```python
### SCRIPT 17 - PYTHON
# PIPELINE EJECUCION DE MODELOS EN PYTHON
# PIPELINE DE PREDICCION CON ARCHIVOS PKL
#
# ====================================================================
# -*- coding: utf-8 -*-

# ============================================================
# PRODUCCIÓN (limpio) — Setup + carga de artefactos y utils
# ============================================================
import os, json, re, unicodedata
import numpy as np
import pandas as pd
from pathlib import Path
import joblib

# Rutas
BASE_DIR    = Path("/content/drive/MyDrive/_Pipeline_produccion_prediccion")
DIR_MODELOS = BASE_DIR / "modelos"
DIR_INPUTS  = BASE_DIR / "inputs"
DIR_OUTPUTS = BASE_DIR / "outputs"
for d in [DIR_MODELOS, DIR_INPUTS, DIR_OUTPUTS]:
    d.mkdir(parents=True, exist_ok=True)

# Artefactos requeridos
paths = {
    "preprocess_meta":   DIR_MODELOS/"preprocess_meta.json",
    "preprocess":        DIR_MODELOS/"preprocess_pipeline.pkl",
    "centroides":        DIR_MODELOS/"centroides_pre.json",
    "rf_base":           DIR_MODELOS/"rf_base.pkl",
    "rf_base_meta":      DIR_MODELOS/"rf_base_meta.json",
    "rf_124":            DIR_MODELOS/"rf_124_basebc.pkl",
    "rf_124_meta":       DIR_MODELOS/"rf_124_basebc_meta.json",
    "xgb_pack":          DIR_MODELOS/"model_Noferta_XGB_v8.pkl",
    "xgb_meta":          DIR_MODELOS/"xgb_meta.json",
    "knn_train":         DIR_MODELOS/"knn_pre_train.joblib",
    "knn_all":           DIR_MODELOS/"knn_pre_all.joblib",
    "knn_params":        DIR_MODELOS/"knn_params.json",
    "knn_num_pipe":      DIR_MODELOS/"knn_num_pipeline.joblib",
    "knn_catalog":       DIR_MODELOS/"knn_catalog_all.parquet",
    "knn_catalog_meta":  DIR_MODELOS/"knn_catalog_meta.json",
    "ensemble_meta":     DIR_MODELOS/"ensemble_meta.json",
}

# Checks mínimos (dejamos catálogo como opcional)
for k, p in paths.items():
    if k in ["knn_catalog","knn_catalog_meta"]:
        continue
    assert p.exists(), f"✗ Falta {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"))

# Catálogo (opcional pero recomendado)
catalog, id_col = None, None
if paths["knn_catalog"].exists():
    catalog = pd.read_parquet(paths["knn_catalog"])
    for cand in ["Identificador","identificador","ID","Id","id_lote","id_contrato"]:
        if cand in catalog.columns:
```

```
            id_col = cand
            break

# Helpers de normalización de texto
def _strip_accents(s: str) -> str:
    s = unicodedata.normalize("NFKD", s)
    return "".join(ch for ch in s if not unicodedata.combining(ch))
def _norm_text(x):
    if pd.isna(x): return x
    s = str(x).strip().lower()
    s = _strip_accents(s)
    s = re.sub(r"\s+", " ", s)
    return s

# Mapa meses ES/EN -> 1..12
MES_MAP = {
    "1":"1","enero":"1","ene":"1","january":"1","jan":"1",
    "2":"2","febrero":"2","feb":"2","february":"2",
    "3":"3","marzo":"3","mar":"3","march":"3",
    "4":"4","abril":"4","abr":"4","april":"4","apr":"4",
    "5":"5","mayo":"5","may":"5",
    "6":"6","junio":"6","jun":"6","june":"6",
    "7":"7","julio":"7","jul":"7","july":"7",
    "8":"8","agosto":"8","ago":"8","august":"8","aug":"8",
    "9":"9","septiembre":"9","setiembre":"9","sep":"9","september":"9",
    "10":"10","octubre":"10","oct":"10","october":"10",
    "11":"11","noviembre":"11","nov":"11","november":"11",
    "12":"12","diciembre":"12","dic":"12","december":"12","dec":"12"
}

# POST variables que se reportarán
POST_PREFS = {
    "N_ofertantes": ["N_ofertantes"],
    "Precio_adjudicacion": ["Precio_adjudicacion","Importe_adjudicacion_lote",
    "Importe_adjudicacion"],
    "Baja_p": ["Baja_p","Baja_percent","Baja_porcentaje","Baja_p_c"],
    "Modi_eco_p": ["Modi_eco_p","Modificacion_economica_p","Modif_eco_p"],
    "Precio_final_e_c": ["Precio_final_e_c","Importe_final_e_c","Importe_final_c"],
    "Ahorro_final": ["Ahorro_final","Ahorro_total","Ahorro_final_c"]
}
def resolve_post_columns(df):
    mapping = {}
    for pref, cands in POST_PREFS.items():
        found = next((c for c in cands if c in df.columns), None)
        mapping[pref] = found
    return mapping
POST_MAP = resolve_post_columns(catalog) if catalog is not None else {k:None for k in
POST_PREFS}

"""Celda 1 — Validación/normalización común de inputs (solo PRE)"""

# ============================================================
# VALIDACIÓN (común) — aliases, Mes_lici, Provincia2, tipos
# ============================================================
def _derive_provincia2(cp):
    if pd.isna(cp): return np.nan
    try:
        s = str(int(cp)).zfill(5)
    except Exception:
        s = re.sub(r"\D", "", str(cp)).zfill(5)
    return s[:2]

def validate_normalize_df(df_in: pd.DataFrame, strict: bool=True):
    df = df_in.copy()
    msgs = []

    # Aliases numéricos (preferidos)
    if "Criterio_precio_p" not in df.columns and "C_precio_p" in df.columns:
        df["Criterio_precio_p"] = df["C_precio_p"]
        msgs.append({"row":"*", "field":"Criterio_precio_p","level":"info","code":
        "alias","hint":"C_precio_p→Criterio_precio_p"})
    if "Plazo_m" not in df.columns and "Plazo_m_c" in df.columns:
```

```
        df["Plazo_m"] = df["Plazo_m_c"]
        msgs.append({"row":"*", "field":"Plazo_m","level":"info","code":"alias","hint":"Plazo_m_c→Plazo_m"})

    # Provincia2 desde CP si falta
    if "Provincia2" not in df.columns and "Codigo_Postal_c" in df.columns:
        df["Provincia2"] = df["Codigo_Postal_c"].apply(_derive_provincia2)
        msgs.append({"row":"*", "field":"Provincia2","level":"info","code":"derive","hint":"Derivado de Codigo_Postal_c"})

    # Mes_lici (1..12)
    if "Mes_lici" in df.columns:
        old = df["Mes_lici"].copy()
        def _map_mes(x):
            if pd.isna(x): return np.nan
            s = _norm_text(x)
            return int(MES_MAP[s]) if s in MES_MAP else np.nan
        df["Mes_lici"] = df["Mes_lici"].apply(_map_mes)
        bad = df["Mes_lici"].isna() & ~old.isna()
        for i in np.where(bad)[0]:
            msgs.append({"row": int(i), "field":"Mes_lici", "level":"error" if strict else "warning",
                         "code":"month_unmapped", "original": old.iloc[i], "hint":"Usa 1-12 o ene/enero/..."})
    else:
        df["Mes_lici"] = np.nan
        msgs.append({"row":"*", "field":"Mes_lici","level":"warning","code":"missing","hint":"Se imputará"})

    # Enforce tipos numéricos básicos
    NUM_USED = ["N_lotes","Criterio_precio_p","N_CPV","N_clasi_empresa",
                "Presupuesto_licitacion_lote_c","Plazo_m","C_juicio_valor_p_c",
                "Intervalo_lici_d_c"]
    for col in NUM_USED:
        if col not in df.columns: df[col] = np.nan
        df[col] = pd.to_numeric(df[col], errors="coerce")

    # Categóricas a string (se codificarán en adaptadores)
    CAT_USED = ["Tipo_de_contrato_c","Tipo_de_Administracion_c","Tipo_de_procedimiento_c",
                "Tramitacion_c","Tipo_ganador_lote_c","Provincia2"]
    for col in CAT_USED:
        if col not in df.columns: df[col] = pd.NA
        df[col] = df[col].astype("string")

    # Dataframe normalizado de salida
    KEEP_COLS = sorted(set(NUM_USED + CAT_USED + ["Mes_lici","Provincia2","Codigo_Postal_c","C_precio_p","Plazo_m_c"]))
    df_norm = df[[c for c in KEEP_COLS if c in df.columns]].copy()

    # Persistencia del reporte (opcional)
    (DIR_OUTPUTS / "report_validacion.json").write_text(json.dumps(msgs, indent=2, ensure_ascii=False), encoding="utf-8")
    return df_norm, msgs

"""Celda 2 — Adaptador y predicción de XGB baja_comp (con políticas de xgb_meta.json)"""

# ============================================================
# XGB — baja_comp (p_bajo y flag) siguiendo xgb_meta.json
# ============================================================
from sklearn.preprocessing import LabelEncoder

def predict_baja_comp_xgb(df_norm: pd.DataFrame):
    meta = xgb_meta["policies"]
    xgb_model   = xgb_pack["modelo"]
    xgb_vars    = xgb_meta["variables_predictoras"]
    encoders    = xgb_pack.get("encoders", {})
    labels      = xgb_meta.get("labels", ["BAJO","MEDIO","ALTO"])
    idx_bajo    = labels.index("BAJO") if "BAJO" in labels else 0
    th_bajo     = float(meta.get("threshold_bajo", 0.38))

```

```
    # Construcción exacta de X (en orden)
    X = pd.DataFrame(index=df_norm.index)
    for col in xgb_vars:
        if col in df_norm.columns:
            X[col] = df_norm[col]
        else:
            # Política de faltantes numéricos: fill_0
            X[col] = 0

    # Encoders de categóricas: fallback_to_mode por columna
    fallback_map = meta.get("fallback_class_por_col", {})
    for col, le in encoders.items():
        if col in X.columns:
            s = X[col].astype(str)
            # Si valor no en repertorio, sustituir por fallback de la columna
            classes = set(map(str, le.classes_))
            fb = str(fallback_map.get(col, next(iter(classes))))
            s = s.where(s.isin(classes), other=fb)
            X[col] = le.transform(s)
        else:
            # columna prevista pero ausente -> 0
            X[col] = 0

    # Mes_lici a int si está
    if "Mes_lici" in X.columns and not pd.api.types.is_integer_dtype(X["Mes_lici"]):
        X["Mes_lici"] = pd.to_numeric(X["Mes_lici"], errors="coerce").fillna(0).astype
        (int)

    proba = xgb_model.predict_proba(X[xgb_vars])
    p_bajo = proba[:, idx_bajo]
    baja_comp = (p_bajo >= th_bajo).astype(int)
    return pd.Series(baja_comp, index=X.index, name="baja_comp"), pd.Series(p_bajo,
    index=X.index, name="p_bajo")

"""Celda 3 — Adaptador y predicción RF base (usando rf_base_meta.json)"""

#
==========================================================================================
=
# RF base — Adaptador ESTRICTO (alias de train + orden + dtypes) + diagnóstico OHE
# Sustituye o coloca esta celda tras la actual "Celda 3 — RF base" (redefine
predict_rf_base)
# Requiere: rf_base, rf_base_meta, DIR_OUTPUTS ya cargados (Celda 0)
#
==========================================================================================
=
from typing import Tuple, List, Dict, Optional
import numpy as np
import pandas as pd
import json
from datetime import datetime

def _ensure_train_used_aliases(df_norm: pd.DataFrame, rf_meta: dict) -> Tuple[pd.
DataFrame, List[dict]]:
    """
    Garantiza que existan en df_norm las columnas EXACTAS con las que se entrenó el
    RF base
    (según rf_base_meta['alias_mapping_used_in_train']). Si faltan, intenta crearlas
    desde
    su 'preferred' o desde alias comunes (C_precio_p <-> Criterio_precio_p, Plazo_m
    <-> Plazo_m_c).
    """
    msgs: List[dict] = []
    out = df_norm.copy()

    alias_train: Dict[str, str] = rf_meta.get("alias_mapping_used_in_train", {}) or {}
    # Fallbacks bidireccionales típicos
    fallback_alias = {
        "C_precio_p": ["Criterio_precio_p"],
        "Criterio_precio_p": ["C_precio_p"],
        "Plazo_m_c": ["Plazo_m"],
        "Plazo_m": ["Plazo_m_c"],
```

```
265         }
266
267         for preferred, used in alias_train.items():
268             # Si la columna "used" de train ya existe, nada que hacer
269             if used in out.columns:
270                 continue
271
272             # 1) si tengo la preferida, la copio a la usada por train
273             if preferred in out.columns:
274                 out[used] = out[preferred]
275                 msgs.append({"level":"info","code":"alias_materialized","used":used,"from"
                    :preferred})
276                 continue
277
278             # 2) si no, pruebo con fallbacks simétricos típicos
279             fb_cands = fallback_alias.get(used, []) + fallback_alias.get(preferred, [])
280             created = False
281             for alt in fb_cands:
282                 if alt in out.columns:
283                     out[used] = out[alt]
284                     msgs.append({"level":"info","code":"alias_fallback_materialized",
                        "used":used,"from":alt})
285                     created = True
286                     break
287
288             # 3) si sigue sin estar, lo creo vacío (NaN) y lo reporto
289             if not created:
290                 out[used] = np.nan
291                 msgs.append({
292                     "level":"warning","code":"alias_missing_filled_nan",
293                     "used":used,"hint":"Columna de entrenamiento no encontrada; creada
                        como NaN"
294                 })
295
296         return out, msgs
297
298     def _subset_exact_order_for_pipe(df_src: pd.DataFrame, rf_meta: dict) -> Tuple[pd.
    DataFrame, List[dict]]:
299         """
300         Subconjunta en el ORDEN EXACTO las columnas crudas que el ColumnTransformer
            espera.
301         Si falta alguna, la crea como NaN (y lo reporta). 'Mes_lici' se castea a category
            si aplica.
302         Convierte numéricas a tipo numérico (el imputer de mediana está dentro del pipe).
303         """
304         msgs: List[dict] = []
305
306         exp: Dict[str, List[str]] = rf_meta.get("expected_raw_features", {}) or {}
307         cols_num = list(exp.get("num", []))
308         cols_cat = list(exp.get("cat", []))
309         ordered = cols_num + cols_cat
310
311         # Construir frame con orden exacto
312         X = pd.DataFrame(index=df_src.index, columns=ordered, dtype="object")
313         for c in ordered:
314             if c in df_src.columns:
315                 X[c] = df_src[c]
316             else:
317                 X[c] = np.nan
318                 msgs.append({"level":"warning","code":"missing_train_col_created_nan",
                    "col":c})
319
320         # Tipos exactos donde importa
321         if rf_meta.get("dtypes_expect", {}).get("Mes_lici") == "category" and "Mes_lici"
            in X.columns:
322             # Igual que en entrenamiento: Int->category (permitiendo NaN)
323             X["Mes_lici"] = pd.to_numeric(X["Mes_lici"], errors="coerce").astype("Int64").
                astype("category")
324
325         # Numéricas a tipo numérico (coherente con imputer de mediana en el pipe)
326         for c in cols_num:
327             X[c] = pd.to_numeric(X[c], errors="coerce")
```

```python

    return X, msgs

def _ohe_unknowns_diagnostics(X: pd.DataFrame, rf_model) -> List[dict]:
    """
    Estima % de 'unknowns' por columna categórica frente al repertorio visto en train
    (OneHotEncoder.categories_). Si hay unknowns, OHE pondrá todo a 0 para esa
    columna.
    """
    msgs: List[dict] = []
    try:
        prep = rf_model.named_steps["prep"]
        cat_cols: List[str] = []
        oh = None
        # Localizar el bloque categórico y su OneHotEncoder
        for name, trans, cols in prep.transformers_:
            if name == "cat":
                cat_cols = list(cols) if isinstance(cols, list) else []
                if hasattr(trans, "named_steps"):  # Pipeline
                    oh = trans.named_steps.get("oh") or trans.named_steps.get("ohe")
                else:
                    oh = trans  # Por si viene el OHE "plano" (raro)
                break
        if oh is None or not hasattr(oh, "categories_"):
            return msgs

        # Alinear orden de columnas con categories_
        n = min(len(cat_cols), len(oh.categories_))
        for i in range(n):
            col = cat_cols[i]
            allowed = set(map(str, oh.categories_[i]))
            s = X[col].astype(str)
            unk = ~s.isin(allowed)
            unk_rate = float(unk.mean()) if len(s) else 0.0
            if unk_rate > 0.0:
                # muestra de desconocidos (máx 5)
                ex = s[unk].dropna().unique().tolist()[:5]
                msgs.append({
                    "level":"warning","code":"ohe_unknown_values",
                    "col": col, "unknown_rate": round(unk_rate, 4),
                    "examples": [str(x) for x in ex]
                })
    except Exception as e:
        msgs.append({"level":"info","code":"ohe_diag_skipped","reason":repr(e)})
    return msgs

def predict_rf_base(df_norm: pd.DataFrame, save_diag: bool = True, diag_tag: Optional[str] = None
                    ) -> Tuple[np.ndarray, np.ndarray, np.ndarray, List[str]]:
    """
    Adaptador ESTRICTO para RF base:
      1) materializa alias EXACTOS usados en train (según rf_base_meta)
      2) sub-conjunta columnas crudas en el ORDEN EXACTO del ColumnTransformer
      3) castea tipos (Mes_lici -> category) y numéricas a float/int
      4) diagnóstico de unknowns en OHE
    Devuelve: (pred, conf, proba, labels)  — igual interfaz que el adaptador previo.
    Si save_diag=True, guarda un JSON con el diagnóstico y un CSV snapshot de X.
    """
    # 1) alias usados en train
    df_used, m1 = _ensure_train_used_aliases(df_norm, rf_base_meta)
    # 2) subset exacto + tipos
    X, m2 = _subset_exact_order_for_pipe(df_used, rf_base_meta)
    # 3) diagnóstico OHE unknowns
    m3 = _ohe_unknowns_diagnostics(X, rf_base)

    # 4) predicción
    proba = rf_base.predict_proba(X)
    labels = [str(c) for c in rf_base.named_steps["clf"].classes_]
    pred_idx = np.argmax(proba, axis=1)
    pred = np.array([labels[i] for i in pred_idx], dtype=object)
    conf = np.array([proba[i, pred_idx[i]] for i in range(len(pred_idx))], dtype=float
    )
```

```
397
398         # 5) guardado diagnóstico
399         if save_diag:
400             ts = datetime.now().strftime("%Y%m%d_%H%M%S")
401             tag = f"_{diag_tag}" if diag_tag else ""
402             ddir = DIR_OUTPUTS / f"rf_base_diag_{ts}{tag}"
403             ddir.mkdir(parents=True, exist_ok=True)
404             # Snapshot del input exacto que entra al RF
405             X.to_csv(ddir / "rf_base_input_snapshot.csv", index=False)
406             # Meta/diagnóstico
407             diag_payload = {
408                 "messages": m1 + m2 + m3,
409                 "expected_raw_features": rf_base_meta.get("expected_raw_features", {}),
410                 "alias_mapping_used_in_train": rf_base_meta.get(
                    "alias_mapping_used_in_train", {}),
411                 "notes":"unknown_rate>0 en alguna categórica indica valores no vistos en
                    train"
412             }
413             with open(ddir/"rf_base_diag.json","w",encoding="utf-8") as f:
414                 json.dump(diag_payload, f, indent=2, ensure_ascii=False)
415             print(f"📎 RF base: diagnóstico guardado en {ddir}")
416
417         return pred, conf, proba, labels
418
419     """Celda 4 — Adaptador y predicción RF_124_BASE+BC (usa baja_comp y contrato
    rf_124_basebc_meta.json)"""
420
421     #
    ================================================================================
    =
422     # RF_124_BASE+BC — Adaptador ESTRICTO (alias de train + orden + dtypes + duplicadas
    BC)
423     # Reemplaza o coloca tras tu Celda 4 (redefine predict_rf124)
424     # Requiere: rf_124, rf_124_meta, DIR_OUTPUTS ya cargados (Celda 0) y baja_comp
    (Series)
425     #
    ================================================================================
    =
426     from typing import Tuple, List, Dict, Optional
427     import numpy as np, pandas as pd, json
428     from datetime import datetime
429
430     def _ensure_train_used_aliases_124(df_norm: pd.DataFrame, rf124_meta: dict) -> Tuple[
    pd.DataFrame, List[dict]]:
431         """
432         Materializa en df_norm las columnas EXACTAS con las que se entrenó el RF_124
            (según
433         rf_124_basebc_meta['alias_mapping_used_in_train']). Si faltan, intenta crearlas
            desde
434         su 'preferred' o alias comunes (C_precio_p <-> Criterio_precio_p, Plazo_m <->
            Plazo_m_c).
435         """
436         msgs: List[dict] = []
437         out = df_norm.copy()
438
439         alias_train: Dict[str, str] = rf124_meta.get("alias_mapping_used_in_train", {}) or
             {}
440         fallback_alias = {
441             "C_precio_p": ["Criterio_precio_p"],
442             "Criterio_precio_p": ["C_precio_p"],
443             "Plazo_m_c": ["Plazo_m"],
444             "Plazo_m": ["Plazo_m_c"],
445         }
446
447         for preferred, used in alias_train.items():
448             if used in out.columns:
449                 continue
450             if preferred in out.columns:
451                 out[used] = out[preferred]
452                 msgs.append({"level":"info","code":"alias_materialized","used":used,"from"
                    :preferred})
453                 continue
```

```
454             fb_cands = fallback_alias.get(used, []) + fallback_alias.get(preferred, [])
455             created = False
456             for alt in fb_cands:
457                 if alt in out.columns:
458                     out[used] = out[alt]
459                     msgs.append({"level":"info","code":"alias_fallback_materialized",
                        "used":used,"from":alt})
460                     created = True
461                     break
462             if not created:
463                 out[used] = np.nan
464                 msgs.append({"level":"warning","code":"alias_missing_filled_nan","used":
                    used})
465         return out, msgs
466
467     def _subset_exact_order_for_pipe_124(df_src: pd.DataFrame, rf124_meta: dict) -> Tuple[
        pd.DataFrame, List[dict]]:
468         """
469         Subconjunta en el ORDEN EXACTO las columnas crudas que el ColumnTransformer del
            RF_124 espera.
470         Si falta alguna, la crea como NaN (y lo reporta). 'Mes_lici' se castea a category
            si aplica.
471         Convierte numéricas a tipo numérico (el imputer de mediana está dentro del pipe).
472         """
473         msgs: List[dict] = []
474         exp: Dict[str, List[str]] = rf124_meta.get("expected_raw_features", {}) or {}
475         cols_num = list(exp.get("num", []))
476         cols_cat = list(exp.get("cat", []))
477         ordered = cols_num + cols_cat
478
479         X = pd.DataFrame(index=df_src.index, columns=ordered, dtype="object")
480         for c in ordered:
481             if c in df_src.columns:
482                 X[c] = df_src[c]
483             else:
484                 X[c] = np.nan
485                 msgs.append({"level":"warning","code":"missing_train_col_created_nan",
                    "col":c})
486
487         # Tipos exactos
488         if rf124_meta.get("alias_mapping_used_in_train", {}) is not None and "Mes_lici" in
             X.columns:
489             X["Mes_lici"] = pd.to_numeric(X["Mes_lici"], errors="coerce").astype("Int64").
                astype("category")
490         for c in cols_num:
491             X[c] = pd.to_numeric(X[c], errors="coerce")
492
493         return X, msgs
494
495     def _attach_baja_comp_dups(X: pd.DataFrame, baja_comp: pd.Series, rf124_meta: dict) ->
         Tuple[pd.DataFrame, List[dict]]:
496         """
497         Inserta 'baja_comp' (categoría) y sus duplicadas EXACTAS tal como se usaron en
            entrenamiento
498         (rf_124_meta['baja_comp_dup_names']).
499         """
500         msgs: List[dict] = []
501         X = X.copy()
502         # base
503         X["baja_comp"] = pd.Categorical(baja_comp)
504         if "baja_comp" not in list(X.columns):
505             msgs.append({"level":"warning","code":"baja_comp_missing_in_expected","hint":
                "Se añadió 'baja_comp' aunque no estuviera en expected_raw_features"})
506         # duplicadas
507         dup_names = rf124_meta.get("baja_comp_dup_names", [])
508         for dn in dup_names:
509             X[dn] = X["baja_comp"]
510         # Nota: si el pipe esperaba las duplicadas en expected_raw_features.cat, ya están
            presentes por nombre exacto.
511         return X, msgs
512
513     def _ohe_unknowns_diagnostics_124(X: pd.DataFrame, rf124_model) -> List[dict]:
```

```
    """
    Diagnóstico de unknowns por columna categórica frente al repertorio de OHE del especialista.
    """
    msgs: List[dict] = []
    try:
        prep = rf124_model.named_steps["prep"]
        cat_cols: List[str] = []
        oh = None
        for name, trans, cols in prep.transformers_:
            if name == "cat":
                cat_cols = list(cols) if isinstance(cols, list) else []
                if hasattr(trans, "named_steps"):
                    oh = trans.named_steps.get("oh") or trans.named_steps.get("ohe")
                else:
                    oh = trans
                break
        if oh is None or not hasattr(oh, "categories_"):
            return msgs

        n = min(len(cat_cols), len(oh.categories_))
        for i in range(n):
            col = cat_cols[i]
            allowed = set(map(str, oh.categories_[i]))
            s = X[col].astype(str)
            unk = ~s.isin(allowed)
            rate = float(unk.mean()) if len(s) else 0.0
            if rate > 0.0:
                ex = s[unk].dropna().unique().tolist()[:5]
                msgs.append({"level":"warning","code":"ohe_unknown_values","col":col,
                "unknown_rate":round(rate,4),"examples":[str(x) for x in ex]})
    except Exception as e:
        msgs.append({"level":"info","code":"ohe_diag_skipped","reason":repr(e)})
    return msgs

def predict_rf124(df_norm: pd.DataFrame, baja_comp: pd.Series, save_diag: bool = True,
 diag_tag: Optional[str] = None
                  ) -> Tuple[np.ndarray, np.ndarray, np.ndarray, List[str]]:
    """
    Adaptador ESTRICTO para RF_124:
      1) materializa alias EXACTOS usados en train (según rf_124_meta)
      2) sub-conjunta columnas crudas en el ORDEN EXACTO del ColumnTransformer
      3) castea tipos (Mes_lici -> category) y numéricas a float/int
      4) inserta 'baja_comp' + duplicadas EXACTAS (nombres del meta)
      5) diagnóstico de unknowns en OHE
    Devuelve: (pred, conf, proba, labels) — misma interfaz.
    """
    # 1) alias de train
    df_used, m1 = _ensure_train_used_aliases_124(df_norm, rf_124_meta)
    # 2) subset exacto + tipos
    X, m2 = _subset_exact_order_for_pipe_124(df_used, rf_124_meta)
    # 3) baja_comp y duplicadas EXACTAS
    X, m3 = _attach_baja_comp_dups(X, baja_comp, rf_124_meta)
    # 4) diagnóstico OHE
    m4 = _ohe_unknowns_diagnostics_124(X, rf_124)

    # 5) predicción
    proba = rf_124.predict_proba(X)
    labels = [str(c) for c in rf_124.named_steps["clf"].classes_]
    pred_idx = np.argmax(proba, axis=1)
    pred = np.array([labels[i] for i in pred_idx], dtype=object)
    conf = np.array([proba[i, pred_idx[i]] for i in range(len(pred_idx))], dtype=float
    )

    # 6) guardado diagnóstico
    if save_diag:
        ts = datetime.now().strftime("%Y%m%d_%H%M%S")
        tag = f"_{diag_tag}" if diag_tag else ""
        ddir = DIR_OUTPUTS / f"rf124_basebc_diag_{ts}{tag}"
        ddir.mkdir(parents=True, exist_ok=True)
        X.to_csv(ddir / "rf124_input_snapshot.csv", index=False)
        with open(ddir/"rf124_diag.json","w",encoding="utf-8") as f:
```

```
582                 json.dump({
583                     "messages": m1 + m2 + m3 + m4,
584                     "expected_raw_features": rf_124_meta.get("expected_raw_features", {}),
585                     "alias_mapping_used_in_train": rf_124_meta.get(
                        "alias_mapping_used_in_train", {}),
586                     "baja_comp_dup_names": rf_124_meta.get("baja_comp_dup_names", []),
587                     "notes":"unknowns>0 en categóricas del especialista degradan su
                        margen vs RF base y reducen flips."
588                 }, f, indent=2, ensure_ascii=False)
589             print(f"🧾 RF_124: diagnóstico guardado en {ddir}")
590 
591         return pred, conf, proba, labels
592 
593     """Celda 5 — Ensemble "seguro" (scope {1,2,4}, margen desde ensemble_meta.json) +
        segundo cluster"""
594 
595     # ============================================================
596     # ENSEMBLE — BASE vs 124 (scope {1,2,4}) + segundo cluster
597     # ============================================================
598     def apply_ensemble_from_parts(pred_base, conf_base, proba_base, labels_base,
599                                   pred_124, conf_124, proba_124, labels_124,
600                                   margin_min=None):
601         if margin_min is None:
602             margin_min = float(ensemble_meta.get("margin_min", 0.01))
603 
604         idx_b = {lb:i for i,lb in enumerate(labels_base)}
605         idx_s = {lb:i for i,lb in enumerate(labels_124)}
606 
607         final_pred = pred_base.copy()
608         fuente     = np.array(["rf_base"]*len(final_pred), dtype=object)
609         conf_final = conf_base.copy()
610         flips = np.zeros(len(final_pred), dtype=bool)
611 
612         # Flip SOLO si la clase base está en {1,2,4} y el especialista gana por margen
613         for i in range(len(final_pred)):
614             base_cls = str(pred_base[i])
615             alt_cls  = str(pred_124[i])
616             if base_cls not in CLASSES_124:
617                 continue
618             if (alt_cls in CLASSES_124) and (alt_cls != base_cls):
619                 mb = proba_base[i, idx_b[base_cls]]
620                 ma = proba_124[i, idx_s[alt_cls]]
621                 if (ma - mb) >= margin_min:
622                     final_pred[i] = alt_cls
623                     fuente[i]     = "rf_124_basebc"
624                     conf_final[i] = ma
625                     flips[i]      = True
626 
627         # Segundo cluster (si conf_final < 0.70) usando el modelo decisor
628         segundo = np.array([None]*len(final_pred), dtype=object)
629         conf2   = np.array([np.nan]*len(final_pred), dtype=float)
630         for i in range(len(final_pred)):
631             if fuente[i] == "rf_124_basebc":
632                 probs, labs = proba_124[i], labels_124
633             else:
634                 probs, labs = proba_base[i], labels_base
635             order = np.argsort(probs)[::-1]
636             top1 = labs[order[0]]
637             p1   = probs[order[0]]
638             # coherencia
639             if str(top1) != str(final_pred[i]):
640                 top1 = final_pred[i]
641                 p1   = conf_final[i]
642                 order = [np.where(np.array(labs)==top1)[0][0]] + [j for j in order if labs
                    [j]!=top1]
643             if p1 < 0.70 and len(order) > 1:
644                 sc = labs[order[1]]; p2 = probs[order[1]]
645                 segundo[i], conf2[i] = sc, float(p2)
646 
647         out = pd.DataFrame({
648             "pred": final_pred,
649             "confianza_pred": conf_final,
```

```
            "fuente_confianza": fuente,
            "segundo_cluster": segundo,
            "confianza_segundo": conf2,
            "flip_ensemble": flips
        })
        return out

    """Celda 6 — kNN similares (NUM-ONLY con su pipeline exacta) + catálogo"""

    # ============================================================
    # kNN — similares (NUM-ONLY) con pipeline exacta + catálogo
    # ============================================================
    def knn_similares(df_norm: pd.DataFrame, topk=3, sim_thr=60.0):
        pre_num_features = knn_params["pre_num_features"]
        p95_nn1_train    = float(knn_params["p95_nn1_train"])
        # Construir matriz NUM-ONLY en el ORDEN exacto y transformar con knn_num_pipeline
        X_num = pd.DataFrame(index=df_norm.index)
        for c in pre_num_features:
            if c in df_norm.columns:
                X_num[c] = pd.to_numeric(df_norm[c], errors="coerce")
            elif c == "C_precio_p" and "Criterio_precio_p" in df_norm.columns:
                X_num[c] = pd.to_numeric(df_norm["Criterio_precio_p"], errors="coerce")
            else:
                X_num[c] = np.nan
        Zq = knn_num_pipe.transform(X_num)

        dist, idx = nn_all.kneighbors(Zq, n_neighbors=max(topk,1), return_distance=True)
        def sim_percent(d):
            return max(0.0, 1.0 - float(d)/p95_nn1_train)*100.0 if p95_nn1_train>0 else
            0.0

        out_rows = []
        for i in range(len(df_norm)):
            pairs = [(float(dist[i,j]), int(idx[i,j])) for j in range(dist.shape[1])]
            sims  = [(j, sim_percent(d)) for (d,j) in pairs]
            keep  = [(j, s) for (j,s) in sims if s >= sim_thr]
            warn = False
            if len(keep) == 0:
                keep = [max(sims, key=lambda t:t[1])]
                warn = True
            keep = sorted(keep, key=lambda t:t[1], reverse=True)[:topk]
            for rank, (j,s) in enumerate(keep, start=1):
                row = {"input_row": i, "rank": rank, "knn_row": j, "sim%": round(float(s),
                2)}
                if catalog is not None:
                    meta = catalog.iloc[j]
                    row["Cluster_6"] = str(meta.get("Cluster_6",""))
                    if id_col: row["id"] = meta.get(id_col, f"row_{j}")
                    for key in ["Presupuesto_licitacion_lote_c","Plazo_m",
                    "Criterio_precio_p","N_ofertantes"]:
                        if key in catalog.columns:
                            row[key] = meta.get(key, np.nan)
                if warn: row["warning"] = "No hay similares con sim>=60%; se muestra el
                más cercano."
                out_rows.append(row)
        return pd.DataFrame(out_rows)

    """Celda 7 — POST stats para pred (y opcional segundo_cluster)"""

    # ============================================================
    # POST stats — medias y std por cluster (pred y segundo)
    # ============================================================
    def post_stats_for_clusters(pred_df: pd.DataFrame):
        if catalog is None or "Cluster_6" not in catalog.columns:
            return pd.DataFrame(), pd.DataFrame()
        groups = catalog.groupby(catalog["Cluster_6"].astype(str))
        def _stats_for(cluster):
            g = groups.get_group(str(cluster)) if str(cluster) in groups.indices else None
            rec = {}
            for pref, col in POST_MAP.items():
                if g is None or col is None:
                    rec[f"{pref}_mean"] = np.nan; rec[f"{pref}_std"] = np.nan
```

```
            else:
                rec[f"{pref}_mean"] = float(g[col].mean(numeric_only=True))
                rec[f"{pref}_std"]  = float(g[col].std(numeric_only=True))
        return rec

    post1_rows, post2_rows = [], []
    for i, row in pred_df.reset_index(drop=True).iterrows():
        cl = row["pred"]
        rec1 = {"input_row": i, "cluster": str(cl)}
        rec1.update(_stats_for(cl))
        post1_rows.append(rec1)
        sc = row.get("segundo_cluster", None)
        if pd.notna(sc):
            rec2 = {"input_row": i, "cluster": str(sc)}
            rec2.update(_stats_for(sc))
            post2_rows.append(rec2)
    return pd.DataFrame(post1_rows), pd.DataFrame(post2_rows)

"""Celda 8 — (Opcional) Coherencia geométrica via centroides (separada)"""

# ============================================================
# (OPCIONAL) Coherencia geométrica por centroides
# ============================================================
def coherencia_geometrica(df_norm: pd.DataFrame):
    Z = preprocess.transform(df_norm)  # espacio PRE (imputación+OHE+escala)
    classes  = [str(c) for c in centroides["classes"]]
    cents    = {str(k): np.array(v, dtype=float) for k,v in centroides["centroids_pre"
    ].items()}
    p95_intra= {str(k): float(v) for k,v in centroides.get("p95_intra", {}).items()}

    out=[]
    for i in range(Z.shape[0]):
        z = Z[i]
        d = {c: float(np.linalg.norm(z - cents[c])) for c in classes}
        items = sorted(d.items(), key=lambda kv: kv[1])
        cmin, dmin = items[0]
        d2 = items[1][1] if len(items)>1 else np.nan
        margin = float(d2-dmin) if np.isfinite(d2) else np.nan
        p95 = p95_intra.get(cmin, np.nan)
        coher = (1.0 - dmin/p95)*100.0 if (p95 and p95>0) else np.nan
        out.append({"min_cluster_geom": cmin, "min_dist": dmin, "geom_margin": margin,
         "coherencia_geom": coher})
    return pd.DataFrame(out)

"""Celda 9 — Orquestación: de inputs → todas las salidas (CSV por lotes)"""

# ============================================================
# ORQUESTADOR — batch: inputs CSV → outputs CSV
# ============================================================
def run_batch(csv_path: Path, topk=3, sim_thr=60.0, add_geom=False):
    df_in = pd.read_csv(csv_path) if csv_path.suffix.lower()==".csv" else pd.
    read_excel(csv_path)
    df_norm, _ = validate_normalize_df(df_in, strict=False)

    # 1) baja_comp (XGB)
    baja, p_bajo = predict_baja_comp_xgb(df_norm)

    # 2) RF base
    pred_b, conf_b, proba_b, labs_b = predict_rf_base(df_norm)

    # 3) RF_124
    pred_s, conf_s, proba_s, labs_s = predict_rf124(df_norm, baja)

    # 4) Ensemble
    ens = apply_ensemble_from_parts(pred_b, conf_b, proba_b, labs_b,
                                    pred_s, conf_s, proba_s, labs_s,
                                    margin_min=None)
    # Enriquecer con baja_comp
    ens.insert(0, "baja_comp", baja.values)
    ens.insert(1, "p_bajo", p_bajo.values)

    # 5) Similares kNN
```

```
    sims = knn_similares(df_norm, topk=topk, sim_thr=sim_thr)

    # 6) POST stats
    post1, post2 = post_stats_for_clusters(ens)

    # 7) (opcional) coherencia geométrica
    geom = None
    if add_geom:
        geom = coherencia_geometrica(df_norm)

    # Guardados
    ts = pd.Timestamp.now().strftime("%Y%m%d_%H%M%S")
    out_dir = DIR_OUTPUTS / f"pred_{ts}"
    out_dir.mkdir(parents=True, exist_ok=True)

    # Adjuntar ID si está
    id_cand = next((c for c in ["Identificador","identificador","ID","Id","id_lote",
    "id_contrato"] if c in df_in.columns), None)
    main = ens.copy()
    if id_cand:
        main.insert(0, "Identificador", df_in[id_cand].values[:len(main)])

    # Añadir POST stats (pred y segundo) en columnas
    if not post1.empty:
        sel = []
        for i in range(len(main)):
            r = post1[post1["input_row"]==i]
            rec={"input_row":i}
            if len(r):
                for pref in POST_PREFS:
                    rec[f"POST_{pref}_mean"] = float(r.iloc[0].get(f"{pref}_mean", np.
                    nan))
                    rec[f"POST_{pref}_std"]  = float(r.iloc[0].get(f"{pref}_std",  np.
                    nan))
            sel.append(rec)
        main = main.join(pd.DataFrame(sel).set_index("input_row"), how="left")
    if not post2.empty:
        sel2=[]
        for i in range(len(main)):
            r = post2[post2["input_row"]==i]
            rec={"input_row":i}
            if len(r):
                for pref in POST_PREFS:
                    rec[f"POST2_{pref}_mean"] = float(r.iloc[0].get(f"{pref}_mean", np
                    .nan))
                    rec[f"POST2_{pref}_std"]  = float(r.iloc[0].get(f"{pref}_std",  np
                    .nan))
            sel2.append(rec)
        main = main.join(pd.DataFrame(sel2).set_index("input_row"), how="left")

    main.to_csv(out_dir/"predicciones.csv", index=False)
    sims.to_csv(out_dir/"similares_topK.csv", index=False)
    post1.to_csv(out_dir/"post_stats_pred.csv", index=False)
    if not post2.empty:
        post2.to_csv(out_dir/"post_stats_segundo.csv", index=False)
    if geom is not None:
        geom.to_csv(out_dir/"coherencia_geom.csv", index=False)

    print(f"✅ Batch guardado en: {out_dir}")
    return {"main": main, "sims": sims, "post1": post1, "post2": post2, "geom": geom}

"""Celda — predict_one para GPT (JSON)"""

# ============================================================
# API para GPT — predict_one(input_dict, k=3, sim_thr=60)
# ============================================================
def predict_one(input_dict: dict, topk: int = 3, sim_thr: float = 60.0,
                include_geom: bool = False, include_validation_messages: bool = True):
    """
    Entrada: dict con SOLO variables PRE (se aceptan alias C_precio_p/Plazo_m_c,
    Mes_lici texto, etc.)
    Salida: dict JSON con:
```

```
        - baja_comp, p_bajo
        - pred, confianza_pred, fuente_confianza, (si <70%) segundo_cluster, confianza_segundo, flip_ensemble
        - post_stats_pred {var: {mean, std}} y opcional post_stats_segundo
        - similares (hasta 3 con sim>=60%; si no, 1 con warning)
        - (opcional) coherencia_geom {min_cluster_geom, min_dist, geom_margin, coherencia_geom}
        - (opcional) validation_messages (para revisar normalizaciones/fallbacks)
    """
    # 0) Validación / normalización (común)
    df_raw = pd.DataFrame([input_dict])
    df_norm, messages = validate_normalize_df(df_raw, strict=False)

    # 1) baja_comp (XGB)
    baja, p_bajo = predict_baja_comp_xgb(df_norm)

    # 2) RF base
    pred_b, conf_b, proba_b, labs_b = predict_rf_base(df_norm)

    # 3) RF_124 (especialista)
    pred_s, conf_s, proba_s, labs_s = predict_rf124(df_norm, baja)

    # 4) Ensemble (scope {1,2,4} + margen)
    ens = apply_ensemble_from_parts(pred_b, conf_b, proba_b, labs_b,
                                    pred_s, conf_s, proba_s, labs_s,
                                    margin_min=None)
    # 5) kNN similares
    sims_df = knn_similares(df_norm, topk=topk, sim_thr=sim_thr)
    sims0 = sims_df[sims_df["input_row"]==0].sort_values("rank").to_dict(orient="records")

    # 6) POST stats (pred y, si aplica, segundo)
    post_pred_df, post_second_df = post_stats_for_clusters(ens)
    post_pred = {}
    if not post_pred_df.empty:
        r = post_pred_df[post_pred_df["input_row"]==0]
        if len(r):
            post_pred = {pref: {"mean": float(r.iloc[0].get(f"{pref}_mean", np.nan)),
                                "std":  float(r.iloc[0].get(f"{pref}_std",  np.nan))}
                         for pref in POST_PREFS}
    post_second = {}
    if not post_second_df.empty and pd.notna(ens.loc[0, "segundo_cluster"]):
        r2 = post_second_df[post_second_df["input_row"]==0]
        if len(r2):
            post_second = {pref: {"mean": float(r2.iloc[0].get(f"{pref}_mean", np.nan)),
                                  "std":  float(r2.iloc[0].get(f"{pref}_std",  np.nan))}
                           for pref in POST_PREFS}

    # 7) (Opcional) coherencia geométrica
    geom = None
    if include_geom:
        geom_df = coherencia_geometrica(df_norm)
        if not geom_df.empty:
            g = geom_df.iloc[0].to_dict()
            # redondeos amigables
            g["min_dist"] = float(g.get("min_dist", np.nan))
            g["geom_margin"] = float(g.get("geom_margin", np.nan)) if pd.notna(g.get("geom_margin", np.nan)) else None
            g["coherencia_geom"] = float(g.get("coherencia_geom", np.nan)) if pd.notna(g.get("coherencia_geom", np.nan)) else None
            geom = g

    # 8) Ensamblar respuesta JSON
    rec = {
        "baja_comp": int(ens.loc[0, "baja_comp"]),
        "p_bajo": float(ens.loc[0, "p_bajo"]),
        "pred": str(ens.loc[0, "pred"]),
        "confianza_pred": float(ens.loc[0, "confianza_pred"]),
        "fuente_confianza": str(ens.loc[0, "fuente_confianza"]),
        "flip_ensemble": bool(ens.loc[0, "flip_ensemble"]),
```

```
            "similares": sims0,
            "post_stats_pred": post_pred
        }
        # segundo cluster si aplica
        if pd.notna(ens.loc[0, "segundo_cluster"]):
            rec["segundo_cluster"] = str(ens.loc[0, "segundo_cluster"])
            rec["confianza_segundo"] = float(ens.loc[0, "confianza_segundo"])
            if post_second:
                rec["post_stats_segundo"] = post_second
        # geom opcional
        if include_geom and geom is not None:
            rec["coherencia_geom"] = geom
        # mensajes de validación opcionales
        if include_validation_messages:
            # Filtrar mensajes relevantes a la fila 0 y globales ("*")
            msgs0 = [m for m in messages if m.get("row") in (0, "*")]
            rec["validation_messages"] = msgs0

        # Info de ensemble (from→to) si hubo flip
        if rec["flip_ensemble"]:
            rec["ensemble_info"] = {
                "from": str(pred_b[0]),
                "to": str(pred_s[0]),
                "margin_delta": float(
                    (proba_s[0, list(labs_s).index(str(pred_s[0]))] if str(pred_s[0]) in
                    labs_s else np.nan)
                    - (proba_b[0, list(labs_b).index(str(pred_b[0]))] if str(pred_b[0]) in
                     labs_b else np.nan)
                )
            }

        return rec

    """Celda — Prueba global (20 TEST + 2 CSV + CM/accuracy de TODO el TEST)"""

    #
    ==========================================================================================
    =
    # PRUEBA GLOBAL — 20 casos TEST + 2 CSV + evaluación completa (TEST entero)
    #
    ==========================================================================================
    =
    from sklearn.metrics import confusion_matrix, accuracy_score

    # Verificaciones mínimas
    assert callable(validate_normalize_df), "Ejecuta primero las celdas del pipeline de
    producción."
    assert callable(predict_baja_comp_xgb) and callable(predict_rf_base) and callable(
    predict_rf124)
    assert callable(apply_ensemble_from_parts) and callable(knn_similares) and callable(
    post_stats_for_clusters)

    # 0) Cargar dataset de inputs (TEST)
    cand_paths = [
        DIR_INPUTS/"df_train_test_v8.xlsx",
        DIR_INPUTS/"df_test_train_v8.xlsx",
        DIR_INPUTS/"df_train_test_v8.csv",
        DIR_INPUTS/"df_test_train_v8.csv",
    ]
    data_path = next((p for p in cand_paths if p.exists()), None)
    assert data_path is not None, f"✗ No se encontró dataset en: {cand_paths}"
    df_all = pd.read_excel(data_path) if data_path.suffix.lower() in [".xlsx",".xls"] else
     pd.read_csv(data_path)
    assert "Dataset" in df_all.columns, "✗ Falta columna 'Dataset' (TRAIN/TEST)"
    df_all["Dataset"] = df_all["Dataset"].astype(str).str.upper()
    df_test = df_all[df_all["Dataset"]=="TEST"].copy().reset_index(drop=True)
    assert len(df_test)>0, "✗ El split TEST está vacío."

    # 1) Selección reproducible de 20 casos
    rng = np.random.RandomState(42)
    idx20 = rng.choice(len(df_test), size=min(20, len(df_test)), replace=False)
    df20 = df_test.iloc[idx20].reset_index(drop=True)
```

```
980
981 # 2) Predicción para los 20 casos
982 df_norm_20, _ = validate_normalize_df(df20, strict=False)
983 baja20, p_bajo20 = predict_baja_comp_xgb(df_norm_20)
984 pb_pred20, pb_conf20, pb_proba20, pb_labs = predict_rf_base(df_norm_20)
985 ps_pred20, ps_conf20, ps_proba20, ps_labs = predict_rf124(df_norm_20, baja20)
986 ens20 = apply_ensemble_from_parts(pb_pred20, pb_conf20, pb_proba20, pb_labs,
987                                   ps_pred20, ps_conf20, ps_proba20, ps_labs,
988                                   margin_min=None)
989 ens20.insert(0, "baja_comp", baja20.values)
990 ens20.insert(1, "p_bajo", p_bajo20.values)
991
992 # POST stats pred/segundo
993 post1_20, post2_20 = post_stats_for_clusters(ens20)
994
995 # Preparar salida de 20 (outputs genéricos de la app)
996 id_col = next((c for c in ["Identificador","identificador","ID","Id","id_lote",
    "id_contrato"] if c in df20.columns), None)
997 main20 = ens20.copy()
998 if id_col:
999     main20.insert(0, "Identificador", df20[id_col].values)
1000
1001 # Anexar POST stats (pred y segundo) en columnas
1002 if not post1_20.empty:
1003     sel=[]
1004     for i in range(len(main20)):
1005         r = post1_20[post1_20["input_row"]==i]
1006         rec={"input_row":i}
1007         if len(r):
1008             for pref in POST_PREFS:
1009                 rec[f"POST_{pref}_mean"] = float(r.iloc[0].get(f"{pref}_mean", np.nan
                    ))
1010                 rec[f"POST_{pref}_std"]  = float(r.iloc[0].get(f"{pref}_std",  np.nan
                    ))
1011         sel.append(rec)
1012     main20 = main20.join(pd.DataFrame(sel).set_index("input_row"), how="left")
1013
1014 if not post2_20.empty:
1015     sel2=[]
1016     for i in range(len(main20)):
1017         r = post2_20[post2_20["input_row"]==i]
1018         rec={"input_row":i}
1019         if len(r):
1020             for pref in POST_PREFS:
1021                 rec[f"POST2_{pref}_mean"] = float(r.iloc[0].get(f"{pref}_mean", np.nan
                    ))
1022                 rec[f"POST2_{pref}_std"]  = float(r.iloc[0].get(f"{pref}_std",  np.nan
                    ))
1023         sel2.append(rec)
1024     main20 = main20.join(pd.DataFrame(sel2).set_index("input_row"), how="left")
1025
1026 # 3) Construir CSV de comparación POST pred vs real (20 casos)
1027 # Resolver mapping de columnas reales en df_all
1028 def resolve_post_columns_df(df_cols):
1029     mapping={}
1030     for pref, cands in POST_PREFS.items():
1031         found = next((c for c in cands if c in df_cols), None)
1032         mapping[pref] = found
1033     return mapping
1034 POST_MAP_DF = resolve_post_columns_df(set(df_all.columns))
1035
1036 rows=[]
1037 for i in range(len(df20)):
1038     rec={}
1039     if id_col: rec["Identificador"] = df20.loc[i, id_col]
1040     rec["pred_cluster"] = str(main20.loc[i, "pred"])
1041     rec["confianza_pred"] = float(main20.loc[i, "confianza_pred"])
1042     rec["baja_comp_pred"] = int(main20.loc[i, "baja_comp"])
1043     rec["p_bajo"] = float(main20.loc[i, "p_bajo"])
1044     # POST pred (medias del cluster)
1045     for pref in POST_PREFS:
1046         rec[f"{pref}_pred_mean"] = float(main20.loc[i, f"POST_{pref}_mean"]) if
```

```
            f"POST_{pref}_mean" in main20.columns else np.nan
            rec[f"{pref}_pred_std"]  = float(main20.loc[i, f"POST_{pref}_std"])  if
            f"POST_{pref}_std"  in main20.columns else np.nan
        # POST reales
        for pref, col in POST_MAP_DF.items():
            rec[f"{pref}_real"] = (None if col is None else df20.loc[i, col])
        # baja_comp_real derivada (si hay N_ofertantes real)
        if POST_MAP_DF["N_ofertantes"]:
            nreal = df20.loc[i, POST_MAP_DF["N_ofertantes"]]
            rec["baja_comp_real"] = (int(nreal <= 4) if pd.notna(nreal) else np.nan)
        else:
            rec["baja_comp_real"] = np.nan
        # cluster real
        rec["cluster_real"] = (str(df20.loc[i, "Cluster_6"]) if "Cluster_6" in df20.
        columns else None)
        rows.append(rec)
    targets20 = pd.DataFrame(rows)

    # 4) Evaluación del ENSEMBLE en TODO el TEST
    df_norm_test, _ = validate_normalize_df(df_test, strict=False)
    baja_t, p_bajo_t = predict_baja_comp_xgb(df_norm_test)
    pb_pred_t, pb_conf_t, pb_proba_t, pb_labs_t = predict_rf_base(df_norm_test)
    ps_pred_t, ps_conf_t, ps_proba_t, ps_labs_t = predict_rf124(df_norm_test, baja_t)
    ens_t = apply_ensemble_from_parts(pb_pred_t, pb_conf_t, pb_proba_t, pb_labs_t,
                                      ps_pred_t, ps_conf_t, ps_proba_t, ps_labs_t,
                                      margin_min=None)

    y_true = df_test["Cluster_6"].astype(str).values
    y_pred = ens_t["pred"].astype(str).values
    labels_sorted = sorted(pd.unique(pd.concat([pd.Series(y_true), pd.Series(y_pred)])),
    key=lambda s:int(s))
    cm = confusion_matrix(y_true, y_pred, labels=labels_sorted)
    acc_global = accuracy_score(y_true, y_pred)
    acc_by_cls = []
    for cl in labels_sorted:
        m = (y_true == cl)
        acc_c = float((y_pred[m]==y_true[m]).mean()) if m.any() else np.nan
        acc_by_cls.append({"cluster": cl, "accuracy": acc_c})
    acc_by_cls = pd.DataFrame(acc_by_cls)

    # 5) Guardados
    ts = pd.Timestamp.now().strftime("%Y%m%d_%H%M%S")
    out_dir = DIR_OUTPUTS / f"test_global_{ts}"
    out_dir.mkdir(parents=True, exist_ok=True)

    main20.to_csv(out_dir/"predicciones_20.csv", index=False)
    targets20.to_csv(out_dir/"targets_pred_vs_real_20.csv", index=False)
    pd.DataFrame(cm, index=[f"real_{c}" for c in labels_sorted], columns=[f"pred_{c}" for
    c in labels_sorted]).to_csv(out_dir/"cm_test.csv")
    acc_by_cls.to_csv(out_dir/"accuracy_por_cluster.csv", index=False)
    with open(out_dir/"summary_eval_test.json","w",encoding="utf-8") as f:
        json.dump({"accuracy_global": float(acc_global), "labels": labels_sorted}, f,
        indent=2, ensure_ascii=False)

    print("✅ Prueba global completada")
    print(f"→ {out_dir/'predicciones_20.csv'}")
    print(f"→ {out_dir/'targets_pred_vs_real_20.csv'}")
    print(f"→ {out_dir/'cm_test.csv'} | {out_dir/'accuracy_por_cluster.csv'} |
    acc_global={acc_global:.4f}")
```