```python
### SCRIPT 18 - PYTHON
# PIPELINE DE EXPORTACION DE LA TRANSFORMACION DE LOS MODELOS
# A ARCHIVOS TEXTO-MATRIZ
#
# ====================================================================
# -*- coding: utf-8 -*-

# Celda 0 — Setup + rutas + comprobación de entorno

import os, json, math, datetime, textwrap
from pathlib import Path

import numpy as np
import pandas as pd
import joblib

# Opcional: XGBoost, si está disponible en el entorno
try:
    import xgboost as xgb
    HAS_XGB = True
except Exception:
    HAS_XGB = False

# Rutas (ajusta si cambias la estructura en Drive)
MODEL_DIR = Path("/content/drive/MyDrive/_Pipeline_produccion_prediccion/modelos")
TEST_XLSX = Path("/content/drive/MyDrive/_Pipeline_produccion_prediccion/inputs/df_test_train_v8.xlsx")
OUT_DIR   = Path("/content/drive/MyDrive/_Pipeline_produccion_prediccion/modelos_portable")

OUT_DIR.mkdir(parents=True, exist_ok=True)

print("MODEL_DIR:", MODEL_DIR)
print("TEST_XLSX:", TEST_XLSX)
print("OUT_DIR  :", OUT_DIR)

assert MODEL_DIR.exists(), f"✗ MODEL_DIR no existe: {MODEL_DIR}"
assert TEST_XLSX.exists(), f"✗ No se encuentra el Excel de test: {TEST_XLSX}"

print("✅ Rutas verificadas.")
print("✅ XGBoost disponible:" if HAS_XGB else "⚠ XGBoost NO disponible; limitar export.")

"""Celda 1 — Carga de artefactos originales"""

# Celda 1 — Carga de artefactos originales

paths = {
    "preprocess_meta":   MODEL_DIR/"preprocess_meta.json",
    "preprocess":        MODEL_DIR/"preprocess_pipeline.pkl",
    "centroides":        MODEL_DIR/"centroides_pre.json",

    "rf_base":           MODEL_DIR/"rf_base.pkl",
    "rf_base_meta":      MODEL_DIR/"rf_base_meta.json",

    "rf_124":            MODEL_DIR/"rf_124_basebc.pkl",
    "rf_124_meta":       MODEL_DIR/"rf_124_basebc_meta.json",

    "xgb_pack":          MODEL_DIR/"model_Noferta_XGB_v8.pkl",
    "xgb_meta":          MODEL_DIR/"xgb_meta.json",

    "knn_train":         MODEL_DIR/"knn_pre_train.joblib",
    "knn_all":           MODEL_DIR/"knn_pre_all.joblib",
    "knn_params":        MODEL_DIR/"knn_params.json",
    "knn_num_pipe":      MODEL_DIR/"knn_num_pipeline.joblib",
    "knn_catalog":       MODEL_DIR/"knn_catalog_all.parquet",
    "knn_catalog_meta":  MODEL_DIR/"knn_catalog_meta.json",

    "ensemble_meta":     MODEL_DIR/"ensemble_meta.json",
}

# Checks mínimos (catálogo opcional pero recomendable)
```

```python
for k, p in paths.items():
    if k in ["knn_catalog", "knn_catalog_meta"]:
        continue
    assert p.exists(), f"❌ Falta {k}: {p}"

# Cargas
preprocess_meta = json.loads(paths["preprocess_meta"].read_text(encoding="utf-8"))
preprocess      = joblib.load(paths["preprocess"])
centroides      = json.loads(paths["centroides"].read_text(encoding="utf-8"))

rf_base         = joblib.load(paths["rf_base"])
rf_base_meta    = json.loads(paths["rf_base_meta"].read_text(encoding="utf-8"))

rf_124          = joblib.load(paths["rf_124"])
rf_124_meta     = json.loads(paths["rf_124_meta"].read_text(encoding="utf-8"))

xgb_pack        = joblib.load(paths["xgb_pack"])
xgb_meta        = json.loads(paths["xgb_meta"].read_text(encoding="utf-8"))

nn_train        = joblib.load(paths["knn_train"])
nn_all          = joblib.load(paths["knn_all"])
knn_params      = json.loads(paths["knn_params"].read_text(encoding="utf-8"))
knn_num_pipe    = joblib.load(paths["knn_num_pipe"])

ensemble_meta   = json.loads(paths["ensemble_meta"].read_text(encoding="utf-8"))

catalog, knn_catalog_meta, knn_id_col, knn_index_key = None, None, None, None
if paths["knn_catalog"].exists():
    catalog = pd.read_parquet(paths["knn_catalog"])
if paths["knn_catalog_meta"].exists():
    knn_catalog_meta = json.loads(paths["knn_catalog_meta"].read_text(encoding="utf-8"))
    knn_id_col = knn_catalog_meta.get("id_col")
    knn_index_key = knn_catalog_meta.get("index_key")

print("✅ Artefactos originales cargados correctamente.")
print("   RF base meta keys:", list(rf_base_meta.keys()))
print("   RF 124 meta keys :", list(rf_124_meta.keys()))
print("   XGB meta keys    :", list(xgb_meta.keys()))

"""Celda 2 — Utilidades de serialización segura a JSON"""

# Celda 2 — Utilidades de serialización segura a JSON

def _to_serializable(obj):
    """Convierte numpy/pandas a tipos básicos JSON (list, float, int, dict...)."""
    if isinstance(obj, (np.floating,)):
        return float(obj)
    if isinstance(obj, (np.integer,)):
        return int(obj)
    if isinstance(obj, (np.bool_,)):
        return bool(obj)
    if isinstance(obj, (np.ndarray,)):
        return obj.tolist()
    if isinstance(obj, (pd.Series, pd.Index)):
        return obj.tolist()
    if isinstance(obj, pd.DataFrame):
        return obj.to_dict(orient="list")
    if isinstance(obj, (dict, list, tuple)):
        if isinstance(obj, dict):
            return {k: _to_serializable(v) for k, v in obj.items()}
        else:
            return [_to_serializable(v) for v in obj]
    return obj

def save_json(obj, path: Path):
    path = Path(path)
    obj2 = _to_serializable(obj)
    path.write_text(json.dumps(obj2, indent=2, ensure_ascii=False), encoding="utf-8")
    print(f"💾 Guardado JSON: {path}")
    return path

```

```python
"""Celda 3 — Inspección del preprocesador PRE (coherencia geométrica)"""

# Celda 3 — Inspección del preprocesador PRE (coherencia geométrica)

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler

assert isinstance(preprocess, ColumnTransformer), "✗ preprocess no es ColumnTransformer."

print("Transformers en preprocess:")
for name, trans, cols in preprocess.transformers_:
    print(f"  - {name}: {type(trans).__name__}, cols={len(cols)}")

def _get_ct_block(ct: ColumnTransformer, block_name: str):
    """
    Devuelve (transformer, lista_de_columnas) para el bloque block_name
    dentro de un ColumnTransformer.
    """
    for name, trans, cols in ct.transformers_:
        if name == block_name:
            return trans, list(cols)
    raise KeyError(f"Bloque '{block_name}' no encontrado en ColumnTransformer.")

def _find_step(pipe: Pipeline, candidate_names, expected_type):
    """
    Busca un paso en un Pipeline:
    1) Primero por nombre (en orden de candidate_names).
    2) Si no, por tipo (expected_type).

    Lanza AssertionError si no encuentra nada coherente.
    """
    # 1) Por nombre
    for nm in candidate_names:
        if nm in pipe.named_steps:
            return pipe.named_steps[nm]

    # 2) Por tipo
    for nm, obj in pipe.named_steps.items():
        if isinstance(obj, expected_type):
            return obj

    raise AssertionError(
        f"No se ha encontrado en el Pipeline ningún paso con nombres {candidate_names} "
        f"ni de tipo {expected_type.__name__}."
    )

# Extraemos los bloques num y cat de forma robusta
num_trans, num_cols = _get_ct_block(preprocess, "num")
cat_trans, cat_cols = _get_ct_block(preprocess, "cat")

assert isinstance(num_trans, Pipeline), "✗ num_trans no es Pipeline."
assert isinstance(cat_trans, Pipeline), "✗ cat_trans no es Pipeline."

print("NUM cols:", num_cols)
print("CAT cols:", cat_cols)

# OJO: aquí usamos nombres reales ("imputer","scaler","ohe") pero con fallback por tipo
num_imp = _find_step(num_trans, ["imp", "imputer"], SimpleImputer)
num_sc  = _find_step(num_trans, ["sc", "scaler"],   StandardScaler)
cat_imp = _find_step(cat_trans, ["imp", "imputer"], SimpleImputer)
cat_oh  = _find_step(cat_trans, ["oh",  "ohe"],     OneHotEncoder)

for obj, name in [(num_imp, "num_imputer"), (num_sc, "num_scaler"),
                  (cat_imp, "cat_imputer"), (cat_oh, "cat_ohe")]:
    assert obj is not None, f"✗ Falta paso {name} en preprocess."

print("✅ Estructura del preprocesador PRE compatible con export.")
```

```python

"""Celda 4 — export del preprocesador"""

# Celda 4 — Export portable del preprocesador PRE + centroides

def export_preprocess_pre_portable(preprocess, centroides, preprocess_meta, out_dir: Path):
    # Usamos los helpers definidos en Celda 3
    num_trans, num_cols = _get_ct_block(preprocess, "num")
    cat_trans, cat_cols = _get_ct_block(preprocess, "cat")

    # Localizamos los pasos clave de forma robusta
    num_imp = _find_step(num_trans, ["imp", "imputer"], SimpleImputer)
    num_sc  = _find_step(num_trans, ["sc", "scaler"],   StandardScaler)
    cat_imp = _find_step(cat_trans, ["imp", "imputer"], SimpleImputer)
    cat_oh  = _find_step(cat_trans, ["oh",  "ohe"],     OneHotEncoder)

    oh_categories = {}
    for col, cats in zip(cat_cols, cat_oh.categories_):
        oh_categories[col] = [str(c) for c in cats]

    try:
        feat_out = preprocess.get_feature_names_out()
        feat_out = [str(f) for f in feat_out]
    except Exception:
        feat_out = None

    raw_contract = (
        preprocess_meta.get("expected_raw_features")
        or preprocess_meta.get("expects_raw_features")
        or {}
    )

    artifact = {
        "artifact": "preprocess_pipeline.pkl",
        "version": 1,
        "created_at": datetime.datetime.utcnow().isoformat() + "Z",
        "expected_raw_features": raw_contract,
        "numeric": {
            "columns": num_cols,
            "imputer": {
                "strategy": num_imp.strategy,
                "statistics": _to_serializable(num_imp.statistics_)
            },
            "scaler": {
                "with_mean": bool(getattr(num_sc, "with_mean", True)),
                "with_std":  bool(getattr(num_sc, "with_std", True)),
                "mean":      _to_serializable(getattr(num_sc, "mean_",  np.zeros(len(num_cols)))),
                "scale":     _to_serializable(getattr(num_sc, "scale_", np.ones(len(num_cols))))
            }
        },
        "categorical": {
            "columns": cat_cols,
            "imputer": {
                "strategy": cat_imp.strategy,
                "fill_values": _to_serializable(cat_imp.statistics_)
            },
            "onehot": {
                "handle_unknown": getattr(cat_oh, "handle_unknown", "ignore"),
                "categories": oh_categories
            }
        },
        "feature_names_out": feat_out,
        "centroids_ref": "centroides_pre.json",
        "centroids": {
            "classes": [str(c) for c in centroides.get("classes", [])],
            "feature_names": centroides.get("feature_names"),
            "centroids_pre": {str(k): _to_serializable(v) for k, v in centroides.get("centroids_pre", {}).items()},
            "p95_intra": {str(k): float(v) for k, v in centroides.get("p95_intra",
```

```
                {}).items()}
278         }
279     }
280 
281     out_path = out_dir / "preprocess_pre_portable.json"
282     save_json(artifact, out_path)
283     return out_path
284 
285 pre_pre_path = export_preprocess_pre_portable(preprocess, centroides, preprocess_meta,
     OUT_DIR)
286 
287 """Celda 5 — Export portable de RandomForest (RF base y RF_124)"""
288 
289 # Celda 5 — Export portable de RandomForest (RF base y RF_124)
290 
291 from sklearn.ensemble import RandomForestClassifier
292 from sklearn.tree import _tree
293 from sklearn.pipeline import Pipeline
294 from sklearn.compose import ColumnTransformer
295 from sklearn.impute import SimpleImputer
296 from sklearn.preprocessing import StandardScaler, OneHotEncoder
297 
298 def _extract_prep_from_pipe(pipe: Pipeline):
299     """
300     Extrae esquema de preprocesado (ColumnTransformer 'prep') de un Pipeline sklearn
301     y el clasificador RandomForest ('clf'), en formato portable.
302     """
303     assert isinstance(pipe, Pipeline), "pipe debe ser sklearn.pipeline.Pipeline"
304     prep = pipe.named_steps.get("prep")
305     clf  = pipe.named_steps.get("clf")
306     assert isinstance(prep, ColumnTransformer), "✗ paso 'prep' no es
    ColumnTransformer."
307     assert isinstance(clf, RandomForestClassifier), "✗ paso 'clf' no es
    RandomForestClassifier."
308 
309     # Localizamos bloques 'num' y 'cat' sin usar dict(...)
310     num_trans = cat_trans = None
311     num_cols = cat_cols = None
312     for name, trans, cols in prep.transformers_:
313         if name == "num":
314             num_trans = trans
315             num_cols = list(cols)
316         elif name == "cat":
317             cat_trans = trans
318             cat_cols = list(cols)
319 
320     assert num_trans is not None and num_cols is not None, "✗ Bloque 'num' no
    encontrado en prep."
321     assert cat_trans is not None and cat_cols is not None, "✗ Bloque 'cat' no
    encontrado en prep."
322     assert isinstance(num_trans, Pipeline), "✗ num_trans no es Pipeline."
323     assert isinstance(cat_trans, Pipeline), "✗ cat_trans no es Pipeline."
324 
325     # Usamos _find_step para no depender de nombres exactos ("imputer"/"scaler"/"ohe")
326     num_imp = _find_step(num_trans, ["imp", "imputer"], SimpleImputer)
327     num_sc  = _find_step(num_trans, ["sc", "scaler"],   StandardScaler)
328     cat_imp = _find_step(cat_trans, ["imp", "imputer"], SimpleImputer)
329     cat_oh  = _find_step(cat_trans, ["oh",  "ohe"],     OneHotEncoder)
330 
331     oh_categories = {}
332     for col, cats in zip(cat_cols, cat_oh.categories_):
333         oh_categories[col] = [str(c) for c in cats]
334 
335     try:
336         feat_out = prep.get_feature_names_out()
337         feat_out = [str(f) for f in feat_out]
338     except Exception:
339         feat_out = None
340 
341     prep_schema = {
342         "numeric": {
343             "columns": num_cols,
```

```
344                 "imputer": {
345                     "strategy": num_imp.strategy,
346                     "statistics": _to_serializable(num_imp.statistics_)
347                 },
348                 "scaler": {
349                     "with_mean": bool(getattr(num_sc, "with_mean", True)),
350                     "with_std":  bool(getattr(num_sc, "with_std", True)),
351                     "mean":      _to_serializable(getattr(num_sc, "mean_",  np.zeros(len(
                        num_cols)))),
352                     "scale":     _to_serializable(getattr(num_sc, "scale_", np.ones(len(
                        num_cols))))
353                 }
354             },
355             "categorical": {
356                 "columns": cat_cols,
357                 "imputer": {
358                     "strategy": cat_imp.strategy,
359                     "fill_values": _to_serializable(cat_imp.statistics_)
360                 },
361                 "onehot": {
362                     "handle_unknown": getattr(cat_oh, "handle_unknown", "ignore"),
363                     "categories": oh_categories
364                 }
365             },
366             "feature_names_out": feat_out,
367         }
368 
369         return prep_schema, clf
370 
371     def _tree_to_dict(tree: _tree.Tree):
372         """Convierte un sklearn.tree._tree.Tree en un dict portable."""
373         n_nodes = tree.node_count
374         return {
375             "n_nodes": int(n_nodes),
376             "children_left":  _to_serializable(tree.children_left),
377             "children_right": _to_serializable(tree.children_right),
378             "feature":        _to_serializable(tree.feature),
379             "threshold":      _to_serializable(tree.threshold),
380             "value":          _to_serializable(tree.value.squeeze(axis=1)),  # [n_nodes,
                n_classes]
381             "impurity":       _to_serializable(tree.impurity),
382             "n_node_samples": _to_serializable(tree.n_node_samples)
383         }
384 
385     def export_rf_portable(pipe: Pipeline, meta: dict, out_name: str, out_dir: Path):
386         """
387         Exporta un Pipeline (prep+clf) de RF a JSON portable:
388         - feature_space.expected_raw_features
389         - feature_space.alias_mapping_used_in_train
390         - feature_space.dtypes_expect
391         - esquema de preprocesado (prep_schema)
392         - bosque {trees: [...]} árbol a árbol
393         """
394         prep_schema, clf = _extract_prep_from_pipe(pipe)
395 
396         trees = []
397         for est in clf.estimators_:
398             trees.append(_tree_to_dict(est.tree_))
399 
400         classes = [str(c) for c in clf.classes_]
401 
402         feature_space = {
403             "expected_raw_features": meta.get("expected_raw_features", {}),
404             "alias_mapping_used_in_train": meta.get("alias_mapping_used_in_train", {}),
405             "dtypes_expect": meta.get("dtypes_expect", {}),
406             "preprocess": prep_schema
407         }
408 
409         artifact = {
410             "artifact": meta.get("artifact", out_name.replace(".json", ".pkl")),
411             "model_type": "RandomForestClassifier",
412             "version": 1,
```

```
413             "created_at": datetime.datetime.utcnow().isoformat() + "Z",
414             "classes": classes,
415             "n_classes": int(len(classes)),
416             "feature_space": feature_space,
417             "forest": {
418                 "n_estimators": int(clf.n_estimators),
419                 "max_depth": int(clf.max_depth) if clf.max_depth is not None else None,
420                 "n_features_in": int(getattr(clf, "n_features_in_", 0)),
421                 "trees": trees
422             }
423         }
424 
425         out_path = out_dir / out_name
426         save_json(artifact, out_path)
427         return out_path
428 
429     # Export de RF base y RF_124
430     rf_base_portable_path = export_rf_portable(rf_base, rf_base_meta,
    "rf_base_portable.json", OUT_DIR)
431     rf_124_portable_path  = export_rf_portable(rf_124, rf_124_meta, "rf_124_portable.json"
    , OUT_DIR)
432     print("✅ Export RF base:", rf_base_portable_path)
433     print("✅ Export RF 124 :", rf_124_portable_path)
434 
435     """Celda 6 — Export portable de XGBoost (baja_comp)"""
436 
437     # Celda 6 — Export portable de XGBoost (baja_comp)
438 
439     if not HAS_XGB:
440         print("⚠ XGBoost no disponible; sólo se exportarán metadatos sin booster JSON.")
441     else:
442         # Modelo entrenado y meta original
443         xgb_model = xgb_pack["modelo"]
444         xgb_vars  = xgb_meta.get("variables_predictoras", [])
445         enc_cols  = xgb_meta.get("encoders_columns", [])
446         labels    = xgb_meta.get("labels", [])
447         policies  = xgb_meta.get("policies", {})
448         threshold_bajo = policies.get("threshold_bajo", None)
449 
450         # Encoders usados en train (LabelEncoder/OrdinalEncoder normalmente)
451         label_encoders = xgb_pack.get("encoders", {})
452 
453         encoders_schema = {}
454         for col, le in label_encoders.items():
455             classes = getattr(le, "classes_", None)
456             encoders_schema[col] = {
457                 "classes": [str(c) for c in classes] if classes is not None else []
458             }
459 
460         # Meta portable = copia enriquecida del meta original
461         xgb_meta_portable = dict(xgb_meta)
462         xgb_meta_portable["created_at"] = datetime.datetime.utcnow().isoformat() + "Z"
463         xgb_meta_portable["artifact_local"] = "model_Noferta_XGB_v8.pkl"
464         xgb_meta_portable["encoders_schema"] = encoders_schema
465 
466         meta_path = OUT_DIR / "xgb_baja_portable.json"
467         save_json(xgb_meta_portable, meta_path)
468 
469         # Booster JSON (modelo en formato nativo XGBoost)
470         booster = xgb_model.get_booster()
471         booster_path = OUT_DIR / "xgb_baja_model.json"
472         booster.save_model(str(booster_path))
473         print(f"💾 Guardado booster XGB en {booster_path}")
474         print(f"💾 Guardado meta portable XGB en {meta_path}")
475 
476     """Celda 7 — Export portable de kNN (espacio PRE_NUM + matrices)"""
477 
478     # Celda 7 — Export portable de kNN (espacio PRE_NUM + matrices)
479 
480     from sklearn.neighbors import NearestNeighbors
481     from sklearn.impute import SimpleImputer
482     from sklearn.preprocessing import StandardScaler
```

```
483
484 assert isinstance(nn_train, NearestNeighbors), "❌ nn_train no es NearestNeighbors."
485 assert isinstance(nn_all, NearestNeighbors), "❌ nn_all no es NearestNeighbors."
486
487 Z_train = getattr(nn_train, "_fit_X", None)
488 Z_all   = getattr(nn_all, "_fit_X", None)
489 assert Z_train is not None, "❌ nn_train no tiene _fit_X."
490 assert Z_all is not None, "❌ nn_all no tiene _fit_X."
491
492 np.savez_compressed(OUT_DIR / "knn_train.npz", Z=Z_train)
493 np.savez_compressed(OUT_DIR / "knn_all.npz",   Z=Z_all)
494 print("🖹 Guardadas matrices knn_train.npz y knn_all.npz")
495
496 # Extraer esquema de num_pipeline de forma robusta
497 num_imp = _find_step(knn_num_pipe, ["imp", "imputer"], SimpleImputer)
498 num_sc  = _find_step(knn_num_pipe, ["sc", "scaler"],   StandardScaler)
499
500 knn_space_portable = {
501     "artifact_train": "knn_pre_train.joblib",
502     "artifact_all":   "knn_pre_all.joblib",
503     "version": 1,
504     "created_at": datetime.datetime.utcnow().isoformat() + "Z",
505     "pre_num_features": knn_params.get("pre_num_features", []),
506     "metric": knn_params.get("metric", getattr(nn_all, "metric", "euclidean")),
507     "k_neighbors": int(knn_params.get("k_neighbors", getattr(nn_all, "n_neighbors", 25
    ))),
508     "p95_nn1_train": float(knn_params.get("p95_nn1_train", 0.0)),
509     "num_pipeline": {
510         "imputer": {
511             "strategy": num_imp.strategy,
512             "statistics": _to_serializable(num_imp.statistics_)
513         },
514         "scaler": {
515             "with_mean": bool(getattr(num_sc, "with_mean", True)),
516             "with_std":  bool(getattr(num_sc, "with_std", True)),
517             "mean":      _to_serializable(getattr(num_sc, "mean_",  np.zeros(Z_train.
                shape[1]))),
518             "scale":     _to_serializable(getattr(num_sc, "scale_", np.ones(Z_train.
                shape[1])))
519         }
520     },
521     "catalog": {
522         "id_col": knn_id_col,
523         "index_key": knn_index_key,
524         "source_df": knn_catalog_meta.get("source_df") if knn_catalog_meta else None,
525         "rows": knn_catalog_meta.get("rows") if knn_catalog_meta else None,
526         "cols": knn_catalog_meta.get("cols") if knn_catalog_meta else None,
527     }
528 }
529
530 save_json(knn_space_portable, OUT_DIR / "knn_space_portable.json")
531 print("✅ Export kNN portable completo.")
532
533 """Celda 8 — Export portable de ensemble"""
534
535 # Celda 8 — Export portable de ensemble
536
537 ensemble_portable = dict(ensemble_meta)  # copia superficial del meta original
538
539 ensemble_portable["version"] = 1
540 ensemble_portable["created_at"] = datetime.datetime.utcnow().isoformat() + "Z"
541
542 # Añadimos referencias explícitas a los artefactos portables
543 depends_on_portable = {
544     "rf_base": "rf_base_portable.json",
545     "rf_124_basebc": "rf_124_portable.json",
546     "xgb_baja": "xgb_baja_portable.json" if HAS_XGB else None,
547     "preprocess_pre": "preprocess_pre_portable.json",
548     "knn_space": "knn_space_portable.json",
549     "centroids": "centroides_pre.json",
550 }
551
```

```
ensemble_portable["depends_on_portable"] = depends_on_portable

save_json(ensemble_portable, OUT_DIR / "ensemble_portable.json")
print("✅ Export ensemble_portable.json")

"""Celda 9 — Test de equivalencia completo para RF base (portable vs original)"""

# Celda 9 — Test de equivalencia completo para RF base (portable vs original, FIX float32)

from collections import OrderedDict
import numpy as np
import pandas as pd
import datetime

# ---------------------------------------------------------------------
# 1. Cargamos datos de test
# ---------------------------------------------------------------------
df_full = pd.read_excel(TEST_XLSX)
assert "Dataset" in df_full.columns, "❌ Falta columna Dataset en df_test_train_v8.xlsx."

df_test = df_full[df_full["Dataset"].astype(str).str.upper() == "TEST"].copy()
df_test.reset_index(drop=True, inplace=True)

print(f"Filas en TEST: {len(df_test)}")

# ---------------------------------------------------------------------
# 2. Cargamos modelos portables RF
# ---------------------------------------------------------------------
rf_base_port = json.loads((OUT_DIR/"rf_base_portable.json").read_text(encoding="utf-8"))
rf_124_port  = json.loads((OUT_DIR/"rf_124_portable.json").read_text(encoding="utf-8"))

rf_base_clf = rf_base.named_steps["clf"]
rf_124_clf  = rf_124.named_steps["clf"]

# ---------------------------------------------------------------------
# 3. Helpers de inferencia portable (preprocesado + árboles)
# ---------------------------------------------------------------------

def _portable_preprocess_df(df_raw: pd.DataFrame, prep_schema: dict) -> np.ndarray:
    """
    Aplica el preprocesado descrito en prep_schema (tal y como se exportó del ColumnTransformer):
      - numeric: imputación + escalado
      - categorical: imputación + one-hot

    En categóricas detecta automáticamente si las categorías son numéricas
    (caso 'Mes_lici' = 1..12) para evitar errores tipo "1" vs "1.0".
    """
    # --- Numérico ---
    num_conf = prep_schema["numeric"]
    num_cols = list(num_conf["columns"])
    if num_cols:
        X_num = df_raw[num_cols].apply(pd.to_numeric, errors="coerce")
        X_num = X_num.to_numpy(dtype=float)

        stats = np.array(num_conf["imputer"]["statistics"], dtype=float)
        assert stats.shape[0] == X_num.shape[1], "Dimensión de estadísticas numéricas no cuadra."

        for j in range(X_num.shape[1]):
            mask = np.isnan(X_num[:, j])
            if mask.any():
                X_num[mask, j] = stats[j]

        sc_conf = num_conf["scaler"]
        with_mean = bool(sc_conf.get("with_mean", True))
        with_std  = bool(sc_conf.get("with_std", True))
        mean = np.array(sc_conf.get("mean", np.zeros_like(stats)), dtype=float)
```

```
            scale = np.array(sc_conf.get("scale", np.ones_like(stats)), dtype=float)

            Xn = X_num.copy()
            if with_mean:
                Xn = Xn - mean
            if with_std:
                scale_safe = np.where(scale == 0, 1.0, scale)
                Xn = Xn / scale_safe
        else:
            Xn = np.zeros((len(df_raw), 0), dtype=float)

        # --- Categórico ---
        cat_conf = prep_schema["categorical"]
        cat_cols = list(cat_conf["columns"])
        if cat_cols:
            fill_values = list(cat_conf["imputer"]["fill_values"])
            assert len(fill_values) == len(cat_cols), "Dimensión de fill_values no cuadra con columnas categóricas."

            cat_onehot_conf = cat_conf["onehot"]
            categories_map = cat_onehot_conf["categories"]  # dict {col: [cats...]}

            Xc_blocks = []
            for j, col in enumerate(cat_cols):
                s = df_raw[col].copy()

                # Imputación "most_frequent" según estadísticas guardadas
                fill_val = fill_values[j]
                s = s.where(~s.isna(), other=fill_val)

                cats = categories_map.get(col, [])
                if not cats:
                    Xc_j = np.zeros((len(s), 0), dtype=float)
                    Xc_blocks.append(Xc_j)
                    continue

                # Detectamos si todas las categorías son numéricas (caso Mes_lici, etc.)
                numeric_cats = True
                cats_num = []
                for c in cats:
                    try:
                        cats_num.append(float(c))
                    except (TypeError, ValueError):
                        numeric_cats = False
                        break

                n = len(s)
                if numeric_cats:
                    # Col categórica codificada como número: trabajamos en float y comparamos numéricamente
                    s_num = pd.to_numeric(s, errors="coerce")
                    s_arr = s_num.to_numpy(dtype=float)
                    cats_arr = np.array(cats_num, dtype=float)

                    Xc_j = np.zeros((n, len(cats_arr)), dtype=float)
                    for k, cat_val in enumerate(cats_arr):
                        mask = (s_arr == cat_val)
                        Xc_j[mask, k] = 1.0

                else:
                    # Col categórica puramente textual: trabajamos en str
                    s_str = s.astype(str)
                    cats_str = [str(c) for c in cats]

                    Xc_j = np.zeros((n, len(cats_str)), dtype=float)
                    val_to_index = {v: idx for idx, v in enumerate(cats_str)}
                    for i, v in enumerate(s_str):
                        idx = val_to_index.get(v, None)
                        if idx is not None:
                            Xc_j[i, idx] = 1.0

                Xc_blocks.append(Xc_j)
```

```
688
689             Xc = np.concatenate(Xc_blocks, axis=1) if Xc_blocks else np.zeros((len(df_raw
                ), 0), dtype=float)
690         else:
691             Xc = np.zeros((len(df_raw), 0), dtype=float)
692
693         # Orden final: primero num, luego cat (igual que ColumnTransformer con bloques
            ["num","cat"])
694         X_pre = np.concatenate([Xn, Xc], axis=1)
695         return X_pre
696
697     def _tree_predict_proba_single(X: np.ndarray, tree_dict: dict, n_classes: int) -> np.
    ndarray:
698         """
699         Recorre un árbol individual (exportado) para todos los samples de X y devuelve
700         una matriz [n_samples, n_classes] con las probabilidades de ese árbol.
701
702         IMPORTANTE: emula el comportamiento de sklearn casteando a float32
703         antes del descenso, porque los árboles internamente comparan en float32.
704         """
705         children_left  = np.array(tree_dict["children_left"], dtype=int)
706         children_right = np.array(tree_dict["children_right"], dtype=int)
707         feature        = np.array(tree_dict["feature"], dtype=int)
708         threshold      = np.array(tree_dict["threshold"], dtype=float)
709         value          = np.array(tree_dict["value"], dtype=float)  # [n_nodes, n_classes]
710
711         # Aquí está la clave para reproducir exactamente sklearn:
712         X32  = X.astype(np.float32, copy=False)
713         thr32 = threshold.astype(np.float32, copy=False)
714
715         n_samples = X32.shape[0]
716         proba = np.zeros((n_samples, n_classes), dtype=float)
717
718         for i in range(n_samples):
719             node = 0
720             # Descenso hasta hoja
721             while children_left[node] != -1:  # _tree.TREE_LEAF == -1
722                 f = feature[node]
723                 t = thr32[node]
724                 x_val = X32[i, f]
725                 # Misma regla que sklearn, pero en float32
726                 if x_val <= t:
727                     node = children_left[node]
728                 else:
729                     node = children_right[node]
730
731             counts = value[node]
732             tot = counts.sum()
733             if tot <= 0:
734                 proba[i, :] = 1.0 / n_classes
735             else:
736                 proba[i, :] = counts / tot
737
738         return proba
739
740     def rf_portable_predict_proba(df_raw: pd.DataFrame, rf_port: dict) -> (np.ndarray, np.
    ndarray):
741         """
742         Calcula predict_proba con el modelo RF portable:
743           - Reconstruye el preprocesado desde rf_port["feature_space"]["preprocess"].
744           - Recorre todos los árboles del bosque.
745         Devuelve (probas, classes) donde:
746           - probas: [n_samples, n_classes]
747           - classes: array de etiquetas en el orden de columnas de probas.
748         """
749         feature_space = rf_port["feature_space"]
750         prep_schema   = feature_space["preprocess"]
751         expected = feature_space.get("expected_raw_features", {})
752         num_cols = expected.get("num", [])
753         cat_cols = expected.get("cat", [])
754         all_cols = list(num_cols) + list(cat_cols)
755
```

```
    missing = [c for c in all_cols if c not in df_raw.columns]
    assert not missing, f"❌ Faltan columnas esperadas en df_raw para RF portable: {missing}"

    df_in = df_raw[all_cols].copy()

    # Preprocesado portable
    X_pre = _portable_preprocess_df(df_in, prep_schema)

    forest = rf_port["forest"]
    trees = forest["trees"]
    n_estimators = int(forest["n_estimators"])
    assert n_estimators == len(trees), "❌ n_estimators no cuadra con nº de árboles exportados."

    classes = np.array(rf_port["classes"])
    n_classes = int(rf_port["n_classes"])
    assert n_classes == len(classes), "❌ n_classes no cuadra con longitud de classes."

    proba_sum = np.zeros((X_pre.shape[0], n_classes), dtype=float)
    for t_dict in trees:
        proba_tree = _tree_predict_proba_single(X_pre, t_dict, n_classes)
        proba_sum += proba_tree

    proba = proba_sum / n_estimators
    return proba, classes

def rf_portable_predict(df_raw: pd.DataFrame, rf_port: dict) -> np.ndarray:
    proba, classes = rf_portable_predict_proba(df_raw, rf_port)
    idx_max = np.argmax(proba, axis=1)
    return classes[idx_max]

# ---------------------------------------------------------------------
# 4. Equivalencia RF base (predict + predict_proba)
# ---------------------------------------------------------------------

# Columnas esperadas por RF base según meta portable
fs_base = rf_base_port["feature_space"]
exp_base = fs_base.get("expected_raw_features", {})
num_base = exp_base.get("num", [])
cat_base = exp_base.get("cat", [])
cols_base = list(num_base) + list(cat_base)

missing_base = [c for c in cols_base if c not in df_test.columns]
if missing_base:
    raise ValueError(f"❌ Faltan columnas requeridas por RF base en df_test: {missing_base}")

df_rf_base = df_test[cols_base].copy()

# Predicciones originales (Pipeline sklearn)
proba_orig_base = rf_base.predict_proba(df_rf_base)
pred_orig_base  = rf_base.predict(df_rf_base)
classes_orig_base = rf_base_clf.classes_

# Predicciones portables (a partir del JSON)
proba_port_base, classes_port_base = rf_portable_predict_proba(df_rf_base, rf_base_port)
pred_port_base = rf_portable_predict(df_rf_base, rf_base_port)

# Alineamos posibles tipos en las labels
classes_orig_base_str = np.array([str(c) for c in classes_orig_base])
classes_port_base_str = np.array([str(c) for c in classes_port_base])

assert np.array_equal(classes_orig_base_str, classes_port_base_str), \
    "❌ El orden/clases de RF base original vs portable no coincide."

# Comprobaciones de equivalencia
pred_equal_base = np.array_equal(
    np.array([str(p) for p in pred_orig_base]),
    np.array([str(p) for p in pred_port_base])
```

```
)

max_abs_diff_base = float(np.max(np.abs(proba_orig_base - proba_port_base)))

# Tolerancia muy estricta (debería ser prácticamente 0 tras el FIX)
PROBA_TOL = 1e-12
proba_close_base = bool(max_abs_diff_base <= PROBA_TOL)

print(f"RF base — pred_equal: {pred_equal_base}, proba_max_abs_diff: {
max_abs_diff_base:.3e}")

rf_base_strict_equiv_ok = bool(pred_equal_base and proba_close_base)

# ---------------------------------------------------------------------
# 5. Metadatos de coherencia y placeholders para el resto
# ---------------------------------------------------------------------
report = OrderedDict()
report["created_at"]  = datetime.datetime.utcnow().isoformat() + "Z"
report["n_test_rows"] = int(len(df_test))

# Metadatos RF base / RF 124
report["rf_base_meta_ok"] = (
    len(rf_base_port["forest"]["trees"]) == int(getattr(rf_base_clf, "n_estimators", -
    1))
    and sorted(rf_base_port["classes"]) == sorted([str(c) for c in rf_base_clf.
    classes_])
)

report["rf_124_meta_ok"] = (
    len(rf_124_port["forest"]["trees"]) == int(getattr(rf_124_clf, "n_estimators", -1
    ))
    and sorted(rf_124_port["classes"]) == sorted([str(c) for c in rf_124_clf.classes_
    ])
)

# XGB: comprobamos sólo que el booster portable se puede cargar
if HAS_XGB:
    try:
        booster_loaded = xgb.Booster()
        booster_loaded.load_model(str(OUT_DIR/"xgb_baja_model.json"))
        report["xgb_booster_load_ok"] = True
    except Exception as e:
        report["xgb_booster_load_ok"] = False
        report["xgb_booster_load_error"] = str(e)
else:
    report["xgb_booster_load_ok"] = None

# Resultados detallados RF base
report["rf_base_strict_equiv_ok"] = rf_base_strict_equiv_ok
report["rf_base_pred_equal"]      = bool(pred_equal_base)
report["rf_base_proba_max_abs_diff"] = max_abs_diff_base
report["rf_base_proba_tol"]       = PROBA_TOL

# RF 124 y pipeline completo -> pendientes de implementar (de momento)
report["rf_124_strict_equiv_ok"]  = None
report["full_pipeline_equiv_ok"]  = None

# Criterio de OK global (de momento sólo RF base + coherencia básica)
report["ok"] = bool(
    report["rf_base_meta_ok"]
    and rf_base_strict_equiv_ok
    and (report["xgb_booster_load_ok"] in [True, None])
)

equiv_path = OUT_DIR / "equivalence_report.json"
save_json(report, equiv_path)
print("📄 Equivalence report actualizado en:", equiv_path)
print("✅ OK global (parcial, centrado en RF base):", report["ok"])
```