03.5_Modelo_2outputs

Objetivo

Entrenar y comparar modelos de clasificación binaria (`Logistic Regression`, `RandomForest` y `GradientBoosting`) a partir de las cuatro categorías de riesgo originales, remapeadas a dos clases (`bajo/medio` vs. `alto`). Se busca evaluar distintas estrategias de ajuste de umbral para priorizar la detección de alto riesgo o la protección de bajo riesgo, **sin aplicar SMOTE** y utilizando `class_weight='balanced'`.

Entradas (Inputs)

- `data/splits/experiments/X_train_17.parquet`
- `data/splits/experiments/X_val_17.parquet`
- `data/splits/experiments/X_test_17.parquet`
- `data/splits/final/y_train.parquet`
- `data/splits/final/y_val.parquet`
- `data/splits/final/y_test.parquet`

Salidas (Outputs)

- `data/splits/final/y_train_2_classes.parquet`
- `data/splits/final/y_val_2_classes.parquet`
- `data/splits/final/y_test_2_classes.parquet`

Resumen Ejecutivo

- Este notebook entrena y compara tres modelos de clasificación binaria (Regresión Logística, Random Forest y Gradient Boosting) sobre los datos remapeados a dos clases (bajo vs. alto riesgo), sin aplicar SMOTE y empleando `class_weight='balanced'`.
- Se cargan los splits entrenados con 17 variables sintéticas y se remapean las etiquetas originales a dos categorías (`y_*_2`).
- Se definen pipelines de preprocesamiento (`StandardScaler`) y clasificador para cada algoritmo.
- Se implementa la función `evaluate_binary_realistic` para calcular accuracy, F1 y AUC realista, mostrando además las matrices de confusión.
- Se ajustan dos umbrales específicos para Random Forest:
    1. **Detector de Alto Riesgo** (maximiza detección de clase alta)
    2. **Protector de Bajo Riesgo** (minimiza errores en clase baja)

- Regresión Logística y Gradient Boosting se evalúan con umbral 0.5; Random Forest con los umbrales optimizados en validación.
- La comparación en validación y test evidencia cómo cada modelo y estrategia de threshold impacta precision/recall de cada clase.

---

1. Montar Google Drive, importar librerías y cargar configuración

Esta celda monta Google Drive, importa y organiza las librerías necesarias (estándar, terceros y locales), añade la raíz del proyecto al `sys.path` y carga las rutas de configuración para los directorios de splits.

```
# MONTAR DRIVE, IMPORTAR LIBRERÍAS Y CARGAR CONFIGURACIÓN

# Standard library
from pathlib import Path
import sys

# Google Colab
from google.colab import drive

# Data processing
import numpy as np
import pandas as pd

# Scikit-learn
from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_s
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# 1. Montar Google Drive
drive.mount('/content/drive', force_remount=True)

# 2. Añadir la raíz del proyecto al path
ROOT_PATH_STR = '/content/drive/MyDrive/TFM-AntonioEsquinas'
if ROOT_PATH_STR not in sys.path:
    sys.path.append(ROOT_PATH_STR)

# 3. Importar las rutas necesarias desde el archivo de configuración
from config import FINAL_SPLITS_DIR, EXP_SPLITS_DIR

print("Drive montado, librerías importadas y configuración de rutas cargada.")
print(f"Directorio de splits experimentales: {EXP_SPLITS_DIR}")
print(f"Directorio de splits finales (para 'y' originales): {FINAL_SPLITS_DIR}")
```

```
Mounted at /content/drive
✅ Módulo de configuración cargado y estructura de carpetas asegurada.
✅ Drive montado, librerías importadas y configuración de rutas cargada.
▶️ Directorio de splits experimentales: /content/drive/MyDrive/Digitech/TFG/ML/Calcu
```

```
Directorio de splits finales (para 'y' originales): /content/drive/MyDrive/Digite
```

2. Cargar conjuntos de datos desde archivos Parquet

Esta celda carga los conjuntos de datos sintéticos de entrenamiento desde archivos Parquet ubicados en `EXP_SPLITS_DIR`, manejando excepciones si no se encuentran.

```
# CARGAR LOS CONJUNTOS DE DATOS

try:
    # Las 'X' con 17 features vienen del experimento anterior (03.4)
    X_train_sint = pd.read_parquet(EXP_SPLITS_DIR / 'X_train_17.parquet')
    X_val_sint   = pd.read_parquet(EXP_SPLITS_DIR / 'X_val_17.parquet')
    X_test_sint  = pd.read_parquet(EXP_SPLITS_DIR / 'X_test_17.parquet')

    # Las 'y' originales vienen del split final (03.1)
    y_train = pd.read_parquet(FINAL_SPLITS_DIR / 'y_train.parquet').squeeze()
    y_val   = pd.read_parquet(FINAL_SPLITS_DIR / 'y_val.parquet').squeeze()
    y_test  = pd.read_parquet(FINAL_SPLITS_DIR / 'y_test.parquet').squeeze()

    print("Datos .parquet cargados correctamente.")
    print("\n Shapes tras cargar splits:")
    print(f"   • X_train_sint: {X_train_sint.shape}, y_train: {y_train.shape}")
    print(f"   • X_val_sint:   {X_val_sint.shape},   y_val:   {y_val.shape}")
    print(f"   • X_test_sint:  {X_test_sint.shape},  y_test:  {y_test.shape}")

except Exception as e:
    print(f"\n Ocurrió un error inesperado al cargar los datos: {e}")
    print("   Asegúrate de que los archivos 'X_..._17.parquet' existen en la carpeta 'dat
```

```
Datos .parquet cargados correctamente.

Shapes tras cargar splits:
   • X_train_sint: (1976, 14), y_train: (1976,)
   • X_val_sint:   (424, 14),   y_val:   (424,)
   • X_test_sint:  (424, 14),  y_test:  (424,)
```

3. Remapear etiquetas a dos clases y guardar resultados

Define la función `remap_to_2` para convertir las etiquetas originales en dos clases binarias, aplica el remapeo a los conjuntos de entrenamiento, validación y prueba, muestra la distribución y guarda los nuevos archivos en `FINAL_SPLITS_DIR`.

```
# REMAPEAR ETIQUETAS A 2 CLASES Y GUARDAR

def remap_to_2(x):
    if x in [1.0, 2.0]:
        return 0.0 # Clase 0: Bajo/Medio Riesgo
```

```
        else:
            return 1.0 # Clase 1: Alto Riesgo

    y_train_2 = y_train.map(remap_to_2)
    y_val_2   = y_val.map(remap_to_2)
    y_test_2  = y_test.map(remap_to_2)

    print("\nDistribución original (4 clases) en y_train:")
    print(y_train.value_counts(normalize=True).rename("proporción"))
    print("\nDistribución binaria (2 clases) en y_train_2:")
    print(y_train_2.value_counts(normalize=True).rename("proporción"))

    y_train_2.to_frame().to_parquet(FINAL_SPLITS_DIR / 'y_train_2_classes.parquet')
    y_val_2.to_frame().to_parquet(FINAL_SPLITS_DIR / 'y_val_2_classes.parquet')
    y_test_2.to_frame().to_parquet(FINAL_SPLITS_DIR / 'y_test_2_classes.parquet')

    print("\n------------------------------------------------")
    print(" Mapeo a 2 clases guardado correctamente.")
    print(f"Archivos guardados en la carpeta: {FINAL_SPLITS_DIR}")
    print("------------------------------------------------")
```

```
    Distribución original (4 clases) en y_train:
    B10
    3.0    0.550101
    2.0    0.273279
    4.0    0.095142
    1.0    0.081478
    Name: proporción, dtype: float64

    Distribución binaria (2 clases) en y_train_2:
    B10
    1.0    0.645243
    0.0    0.354757
    Name: proporción, dtype: float64

    ------------------------------------------------
    ✅ Mapeo a 2 clases guardado correctamente.
    Archivos guardados en la carpeta: /content/drive/MyDrive/Digitech/TFG/ML/Calculo-Ries
    ------------------------------------------------
```

```
    # ------------------------------
    #    NO usar SMOTE para no alterar la distribución real.
    #    En su lugar, entrenaremos con `class_weight='balanced'`
    #    para que el modelo compense la clase minoritaria.
    # ------------------------------
    # (Simplemente dejamos las tablas X_train_sint, y_train_2 con proporción ≈35/65)

    print("\nShapes para entrenamiento realista (sin SMOTE):", X_train_sint.shape, y_train_2.
    print("Distribución en entrenamiento (sin SMOTE):")
    print(y_train_2.value_counts(normalize=True).rename("proporción"))
```

```
    Shapes para entrenamiento realista (sin SMOTE): (1976, 14) (1976,)
    Distribución en entrenamiento (sin SMOTE):
```

```
B10
1.0    0.645243
0.0    0.354757
Name: proporción, dtype: float64
```

5. Definir pipelines de clasificación binaria

Crea pipelines que combinan estandarización (`StandardScaler`) y clasificadores binarios (`LogisticRegression` , `RandomForestClassifier` , `GradientBoostingClassifier`) con manejo de `class_weight` cuando procede.

```
# Definir pipelines binarios (estandarizar + clasificador con class_weight)

# 1. Logistic Regression (binario, class_weight balanceado)
pipe2_lr = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression(
        solver='liblinear',
        class_weight='balanced',
        random_state=42,
        max_iter=1000
    ))
])

# 2. Random Forest (binario, class_weight balanceado)
pipe2_rf = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', RandomForestClassifier(
        class_weight='balanced',
        n_estimators=100,
        random_state=42
    ))
])

# 3. Gradient Boosting (binario)
#     (GBM no soporta class_weight directamente, pero al validar sobre datos reales
#      veremos su comportamiento; si se requiere, podríamos usar sample_weight)
pipe2_gb = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', GradientBoostingClassifier(
        n_estimators=100,
        random_state=42
    ))
])
```

6. Entrenar modelos binarios sin SMOTE

Entrena cada pipeline sobre el conjunto `X_train_sint` y las etiquetas `y_train_2` (sin aplicar SMOTE), e imprime el estado de entrenamiento de cada modelo.

```
# Entrenar cada modelo sobre X_train_sint, y_train_2 (sin SMOTE)

print("\nEntrenando modelos binarios (distribución real, sin SMOTE)...")

pipe2_lr.fit(X_train_sint, y_train_2)
print(" • LogisticRegression entrenada.")

pipe2_rf.fit(X_train_sint, y_train_2)
print(" • RandomForest entrenada.")

pipe2_gb.fit(X_train_sint, y_train_2)
print(" • GradientBoosting entrenada.")
```

```
Entrenando modelos binarios (distribución real, sin SMOTE)...
 • LogisticRegression entrenada.
 • RandomForest entrenada.
 • GradientBoosting entrenada.
```

7. Ajustar umbrales de clasificación basados en validación

Define funciones para optimizar el umbral de decisión tanto para la clase “alto riesgo” como para la clase “bajo riesgo” en el modelo `RandomForest`, y calcula ambos umbrales usando el conjunto de validación.

```
# Ajustar umbral para datos reales (opcional pero recomendado)

# --- FUNCIÓN ORIGINAL: Optimiza para Clase 1 (Alto Riesgo) ---
def find_best_threshold(pipe, X_val, y_val):
    prob_val = pipe.predict_proba(X_val)[:,1]
    best_t, best_f1 = 0.5, 0
    for t in np.linspace(0.1, 0.9, 33):
        yv_pred_t = (prob_val >= t).astype(int)
        f1_t = f1_score(y_val, yv_pred_t, pos_label=1) # F1 para la clase 1
        if f1_t > best_f1:
            best_f1, best_t = f1_t, t
    return best_t, best_f1

# --- NUEVA FUNCIÓN: Optimiza para Clase 0 (Bajo Riesgo) ---
def find_best_threshold_for_class_0(pipe, X_val, y_val):
    """
    Encuentra el umbral de probabilidad que maximiza el F1-Score para la CLASE 0.
    Un umbral más alto hará que el modelo sea más 'estricto' para predecir la Clase 1,
    protegiendo así a la Clase 0.
    """
    prob_val = pipe.predict_proba(X_val)[:,1] # Probabilidades para la clase 1
    best_t, best_f1_0 = 0.5, 0
    # Buscamos en un rango amplio de umbrales
    for t in np.linspace(0.1, 0.9, 81):
        yv_pred_t = (prob_val >= t).astype(int)
        # Calculamos el F1-Score específicamente para la clase 0 (pos_label=0)
```

```
            f1_t_0 = f1_score(y_val, yv_pred_t, pos_label=0)
            if f1_t_0 > best_f1_0:
                best_f1_0, best_t = f1_t_0, t
        return best_t, best_f1_0

    # --- CÁLCULO DE AMBOS UMBRALES ---

    # 1. Umbral original (maximizando F1 para Alto Riesgo)
    best_t_rf, best_f1_rf = find_best_threshold(pipe2_rf, X_val_sint, y_val_2)
    print(f"→ Threshold original para RF (maximizando F1 Alto Riesgo): {best_t_rf:.2f}, con F

    # 2. NUEVO umbral invertido (maximizando F1 para Bajo Riesgo)
    #    CORRECCIÓN: Usamos el pipeline 'pipe2_rf' que ya está definido y entrenado.
    best_t_rf_inverted, best_f1_rf_0 = find_best_threshold_for_class_0(pipe2_rf, X_val_sint,
    print(f"→ NUEVO Threshold para RF (maximizando F1 Bajo Riesgo): {best_t_rf_inverted:.2f},
```

```
→ Threshold original para RF (maximizando F1 Alto Riesgo): 0.40, con F1_val: 0.8025
→ NUEVO Threshold para RF (maximizando F1 Bajo Riesgo): 0.72, con F1_val: 0.5891
```

8. Definir función de evaluación binaria

Implementa `evaluate_binary_realistic`, que evalúa un pipeline binario en validación y prueba, imprimiendo métricas clave (accuracy, F1, AUC) y mostrando las matrices de confusión.

```
# Función de evaluación binaria (usando threshold = 0.5 o el óptimo encontrado)

def evaluate_binary_realistic(pipe, X_val, y_val, X_test, y_test, name, threshold=0.5):
    print(f"\n--- {name} sobre VALIDATION (2 clases, distrib real) ---")
    prob_val = pipe.predict_proba(X_val)[:,1]
    yv_pred  = (prob_val >= threshold).astype(int)
    print(f"Threshold usado: {threshold:.2f}")
    print(f"Accuracy:  {accuracy_score(y_val, yv_pred):.4f}")
    print(f"F1:        {f1_score(y_val, yv_pred):.4f}")
    print(f"AUC:       {roc_auc_score(y_val, prob_val):.4f}")
    print("\nClassification Report (val):\n", classification_report(y_val, yv_pred))
    cm_val = confusion_matrix(y_val, yv_pred)
    print("Confusion Matrix (val):\n", pd.DataFrame(cm_val,
          index=['True 0','True 1'], columns=['Pred 0','Pred 1']))

    print(f"\n--- {name} sobre TEST (2 clases, distrib real) ---")
    prob_test = pipe.predict_proba(X_test)[:,1]
    yt_pred   = (prob_test >= threshold).astype(int)
    print(f"Threshold usado: {threshold:.2f}")
    print(f"Accuracy:  {accuracy_score(y_test, yt_pred):.4f}")
    print(f"F1:        {f1_score(y_test, yt_pred):.4f}")
    print(f"AUC:       {roc_auc_score(y_test, prob_test):.4f}")
    print("\nClassification Report (test):\n", classification_report(y_test, yt_pred))
    cm_test = confusion_matrix(y_test, yt_pred)
    print("Confusion Matrix (test):\n", pd.DataFrame(cm_test,
          index=['True 0','True 1'], columns=['Pred 0','Pred 1']))
    print("-"*60)
```

9. Evaluar y comparar modelos

Ejecuta la evaluación de los modelos básicos y compara dos estrategias de umbral para `RandomForest` (detector de alto riesgo vs. protector de bajo riesgo), mostrando los resultados en consola.

```
# Evaluar todos los modelos y comparar estrategias de RandomForest

print("--- ANÁLISIS DE MODELOS BASE ---")
# Evaluación de Logistic Regression (sin cambios)
evaluate_binary_realistic(pipe2_lr, X_val_sint, y_val_2, X_test_sint, y_test_2,
                          name='LogisticRegression2', threshold=0.5)

# Evaluación de Gradient Boosting (sin cambios)
evaluate_binary_realistic(pipe2_gb, X_val_sint, y_val_2, X_test_sint, y_test_2,
                          name='GradientBoosting2', threshold=0.5)


print("\n\n--- ANÁLISIS COMPARATIVO DE ESTRATEGIAS PARA RANDOMFOREST ---")
# --- Estrategia 1: Maximizar la detección de ALTO RIESGO ---
print("\n--- Estrategia Original: 'Detector de Alto Riesgo' ---")
evaluate_binary_realistic(pipe2_rf, X_val_sint, y_val_2, X_test_sint, y_test_2,
                          name='RandomForest2 (Detector)', threshold=best_t_rf)


# --- Estrategia 2: Minimizar el error en BAJO RIESGO ---
print("\n--- Estrategia Nueva: 'Protector de Bajo Riesgo' (SELECCIONADA) ---")
evaluate_binary_realistic(pipe2_rf, X_val_sint, y_val_2, X_test_sint, y_test_2,
                          name='RandomForest2 (Protector)', threshold=best_t_rf_inverted)
```

```
--- ANÁLISIS DE MODELOS BASE ---

--- LogisticRegression2 sobre VALIDATION (2 clases, distrib real) ---
Threshold usado: 0.50
Accuracy:  0.6863
F1:        0.7532
AUC:       0.7304

Classification Report (val):
              precision    recall  f1-score   support

         0.0       0.55      0.59      0.57       150
         1.0       0.77      0.74      0.75       274

    accuracy                           0.69       424
   macro avg       0.66      0.66      0.66       424
weighted avg       0.69      0.69      0.69       424

Confusion Matrix (val):
         Pred 0  Pred 1
True 0       88      62
True 1       71     203

--- LogisticRegression2 sobre TEST (2 clases, distrib real) ---
```

```
Threshold usado: 0.50
Accuracy:  0.6651
F1:        0.7321
AUC:       0.6855

Classification Report (test):
              precision    recall  f1-score   support

         0.0       0.52      0.59      0.55       150
         1.0       0.76      0.71      0.73       274

    accuracy                           0.67       424
   macro avg       0.64      0.65      0.64       424
weighted avg       0.68      0.67      0.67       424

Confusion Matrix (test):
         Pred 0  Pred 1
True 0      88      62
True 1      80     194
------------------------------------------------------------

--- GradientBoosting2 sobre VALIDATION (2 clases, distrib real) ---
Threshold usado: 0.50
Accuracy:  0.7170
F1:        0.8052
AUC:       0.7034

Classification Report (val):
              precision    recall  f1-score   support

         0.0       0.68      0.37      0.48       150
         1.0       0.73      0.91      0.81       274

    accuracy                           0.72       424
```

Conclusiones Finales

- La **Regresión Logística** con umbral fijo (0.5) sirve de baseline estable, pero presenta limitaciones en recall de la clase minoritaria.
- **Gradient Boosting** mejora ligeramente el AUC general frente a LR, aunque no supera a Random Forest en balance de clases.
- **Random Forest + umbral "Protector de Bajo Riesgo"** logra el mejor compromiso global, reduciendo falsos positivos de alto riesgo y minimizando errores en detección de bajo riesgo.
- La estrategia "Detector de Alto Riesgo" aumenta recall de la clase alta pero eleva los falsos negativos de la clase baja, haciendo trade-off inadecuado para este caso.
- La optimización de umbrales basada en validación es esencial para ajustar el comportamiento del modelo a objetivos de negocio y riesgo financiero.
- Los thresholds calculados (`best_t_rf` y su inverso) confirman que apartarse de 0.5 mejora las métricas en escenarios realistas.

- El pipeline final recomendado es **Random Forest con threshold invertido** (“Protector de Bajo Riesgo”), por su capacidad de controlar errores de clasificación críticos.
- Este enfoque de evaluación realista y ajuste de umbral sienta una base sólida para el despliegue de clasificadores en entornos financieros sensibles.