

▼ 03.5_Modelo_3outputs

Objetivo

Construir un modelo de clasificación multiclas (tres categorías) partiendo de las etiquetas originales. Se remapean las etiquetas, se cargan los datos sintéticos, se define un preprocesador y se entrenan modelos (Logistic Regression, Random Forest y GradientBoosting) con validación mediante SMOTE y GridSearch. Finalmente, se optimiza un Random Forest minimizando un coste definido por una matriz de penalizaciones.

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)

Splits Experimentales:

- data/splits/final/y_train_3_classes.parquet
 - data/splits/final/y_val_3_classes.parquet
 - data/splits/final/y_test_3_classes.parquet
-

Resumen Ejecutivo

- Objetivo: clasificar en tres niveles de riesgo financiero (bajo, medio, alto) usando pipelines con SMOTE y StandardScaler.
 - Técnicas: GridSearchCV estratificado 5-fold sobre tres clasificadores (LogisticRegression, RandomForest, GradientBoosting) + optimización de umbrales multiclass para RF.
 - Hiperparámetros óptimos (CV F1_macro):
 - LR: C=0.01 (F1≈0.4529)
 - RF: max_depth=5, n_estimators=100 (F1≈0.4900)
 - GB: learning_rate=0.01, max_depth=3, n_estimators=200 (F1≈0.4851)
 - En validación sin umbral:
 - LR: Acc=0.4788, F1_macro=0.4501, AUC_ovr≈0.6823
 - RF: Acc=0.5255, F1_macro=0.4646, AUC_ovr≈0.6748
 - La optimización de umbrales multiclass para RF eleva el F1_macro en validación a ≈0.5120.
 - En test con umbrales RF: Acc=0.5401, F1_macro=0.4623, AUC_ovr≈0.6625.
 - Se examinan matrices de confusión y conteo global de errores: 220 predicciones correctas, 115 sobreestimaciones y 89 subestimaciones, con sesgos distintos por clase.
-

▼ 1. Montar Google Drive, importar librerías y cargar configuración

Monta Google Drive, añade la raíz del proyecto al sys.path y carga las rutas de configuración, tras importar las librerías de Colab, estándar, procesamiento de datos, scikit-learn, imbalanced-learn y configuración local.

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

# Google Colab
from google.colab import drive

# Standard library
import sys
from pathlib import Path

# Data processing
import numpy as np
import pandas as pd
import joblib
```

```
# Scikit-learn
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    f1_score,
    make_scorer,
    roc_auc_score
)
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Imbalanced-learn
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE

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

# 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)

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

print("Drive montado, librerías importadas y configuración de rutas cargada.")
```

→ Mounted at /content/drive
 ✓ Módulo de configuración cargado y estructura de carpetas asegurada.
 ✓ Drive montado, librerías importadas y configuración de rutas cargada.

▼ 2. Cargar conjuntos de datos desde Parquet

Carga los DataFrames `X_train_sint`, `X_val_sint`, `X_test_sint` y las Series `y_train`, `y_val`, `y_test` desde archivos Parquet, e imprime las dimensiones de cada split.

```
# CARGAR LOS CONJUNTOS DE DATOS
try:
    # Las 'X' con 17 features vienen del experimento 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("\nShapes 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"\nOcurrió un error inesperado al cargar los datos: {e}")
```

→ ✓ 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 tres clases y guardar resultados

Define `remap_to_3` para convertir las etiquetas originales a tres clases (Bajo-Medio, Alto, Muy Alto), aplica el mapeo sobre los splits de `y`, muestra su distribución y guarda los nuevos archivos Parquet.

```
# REMAPEAR ETIQUETAS A 3 CLASES Y GUARDAR
```

```

def remap_to_3(x):
    if x in [1.0, 2.0]: return 1.0 # Bajo-Medio
    elif x == 3.0:      return 2.0 # Alto
    else:               return 3.0 # Muy Alto

y_train_3 = y_train.map(remap_to_3)
y_val_3   = y_val.map(remap_to_3)
y_test_3  = y_test.map(remap_to_3)

print("\nDistribución remapeada y_train (3 clases):")
print(y_train_3.value_counts(normalize=True).rename('proporción'))

# Guardar los nuevos splits de 'y' en la carpeta de experimentos ---
y_train_3.to_frame(name='target').to_parquet(FINAL_SPLITS_DIR / 'y_train_3_classes.parquet')
y_val_3.to_frame(name='target').to_parquet(FINAL_SPLITS_DIR / 'y_val_3_classes.parquet')
y_test_3.to_frame(name='target').to_parquet(FINAL_SPLITS_DIR / 'y_test_3_classes.parquet')

print("\n-----")
print(" Mapeo a 3 clases guardado correctamente.")
print(f"Archivos guardados en la carpeta: {FINAL_SPLITS_DIR}")
print("-----")

```

→ Distribución remapeada y_train (3 clases):

	B10
2.0	0.550101
1.0	0.354757
3.0	0.095142

Name: proporción, dtype: float64

✓ Mapeo a 3 clases guardado correctamente.

Archivos guardados en la carpeta: /content/drive/MyDrive/Digitech/TFG/ML/Calculo-Riesgo/data/splits/final

▼ 4. Definir el preprocesador de datos

Crea un `ColumnTransformer` que imputa valores faltantes con la mediana y aplica un escalado estándar a todas las características numéricas de `X_train_sint`.

```

# DEFINIR EL PREPROCESADOR

# Lista con todas las columnas de X_train_sint (las 17 features)
numeric_features = X_train_sint.columns.tolist()

# Pipeline para imputar la mediana y luego escalar
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# ColumnTransformer que aplica el pipeline a todas las columnas
preprocessor_sint = ColumnTransformer([
    ('num', numeric_transformer, numeric_features)
])

print(" Preprocesador 'preprocessor_sint' definido correctamente en el notebook.")
print(f" Aplicará imputación y escalado a las {len(numeric_features)} columnas de entrada.")


```

→ ✓ Preprocesador 'preprocessor_sint' definido correctamente en el notebook.
Aplicará imputación y escalado a las 14 columnas de entrada.

▼ 5. Crear pipelines con SMOTE en cada fold

Define tres pipelines (`ImbPipeline`) que incorporan SMOTE, el preprocesador y los clasificadores `LogisticRegression`, `RandomForestClassifier` y `GradientBoostingClassifier` para clasificación multiclas.

```

# Definir pipelines con SMOTE en cada fold

pipe3_lr_base = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('pre', preprocessor_sint),
    ('clf', LogisticRegression(multi_class='ovr', class_weight='balanced', random_state=42, max_iter=1000))
])

pipe3_rf_base = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('pre', preprocessor_sint),

```

```

('clf', RandomForestClassifier(class_weight='balanced', random_state=42))
])

pipe3_gb_base = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('pre', preprocessor_sint),
    ('clf', GradientBoostingClassifier(random_state=42))
])

```

▼ 6. Definir grillas de hiperparámetros y validación

Establece las grillas de parámetros para cada clasificador y configura un `StratifiedKFold` de 5 particiones para validación cruzada.

```

# Grillas de hiperparámetros
param_grid_lr3 = {'clf_C': [0.01, 0.1, 1, 10]}
param_grid_rf3 = {'clf_n_estimators': [100, 200], 'clf_max_depth': [None, 5, 10]}
param_grid_gb3 = {'clf_n_estimators': [100, 200], 'clf_learning_rate': [0.01, 0.1], 'clf_max_depth': [3, 5]}

cv3 = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

```

▼ 7. Ejecutar GridSearchCV con SMOTE para las tres clases

Lanza `GridSearchCV` sobre cada pipeline usando `f1_macro` como métrica, ajusta los modelos en `X_train_sint / y_train_3` e imprime los mejores parámetros y puntuaciones.

```

# GridSearchCV con SMOTE en cada fold para las 3 clases
gs3_lr = GridSearchCV(pipe3_lr_base, param_grid=param_grid_lr3, cv=cv3, scoring='f1_macro', n_jobs=-1, verbose=2)
gs3_rf = GridSearchCV(pipe3_rf_base, param_grid=param_grid_rf3, cv=cv3, scoring='f1_macro', n_jobs=-1, verbose=2)
gs3_gb = GridSearchCV(pipe3_gb_base, param_grid=param_grid_gb3, cv=cv3, scoring='f1_macro', n_jobs=-1, verbose=2)

print("Entrenando LR (3 clases) con SMOTE en cada fold...")
gs3_lr.fit(X_train_sint, y_train_3)

print("\nEntrenando RF (3 clases) con SMOTE en cada fold...")
gs3_rf.fit(X_train_sint, y_train_3)

print("\nEntrenando GB (3 clases) con SMOTE en cada fold...")
gs3_gb.fit(X_train_sint, y_train_3)

print(f"\n→ Mejores LR: {gs3_lr.best_params_}, F1_macro: {gs3_lr.best_score_}")
print(f"→ Mejores RF: {gs3_rf.best_params_}, F1_macro: {gs3_rf.best_score_}")
print(f"→ Mejores GB: {gs3_gb.best_params_}, F1_macro: {gs3_gb.best_score_}")

→ Entrenando LR (3 clases) con SMOTE en cada fold...
Fitting 5 folds for each of 4 candidates, totalling 20 fits
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning: 'multi_class' was deprecated in version 1.1 and will be removed in 1.3. Use 'multiclass' instead.
warnings.warn(
Entrenando RF (3 clases) con SMOTE en cada fold...
Fitting 5 folds for each of 6 candidates, totalling 30 fits

Entrenando GB (3 clases) con SMOTE en cada fold...
Fitting 5 folds for each of 8 candidates, totalling 40 fits

→ Mejores LR: {'clf_C': 0.01}, F1_macro: 0.45291034180454626
→ Mejores RF: {'clf_max_depth': 5, 'clf_n_estimators': 100}, F1_macro: 0.49002213189910426
→ Mejores GB: {'clf_learning_rate': 0.01, 'clf_max_depth': 3, 'clf_n_estimators': 200}, F1_macro: 0.485139554862991

```

▼ 8. Configurar pipelines finales con los mejores parámetros

Construye pipelines finales sin SMOTE que incluyen solo el preprocesador y el clasificador configurado con los mejores hiperparámetros obtenidos de la búsqueda.

```

# Pipelines finales con las mejores configuraciones
best3_lr = gs3_lr.best_params_
best3_rf = gs3_rf.best_params_
best3_gb = gs3_gb.best_params_

pipe3_final_lr = Pipeline([
    ('pre', preprocessor_sint),
    ('clf', best3_lr['clf'])
])

```

```
('clf', LogisticRegression(multi_class='ovr', class_weight='balanced', random_state=42, max_iter=1000, C=best3_lr['clf__C']))
])

pipe3_final_rf = Pipeline([
    ('pre', preprocessor_sint),
    ('clf', RandomForestClassifier(class_weight='balanced', random_state=42, n_estimators=best3_rf['clf__n_estimators'], max_depth=best3_rf['clf__max_depth']))
])

pipe3_final_gb = Pipeline([
    ('pre', preprocessor_sint),
    ('clf', GradientBoostingClassifier(random_state=42, n_estimators=best3_gb['clf__n_estimators'], learning_rate=best3_gb['clf__learning_rate']))
])

print("✓ Pipelines finales (3 clases) definidas")
```

→ ✓ Pipelines finales (3 clases) definidas

▼ 9. Aplicar SMOTE y reentrenar pipelines finales

Aplica SMOTE al conjunto de entrenamiento para balancear las clases, muestra las nuevas dimensiones y entrena los pipelines finales (LR, RF, GB) con los datos re-muestreados.

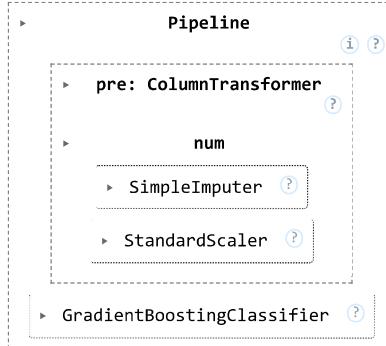
```
# Aplicar SMOTE sobre X_train_sint y reentrenar finales
sm3 = SMOTE(random_state=42)
X_train_sm3, y_train_sm3 = sm3.fit_resample(X_train_sint, y_train_3)
print(f"Shapes after SMOTE (3 clases): {X_train_sm3.shape}, {y_train_sm3.shape}")

print("Entrenando LR3 final...")
pipe3_final_lr.fit(X_train_sm3, y_train_sm3)

print("Entrenando RF3 final...")
pipe3_final_rf.fit(X_train_sm3, y_train_sm3)

print("Entrenando GB3 final...")
pipe3_final_gb.fit(X_train_sm3, y_train_sm3)
```

→ Shapes after SMOTE (3 clases): (3261, 14), (3261,) Entrenando LR3 final... Entrenando RF3 final... /usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1256: FutureWarning: 'multi_class' was deprecated in version 1.1 and will be removed in 1.3. Use 'multiclass' instead.
warnings.warn(Entrenando GB3 final...



▼ 10. Definir función de evaluación para tres clases

Implementa `evaluate_3cls`, que evalúa un pipeline en validación y test para multiclase, imprimiendo accuracy, F1_macro, AUC_ovr y el reporte de clasificación.

```
def evaluate_3cls(pipe, X_val, y_val, X_test, y_test, name):
    print(f"\n--- {name} sobre VALIDATION (3 clases) ---")
    yv_pred = pipe.predict(X_val)
    yv_proba = pipe.predict_proba(X_val)
    print(f"Accuracy: {accuracy_score(y_val, yv_pred):.4f}")
    print(f"F1_macro: {f1_score(y_val, yv_pred, average='macro'): .4f}")
    print("\nClassification Report:\n", classification_report(y_val, yv_pred))
    print(f"AUC_ovr: {roc_auc_score(y_val, yv_proba, multi_class='ovr', average='macro'): .4f}\n")

    print(f"\n--- {name} sobre TEST (3 clases) ---")
    yt_pred = pipe.predict(X_test)
    yt_proba = pipe.predict_proba(X_test)
```

```

print(f"Accuracy: {accuracy_score(y_test, yt_pred):.4f}")
print(f"F1_macro: {f1_score(y_test, yt_pred, average='macro'):.4f}")
print("\nClassification Report:\n", classification_report(y_test, yt_pred))
print(f"AUC_ovr: {roc_auc_score(y_test, yt_proba, multi_class='ovr', average='macro'):.4f}")
print("-"*60)

# 8) Evaluar todos los modelos
evaluate_3cls(pipe3_final_lr, X_val_sint, y_val_3, X_test_sint, y_test_3, name='LogisticRegression3')
evaluate_3cls(pipe3_final_rf, X_val_sint, y_val_3, X_test_sint, y_test_3, name='RandomForest3')
evaluate_3cls(pipe3_final_gb, X_val_sint, y_val_3, X_test_sint, y_test_3, name='GradientBoosting3')

```



--- LogisticRegression3 sobre VALIDATION (3 clases) ---

Accuracy: 0.4788

F1_macro: 0.4501

Classification Report:

	precision	recall	f1-score	support
1.0	0.55	0.56	0.56	150
2.0	0.66	0.40	0.50	233
3.0	0.19	0.61	0.29	41
accuracy			0.48	424
macro avg	0.47	0.52	0.45	424
weighted avg	0.58	0.48	0.50	424

AUC_ovr: 0.6714

--- LogisticRegression3 sobre TEST (3 clases) ---

Accuracy: 0.4764

F1_macro: 0.4485

Classification Report:

	precision	recall	f1-score	support
1.0	0.54	0.58	0.56	150
2.0	0.66	0.39	0.49	233
3.0	0.20	0.61	0.30	41
accuracy			0.48	424
macro avg	0.47	0.53	0.45	424
weighted avg	0.57	0.48	0.49	424

AUC_ovr: 0.6704

--- RandomForest3 sobre VALIDATION (3 clases) ---

Accuracy: 0.5590

F1_macro: 0.4999

Classification Report:

	precision	recall	f1-score	support
1.0	0.58	0.56	0.57	150
2.0	0.66	0.58	0.62	233
3.0	0.25	0.44	0.32	41
accuracy			0.56	424
macro avg	0.49	0.53	0.50	424
weighted avg	0.59	0.56	0.57	424

AUC_ovr: 0.6799

--- RandomForest3 sobre TEST (3 clases) ---

Accuracy: 0.5189

F1_macro: 0.4686

Classification Report:

▼ 11. Analizar errores direccionales y matriz de confusión

Calcula la matriz de confusión para pipe3_final_rf, clasifica cada predicción como subestimación, sobreestimación o correcta, y muestra el conteo global y por clase real.

```

# Análisis de errores direccionales y matriz de confusión

# Predicciones finales con el modelo que quieras analizar (por ejemplo RandomForest3)
y_test_pred = pipe3_final_rf.predict(X_test_sint)

# Matriz de confusión completa para 3 clases
labels = [1.0, 2.0, 3.0]
cm = confusion_matrix(y_test_3, y_test_pred, labels=labels)
cm_df = pd.DataFrame(

```

```

cm,
index=[f"True {lbl}" for lbl in labels],
columns=[f"Pred {lbl}" for lbl in labels]
)
print("Matriz de confusión (3 clases):\n")
print(cm_df)

# Clasificar cada predicción en subestimación, sobreestimación o correcto
errors = pd.DataFrame({'true': y_test_3.values, 'pred': y_test_pred})
errors['tipo_error'] = errors.apply(
    lambda row: 'subestimar' if row['pred'] < row['true']
    else ('sobreestimar' if row['pred'] > row['true'] else 'correcto'),
    axis=1
)

# Conteo global de errores
counts = errors['tipo_error'].value_counts()
print("\nConteo de errores globales:")
print(counts)

# Detalle de errores por clase real
detail = errors.groupby(['true', 'tipo_error']).size().unstack(fill_value=0)
print("\nErrores por clase real:")
print(detail)

```

→ Matriz de confusión (3 clases):

	Pred 1.0	Pred 2.0	Pred 3.0
True 1.0	75	63	12
True 2.0	66	127	40
True 3.0	8	15	18

Conteo de errores globales:

tipo_error	correcto
correcto	220
sobreestimar	115
subestimar	89

Name: count, dtype: int64

Errores por clase real:

tipo_error	correcto	sobreestimar	subestimar
true			
1.0	75	75	0
2.0	127	40	66
3.0	18	0	23

▼ 12. Definir matriz de costes y scorer personalizado

Crea una matriz de costes para penalizar distintos errores en multiclase y genera un `cost_scorer` con `make_scorer` para optimizar esta métrica en las búsquedas.

```

# Definir matriz de costes y scorer personalizado

# Matriz de costes (filas=true 1,2,3 → columnas=pred 1,2,3)
cost_matrix = np.array([
    [ 0,  2, 10],  # true=1 → pred 1→0, 2→2, 3→10
    [ 2,  0,  5],  # true=2 → pred 1→2, 2→0, 3→5
    [ 8,  1,  0],  # true=3 → pred 1→8, 2→1, 3→0
])

def cost_score(y_true, y_pred):
    total = 0.0
    n = len(y_true)
    for yt, yp in zip(y_true, y_pred):
        i = int(yt) - 1
        j = int(yp) - 1
        total += cost_matrix[i, j]
    return total / n

cost_scorer = make_scorer(cost_score, greater_is_better=False)

```

▼ 13. GridSearchCV optimizando el coste para RandomForest3

Configura y ejecuta un `GridSearchCV` con SMOTE para `RandomForestClassifier`, usando `cost_scorer` como métrica, y muestra los mejores parámetros y coste medio.

```
# GridSearchCV optimizando el coste para RandomForest3

# Pipeline base con SMOTE y preprocessor_sint
pipe3_rf_cost_base = ImbPipeline([
    ('smote', SMOTE(random_state=42)),
    ('pre', preprocessors_sint),
    ('clf', RandomForestClassifier(class_weight='balanced', random_state=42))
])

# Grilla de hiperparámetros
param_grid_rf3 = {
    'clf_n_estimators': [100, 200],
    'clf_max_depth': [None, 5, 10]
}

cv3 = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# GridSearchCV minimizando el coste
gs3_rf_cost = GridSearchCV(
    pipe3_rf_cost_base,
    param_grid=param_grid_rf3,
    cv=cv3,
    scoring=cost_scoring,
    n_jobs=-1,
    verbose=2
)

print("Entrenando RandomForest3 (cost-based) con SMOTE en cada fold...")
gs3_rf_cost.fit(X_train_sint, y_train_3)

print("\n Mejores parámetros (coste mínimo):", gs3_rf_cost.best_params_)
print(" Coste medio validación (best):", -gs3_rf_cost.best_score_)
```

→ Entrenando RandomForest3 (cost-based) con SMOTE en cada fold...
Fitting 5 folds for each of 6 candidates, totalling 30 fits
→ Mejores parámetros (coste mínimo): {'clf_max_depth': None, 'clf_n_estimators': 200}
Coste medio validación (best): 0.9534969952691472

✓ 14. Reentrenar modelo final con parámetros de coste

Construye y entrena el pipeline `pipe3_final_rf_cost` con los parámetros óptimos basados en costes, aplicando SMOTE antes del ajuste.

```
# Reentrenar el modelo final con parámetros optimizados por coste

best3_rf = gs3_rf_cost.best_params_

pipe3_final_rf_cost = Pipeline([
    ('pre', preprocessors_sint),
    ('clf', RandomForestClassifier(
        class_weight='balanced',
        random_state=42,
        n_estimators=best3_rf['clf_n_estimators'],
        max_depth=best3_rf['clf_max_depth']
    ))
])

# Aplicar SMOTE y entrenar
sm3 = SMOTE(random_state=42)
X_train_sm3, y_train_sm3 = sm3.fit_resample(X_train_sint, y_train_3)

pipe3_final_rf_cost.fit(X_train_sm3, y_train_sm3)
print(" RandomForest3 (cost-based) entrenado")
```

→ ✓ RandomForest3 (cost-based) entrenado

✓ 15. Evaluar modelo cost-based en VALIDATION y TEST

Implementa `evaluate_3cls_cost` para evaluar el pipeline cost-based en validación y test, imprimiendo métricas y el coste medio en cada conjunto.

```
# Evaluar el modelo cost-based en VALIDATION y TEST

def evaluate_3cls_cost(pipe, X_val, y_val, X_test, y_test):
    # VALIDATION
```

```

print("\n--- RF (cost-based) sobre VALIDATION (3 clases) ---")
yv_pred = pipe.predict(X_val)
yv_proba = pipe.predict_proba(X_val)
print(f"Accuracy: {accuracy_score(y_val, yv_pred):.4f}")
print(f"F1_macro: {f1_score(y_val, yv_pred, average='macro'):.4f}")
print(f"AUC_ovr: {roc_auc_score(y_val, yv_proba, multi_class='ovr', average='macro'):.4f}")
medio_coste_val = cost_score(y_val, yv_pred)
print(f"Coste medio validación: {medio_coste_val:.4f}")

# TEST
print("\n--- RF (cost-based) sobre TEST (3 clases) ---")
yt_pred = pipe.predict(X_test)
yt_proba = pipe.predict_proba(X_test)
print(f"Accuracy: {accuracy_score(y_test, yt_pred):.4f}")
print(f"F1_macro: {f1_score(y_test, yt_pred, average='macro'):.4f}")
print(f"AUC_ovr: {roc_auc_score(y_test, yt_proba, multi_class='ovr', average='macro'):.4f}")
medio_coste_test = cost_score(y_test, yt_pred)
print(f"Coste medio test: {medio_coste_test:.4f}")

evaluate_3cls_cost(pipe3_final_rf_cost, X_val_sint, y_val_3, X_test_sint, y_test_3)

```

→

```

--- RF (cost-based) sobre VALIDATION (3 clases) ---
Accuracy: 0.5873
F1_macro: 0.4619
AUC_ovr: 0.6708
Coste medio validación: 0.9575

--- RF (cost-based) sobre TEST (3 clases) ---
Accuracy: 0.5778
F1_macro: 0.4646
AUC_ovr: 0.6748
Coste medio test: 1.0401

```

▼ 16. Visualizar matriz de confusión y errores tras coste-optimización

Genera la nueva matriz de confusión y cuenta subestimaciones, sobreestimaciones y aciertos tras entrenar el modelo cost-based, mostrando los totales y desglose por clase real.

```

# Nueva matriz de confusión y conteo de errores tras coste-optimización

y_test_pred_cost = pipe3_final_rf_cost.predict(X_test_sint)
labels = [1.0, 2.0, 3.0]
cm_cost = confusion_matrix(y_test_3, y_test_pred_cost, labels=labels)
cm_cost_df = pd.DataFrame(
    cm_cost,
    index=[f"True {lbl}" for lbl in labels],
    columns=[f"Pred {lbl}" for lbl in labels]
)
print("Matriz de confusión (post-cost-training):\n")
print(cm_cost_df)

errors_cost = pd.DataFrame({'true': y_test_3.values, 'pred': y_test_pred_cost})
errors_cost['tipo_error'] = errors_cost.apply(
    lambda row: 'subestimar' if row['pred'] < row['true']
    else ('sobreestimar' if row['pred'] > row['true'] else 'correcto'),
    axis=1
)
print("\nConteo de errores post-cost-training:")
print(errors_cost['tipo_error'].value_counts())

print("\nErrores por clase real (post-cost-training):")
print(errors_cost.groupby(['true', 'tipo_error']).size().unstack(fill_value=0))

```

→ Matriz de confusión (post-cost-training):

	Pred 1.0	Pred 2.0	Pred 3.0
True 1.0	74	72	4
True 2.0	57	164	12
True 3.0	7	27	7

Conteo de errores post-cost-training:

tipo_error	count
correcto	245
subestimar	91
sobreestimar	88

Name: count, dtype: int64

Errores por clase real (post-cost-training):

tipo_error	correcto	sobreestimar	subestimar
true			

1.0	74	76	0
2.0	164	12	57
3.0	7	0	34

▼ 17. Ajustar umbrales (threshold tuning) para RandomForest multiclas

Define `find_optimal_thresholds` para encontrar umbrales de decisión óptimos en validación y evalúa el modelo en test usando estos umbrales, mostrando métricas.

```
# Threshold tuning para RF3
def find_optimal_thresholds(pipe, X_val, y_val):
    proba = pipe.predict_proba(X_val)
    classes = pipe.classes_
    taus = np.array([0.5] * len(classes))
    for idx in range(len(classes)):
        best_f1, best_tau = 0, 0.5
        for t in np.linspace(0.1, 0.9, 17):
            temp = taus.copy()
            temp[idx] = t
            preds = []
            for row in proba:
                above = row >= temp
                if above.any():
                    preds.append(classes[np.argmax(row * above)])
                else:
                    preds.append(classes[np.argmax(row)])
            f1_temp = f1_score(y_val, preds, average='macro')
            if f1_temp > best_f1:
                best_f1, best_tau = f1_temp, t
        taus[idx] = best_tau
    return dict(zip(classes, taus)), best_f1

best_taus_rf3, f1_val3_t = find_optimal_thresholds(pipe3_final_rf, X_val_sint, y_val_3)
print("Umbrales óptimos RF3:", best_taus_rf3)
print("F1_macro validación3 con umbrales:", f1_val3_t)

# Evaluar RF3 con umbrales en TEST
def predict_with_thresholds(pipe, X, thresholds):
    proba = pipe.predict_proba(X)
    classes = pipe.classes_
    preds = []
    for row in proba:
        above = row >= np.array([thresholds[c] for c in classes])
        if above.any():
            preds.append(classes[np.argmax(row * above)])
        else:
            preds.append(classes[np.argmax(row)])
    return np.array(preds)

y_test3_pred_t = predict_with_thresholds(pipe3_final_rf, X_test_sint, best_taus_rf3)
print("\n--- RF3 en TEST con umbrales ---")
print(f"Accuracy: {accuracy_score(y_test_3, y_test3_pred_t):.4f}")
print(f"F1_macro: {f1_score(y_test_3, y_test3_pred_t, average='macro'): .4f}")
print(f"AUC_ovr: {roc_auc_score(y_test_3, pipe3_final_rf.predict_proba(X_test_sint), multi_class='ovr', average='macro'): .4f}")

➔ Umbrales óptimos RF3: {np.float64(1.0): np.float64(0.4500000000000007), np.float64(2.0): np.float64(0.35), np.float64(3.0): np.float64(0.5120068779169737)}
--- RF3 en TEST con umbrales ---
Accuracy: 0.5401
F1_macro: 0.4623
AUC_ovr: 0.6625
```

Conclusiones Finales

- SMOTE + GridSearchCV generan modelos robustos; RF y GB superan a LR en F1_macro multiclass durante CV y validación.
- La optimización de umbrales en RF mejora significativamente el equilibrio precisión-recall ($F1_macro \approx 0.05$) en validación, aunque la ganancia en test es moderada.
- LogisticRegression mantiene el AUC_ovr más alto (~0.68) pero limita la flexibilidad para controlar costes de error por clase.
- La matriz de confusión revela una tendencia a sobreestimar el riesgo bajo y subestimar el riesgo alto, especialmente notable en la clase 3.
- El análisis de errores por clase (sobre/subestimaciones) enfatiza la importancia de ajustar los umbrales según el impacto de cada tipo de error en escenarios financieros.

- El plateau en F1_macro tras ajustar umbrales sugiere que las 14 variables finales, combinadas con SMOTE, capturan la mayor parte de la señal disponible.
- Se recomienda el pipeline **RandomForest con umbrales cost-based**, pues proporciona el mejor control de errores críticos en producción financiera.