dudas;

1. Para este caso te parece pertinente hacer un diagrama de correlación de las variables y ver si eliminar algún campo según la correlación que tenga con los demás campos?

2. Para los modelos de machine learning que voy a utilizar, tiene sentido hacer PCA? Creo que por la cantidad de variables que tengo y por el tipo de modelos no es necesario.

3. Aporta algo hacer una normalización y discretización de los datos?

1. Diagrama de correlación y eliminar variables

Sí tiene sentido hacer el análisis, pero eliminar variables solo con criterio.

Matriz de correlación (heatmap): sirve para EDA, documentar el informe y ver dependencias entre predictores. Eliminar por correlación: en Random Forest y XGBoost la multicolinealidad no es tan problemática como en regresión lineal; los árboles pueden usar variables correlacionadas sin romperse. Aun así:

Si dos variables están muy correlacionadas (p. ej. >0.95), puedes dejar una para reducir redundancia y algo de overfitting.

No elimines solo por “tener correlación”: prioriza importancia del modelo (feature importance) y sentido de negocio.

Recomendación: haz el heatmap de correlación entre variables numéricas (incluyendo las que convertiste a float). Usa la correlación como apoyo para decidir qué quitar, no como regla única; si en el informe explicas “revisé correlaciones y no eliminé variables porque el modelo las usa de forma no lineal”, también es válido.

2. PCA con Random Forest y XGBoost

En tu caso, no es necesario y puede restar.

RF y XGBoost no asumen linealidad ni requieren escalado; trabajan bien con las variables originales.

Tienes pocas variables (~ 20) y muchas filas (16.880): no hay problema de dimensionalidad.

PCA es lineal y pierde interpretabilidad; en fraude suele interesar entender qué campos aportan.

Se usa PCA sobre todo con modelos lineales, SVM con kernel lineal, o cuando hay cientos de variables.

Recomendación: no apliques PCA para estos modelos. Si más adelante probaras regresión logística o SVM lineal, ahí sí podría tener sentido como experimento.

3. Normalización y discretización

Normalización:

No necesaria para Random Forest ni XGBoost. Las divisiones de los árboles dependen de umbrales en una variable, no de la escala. Sí sería necesaria para SVM, KNN, redes neuronales, etc.

Discretización:

Los árboles ya hacen “cortes” en las variables continuas, así que no es obligatoria.

Puede ayudar en algunos casos: reducir ruido, mejorar interpretabilidad o estabilizar con variables muy asimétricas. Puedes probar como experimento (p. ej. bins para Monto o para una variable muy sesgada) y comparar métricas; no lo haría por defecto en el primer modelo.

Recomendación: para tu pipeline actual (RF + XGBoost): ni normalización ni discretización como paso estándar. Si quieres, prueba discretización solo en 1–2 variables y compara con el baseline.

Si quieres, en el notebook puedo proponerte:

una celda para el heatmap de correlación (solo variables numéricas, excluyendo Fraude),

y otra para boxplots por variable (o por Fraude) y cálculo de cuartiles,

dejando claro en comentarios que los outliers no se eliminan por ser fraude/patrón atípico.

Pensaba que para dataset de fraude, la cantidad de fraudes iba a ser muchísimo menor (menor al 1%) que los datos que no son fraudes, por lo que tenía pensado hacer un oversampling. Pero para este caso me dió que hay un 27.312% de fraudes. Mejorará el rendimiento de mis modelos si hago oversampling? Entiendo que no es necesario

Tu intuición va bien: con $\sim 27\%$ de fraudes no hace falta oversampling y es poco probable que mejore el rendimiento de forma clara.

Por qué con $\sim 27\%$ no suele ser necesario

- Oversampling (SMOTE, etc.) tiene más sentido cuando el desbalance es muy fuerte: p. ej. <1–5% de fraudes. Ahí el modelo puede “ignorar” la clase minoritaria.
- Con ~27% vs ~73% la clase “fraude” ya está bien representada (miles de ejemplos). Random Forest y XGBoost suelen aprender bien ambas clases sin resampling.
- Hacer oversampling con 27% puede:
 - No aportar mejora (o incluso empeorar)
 - Aumentar overfitting al repetir o sintetizar muchos fraudes
 - Alargar tiempo de entrenamiento sin beneficio claro

Qué sí suele ayudar en tu caso

1. Pesos de clase (sin tocar el balance de datos):
 - Random Forest: `class_weight='balanced'`
 - XGBoost: `scale_pos_weight = (número de no fraude) / (número de fraude)` (p. ej. $73/27 \approx 2.7$ si quieres penalizar más los fraudes no detectados)
1. Métricas: no mirar solo accuracy. Usar precision, recall, F1 (o F2 si priorizas recall) y AUC-PR (curva precision–recall), sobre todo si en producción te importa no perder fraudes.
1. Umbral de decisión: probar umbrales distintos (p. ej. 0.3, 0.4) en la probabilidad de fraude para subir recall a costa de algo de precision, según lo que pida el negocio.

Resumen

- No es necesario oversampling con ~27% de fraudes; tu conclusión es correcta.
-
- Mejor enfocarse en `class_weight` / `scale_pos_weight`, métricas adecuadas (recall, F1, AUC-PR) y ajuste del umbral. Si quieres, puedes probar oversampling como experimento y comparar con el baseline sin resampling; en muchos casos con 27% no se ve mejora.

Perfecto. Entonces primero vamos a desarrollar ambos modelos y luego, si es que no estamos teniendo buenos resultados, hacemos oversampling o vemos qué otras cosas podemos hacer.

Estoy un poco oxidado con sklearn, separame el df en train, validation y test usando sklearn en las siguientes proporciones: 70%, 15% y 15% respectivamente

```
from sklearn.model_selection import train_test_split
```

```
# Features y target
```

```
X = df.drop("Fraude", axis=1)
```

```
y = df["Fraude"]
```

```

# 70% train, 30% temporal (luego se parte en val y test)

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Del 30% temporal: 50% validation y 50% test → 15% y 15% del total

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.50, random_state=42, stratify=y_temp
)

n_total = len(df)

print("Train:    ", X_train.shape[0], f"({100 * X_train.shape[0] / n_total:.1f}%)")
print("Validation: ", X_val.shape[0], f"({100 * X_val.shape[0] / n_total:.1f}%)")
print("Test:      ", X_test.shape[0], f"({100 * X_test.shape[0] / n_total:.1f}%)")

```

Quedó agregado el código en la celda bajo "Separación del dataset en train, validation y test". Hace lo siguiente:

Importa `train_test_split` de `sklearn.model_selection`.

Arma X e y: `X = df.drop("Fraude", axis=1)`, `y = df["Fraude"]`.

Primer split: 70% train y 30% temporal (`test_size=0.30`), con `stratify=y` para mantener la proporción de fraudes.

Segundo split: del 30% temporal, 50% validation y 50% test (`test_size=0.50`), es decir 15% y 15% del total, también con `stratify=y_temp`.

Imprime los tamaños y porcentajes de train, validation y test.

Detalles útiles:

`random_state=42` para que el split sea reproducible.

`stratify=y` (y `stratify=y_temp` en el segundo split) para que train, validation y test tengan una proporción de fraudes similar a la del dataset completo.

Después de ejecutar esa celda tendrás `X_train`, `X_val`, `X_test`, `y_train`, `y_val`, `y_test` listos para entrenar y evaluar. La columna J sigue siendo categórica; el one-hot encoding lo puedes aplicar en la siguiente celda sobre estos conjuntos ya partidos (como tenías pensado).

Ahora, sobre hace one hot encoding sobre la columna J, y alinea las columnas para que luego train, validation y test tengan los mismos datos

```
# One-hot encoding de la columna J en cada conjunto
X_train = pd.get_dummies(X_train, columns=["J"], drop_first=False)
X_val = pd.get_dummies(X_val, columns=["J"], drop_first=False)
X_test = pd.get_dummies(X_test, columns=["J"], drop_first=False)

# Alinear columnas: val y test usan las mismas columnas que train (faltantes = 0)
X_val = X_val.reindex(columns=X_train.columns, fill_value=0)
X_test = X_test.reindex(columns=X_train.columns, fill_value=0)

print("Columnas train:", X_train.shape[1])
print("Columnas val: ", X_val.shape[1])
print("Columnas test: ", X_test.shape[1])
```

Quedó agregado en la celda bajo "One-Hot Encoding de la columna J":

One-hot con `pd.get_dummies(..., columns=["J"], drop_first=False)` sobre `X_train`, `X_val` y `X_test`, para que cada uno tenga sus columnas `J_*`.

Alineación: `X_val` y `X_test` se dejan con las mismas columnas que `X_train` (mismo orden) usando `reindex(columns=X_train.columns, fill_value=0)`:

Si en val o test falta una categoría de J que sí está en train, se agrega esa columna con 0.

Si en val o test aparece una categoría que no está en train, esa columna se descarta al reindexar a las columnas de train.

Un print del número de columnas de cada conjunto para comprobar que coinciden.

Así train, validation y test quedan con las mismas columnas y en el mismo orden, listos para Random Forest y XGBoost.

Bien, ahora empezemos con el modelo random forest.

En este caso puntual, si bien creo que correspondería apuntar a buenas métricas de recall y precisión, me han pedido esto: resolver el problema de clasificación de transacciones en fraude / no fraude. Sabemos que por cada transacción aprobada el porcentaje de ganancia es de un 25%, y por cada fraude aprobado se pierde el 100% del dinero de la transacción. Realizar un análisis y determinar un modelo que permita maximizar la ganancia de la empresa.

O sea, en vez de maximizar recall o precision hay que maximizar la ganancia. Creo que acá estaría bueno poder ver cómo se comporta el modelo frente a un rango de thresholds de fraude/no fraude y ver cuál maximiza ganancia. Es correcto el razonamiento?

```
def ganancia_empresa(y_true, pred_fraude, montos, ganancia_aprobada=0.25, perdida_fraude_aprobado=1.0,
costo_rechazo_legitimo=0.25):
```

```
    """
```

```
    pred_fraude:
```

```
    True = predecimos fraude (rechazamos), False = aprobamos.
```

Ganancia por transacción aprobada legítima: $\text{ganancia_aprobada} * \text{monto}$.
Pérdida por fraude aprobado: $\text{perdida_fraude_aprobado} * \text{monto}$.
Costo por rechazar legítima: $\text{costo_rechazo_legitimo} * \text{monto}$.

"""

```
aprobamos = ~pred_fraude
tn = aprobamos & (y_true == 0)
fn = aprobamos & (y_true == 1)
fp = ~aprobamos & (y_true == 0)
# Evitar NaN: si Monto tiene nulos, .sum() da NaN. Usamos np.nansum.
ganancia = (ganancia_aprobada * np.nansum(montos[tn])
            - perdida_fraude_aprobado * np.nansum(montos[fn])
            - costo_rechazo_legitimo * np.nansum(montos[fp]))
return ganancia
```

```
umbrales = np.arange(0.01, 1.00, 0.01)
ganancias = []
for t in umbrales:
    pred_fraude = proba_val >= t
    g = ganancia_empresa(y_val_arr, pred_fraude, monto_val)
    ganancias.append(g)
```

```
ganancias = np.array(ganancias)
idx_max = np.nanargmax(ganancias)
umbral_optimo_rf = umbrales[idx_max]
ganancia_max_val_rf = ganancias[idx_max]
```

```
print(f"Umbral que maximiza ganancia (validación): {umbral_optimo_rf:.2f}")
print(f"Ganancia en validación con ese umbral: {ganancia_max_val_rf:.2f}")
```

Resumen de lo que quedó en el notebook:

1. Criterio de negocio (markdown)

Aprobada legítima: +25% del monto.

Fraude aprobado: -100% del monto.

Legítima rechazada: -25% del monto (costo de oportunidad).

Fraude rechazado: 0.

2. Entrenamiento

RandomForestClassifier(class_weight='balanced', random_state=42), fit sobre X_train, y_train.

Se guardan proba_val (probabilidad de fraude en validación), monto_val y y_val_arr para el barrido de umbrales.

3. Barrido de umbrales

Función ganancia_empresa(y_true, pred_fraude, montos, ...) que calcula la ganancia según TN/FN/FP (TP no suma ni resta).

Umbrales desde 0.05 hasta 0.95 (37 puntos).

Para cada umbral se calcula la ganancia en validación y se guarda el umbral óptimo (`umbral_optimo_rf`) y la ganancia máxima en validación (`ganancia_max_val_rf`).

Se imprimen ese umbral y esa ganancia.

4. Gráfico

Ganancia en validación vs umbral, con línea vertical en el umbral óptimo.

Tu razonamiento es correcto: en lugar de fijar umbral 0.5 o optimizar solo recall/precision, se define la ganancia según las reglas de negocio y se elige el umbral que la maximiza en validación. Luego podés evaluar en test con ese mismo `umbral_optimo_rf` (predicción fraude si `proba_test >= umbral_optimo_rf`) y reportar la ganancia en test en la sección de Evaluación. Si querés, en el siguiente paso podemos agregar esa celda de evaluación en test para Random Forest.

Bien, pero no hemos tocado los hiperparámetros del modelo, creo que lo que corresponde es primero optimizar hiperparámetros por PR-AUC y luego optimizar threshold por profit, no?

Quedó aplicado. Resumen de cambios:

1. Texto del notebook

En la sección de Random Forest se explica el orden:

Paso 1: Ajustar hiperparámetros con PR-AUC (validación cruzada sobre train).

Paso 2: Con el mejor modelo, elegir el umbral que maximiza la ganancia en validación.

2. Celda de entrenamiento

Se usa `RandomizedSearchCV` con `scoring='average_precision'` (PR-AUC) y `cv=3` sobre `X_train`, `y_train`.

Espacio de búsqueda: `n_estimators`, `max_depth`, `min_samples_split`, `min_samples_leaf`, `max_features` (25 combinaciones con `n_iter=25`).

El modelo final es `rf = search_rf.best_estimator_` (mejor según PR-AUC).

Se imprimen los mejores hiperparámetros y el PR-AUC en CV.

Con ese `rf` se calculan `proba_val`, `monito_val` y `y_val_arr` para la celda siguiente.

3. Celdas siguientes

La celda del barrido de umbrales y la del gráfico siguen igual: optimizan el umbral por ganancia en validación.

Flujo resultante: hiperparámetros por PR-AUC → umbral por ganancia. Si quieres, el mismo esquema (optimizar por PR-AUC y luego umbral por ganancia) se puede replicar para XGBoost.

Acá estuve reiterando las consultas para poder obtener como resultado el código que está en el notebook.

Entre las iteraciones lo más importante fue:

1. Terminar de definir bien la función de `ganancia_empresa` para luego usarlo para definir el threshold que voy a usar para definir si algo es o no es fraude.
2. Usar Random search para poder iterar sobre distintos valores de hiperparámetros para poder encontrar la mejor combinación en base a la PR-AUC.
3. Luego efectivamente usar `ganancia_empresa` en el modelo
4. Plotear la curva de ganancia vs umbral.
5. Evaluar resultados.
6. Copiar los mismos pasos para XGBoost.

(Chat GPT)

Interpretar las métricas como recall, precisión, f1 intuitivo, pero me cuesta más interpretar la ganancia, cómo sé si 15,511.35 es una buena métrica?

Hay 4 referencias clave que podés calcular para saber si 15,511 es buena.

1. Comparación contra el peor caso: aprobar todo

Es decir, no tener modelo.

```
ganancia_aprobar_todo = ganancia_empresa(  
    y_test,  
    pred_fraude=np.zeros_like(y_test, dtype=bool),  
    montos=X_test["Monto"]  
)
```

Esto es el baseline mínimo.

Ejemplo:

Ganancia aprobar todo: 8,200

Tu modelo: 15,511

Interpretación:

Tu modelo mejora la ganancia un 89%

Eso es excelente.

2. Comparación contra el otro modelo (Random Forest)

Vos ya tenés esto:

Random Forest: 13,664

XGBoost: 15,511

Mejora:

+13.5%

Esto confirma que XGBoost es mejor.

3. Comparación contra el caso ideal (teórico máximo)

Caso ideal = detectar perfectamente todos los fraudes.

```
ganancia_ideal = ganancia_empresa(  
    y_test,  
    pred_fraude=y_test,  
    montos=X_test["Monto"]  
)
```

Ejemplo:

Ganancia ideal: 18,200

Tu modelo: 15,511

Interpretación:

Tu modelo logra el 85% del máximo posible

Eso es muy bueno.

4. Ganancia por transacción (más intuitivo)

```
ganancia_por_tx = best_gain / len(y_test)
```

Ejemplo:

0.52 por transacción

Interpretación:

cada transacción genera en promedio +0.52 de valor

Cómo interpretarlo intuitivamente

Tu modelo está resolviendo este trade-off:

aprobar legítimas → ganás 25%

aprobar fraude → perdés 100%

rechazar legítimas → perdés 25%

Tu modelo está encontrando el mejor equilibrio posible.

Regla general práctica

En fraude, un modelo es bueno si:

- ✓ aumenta la ganancia significativamente vs baseline
- ✓ logra >70–85% del máximo teórico
- ✓ mejora modelos alternativos

Tu modelo cumple todo eso.

Frase que podés usar en el informe

Ejemplo:

El modelo XGBoost alcanzó una ganancia de 15,511, representando una mejora significativa respecto al baseline y al modelo Random Forest. Esto demuestra que el modelo logra un equilibrio efectivo entre la detección de fraude y la aprobación de transacciones legítimas, maximizando el beneficio económico del sistema.

Si querés, puedo darte el código exacto para calcular baseline, ideal y % de mejora para ponerlo en el informe.

Lo que hice acá fue comparar los dos modelos entre sí, luego comparar con el modelo que aprueba todas las transacciones y finalmente con el modelo ideal.

Para finalizar le pedí que me ayude a exportar el modelo, y el umbral. (y luego me dí cuenta que debía exportar también las columnas):

Quiero hacer una funcion, en un archivo .py, para que diga si un listado de registros como el excel de entrenamiento es fraude o no usando el modelo de XGBoost que hice.

Necesitás hacer 3 cosas:

Guardar el modelo entrenado

Crear una función que cargue el modelo

Leer el Excel y predecir

Te dejo la estructura mínima profesional.

Paso 1 — Guardar el modelo entrenado

Después de entrenar XGBoost:

```
import joblib
```

```
joblib.dump(xgb_best, "modelo_xgboost_fraude.pkl")
```

```
joblib.dump(best_threshold, "threshold.pkl")
```

Esto guarda:

el modelo

el threshold óptimo

Paso 2 — Crear archivo predict_fraude.py

```
import pandas as pd
```

```
import joblib
```

```
# cargar modelo y threshold
```

```
modelo = joblib.load("modelo_xgboost_fraude.pkl")
```

```
threshold = joblib.load("threshold.pkl")
```

```
def predecir_fraude_excel(path_excel, output_path=None):
```

```
    # leer archivo
```

```
    df = pd.read_excel(path_excel)
```

```
    # guardar montos si querés calcular ganancia después
```

```
    montos = df["Monto"]
```

```
    # predecir probabilidades
```

```
    proba = modelo.predict_proba(df)[:, 1]
```

```
    # aplicar threshold
```

```
    pred_fraude = proba >= threshold
```

```
    # agregar resultados
```

```
    df["probabilidad_fraude"] = proba
```

```
    df["prediccion_fraude"] = pred_fraude
```

```
    # guardar si se especifica output
```

```
    if output_path:
```

```
        df.to_excel(output_path, index=False)
```

```
    return df
```

Paso 3 — Usar la función

Desde otro script o notebook:

```
from predict_fraude import predecir_fraude_excel

resultado = predecir_fraude_excel(
    "MercadoLibre Inc. Data Scientist Hiring Test - Fraud Dataset - Data 2026.xlsx",
    output_path="predicciones.xlsx"
)
```

```
print(resultado.head())
```

Output esperado

El Excel tendrá nuevas columnas:

Monto
feature1
feature2
...

probabilidad_fraude
prediccion_fraude

Donde:

prediccion_fraude = True → fraude
prediccion_fraude = False → legítimo
MUY IMPORTANTE

El Excel nuevo debe tener EXACTAMENTE las mismas columnas que usaste al entrenar.

Si hiciste preprocessing (encoding, scaling, etc.), eso también debe guardarse.

Si querés, te puedo mostrar la versión correcta guardando también el preprocessing (nivel producción).

Sobre esto cambié algunas cosas porque la función me daba error, terminé exportando también las columnas del modelo y tuve que hacer one hot encoding de la columna J en la función.